z/Architecture

Principles of Operation

SA22-7832-11

z/Architecture

Principles of Operation

SA22-7832-11

Note:

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xvii.

Softcopy Note:

The reader should be aware of the fact that this publication contains many symbols, such as superscripts, that may not display correctly with any given hardware or software. The definitive version of this publication is the hardcopy version.

Twelfth Edition (September, 2017)

This edition obsoletes and replaces *z/Architecture Principles of Operation*, SA22-7832-10.

This publication is provided for use in conjunction with other relevant IBM publications, and IBM makes no warranty, express or implied, about its completeness or accuracy. The information in this publication is current as of its publication date but is subject to change without notice.

Additional copies of this and other IBM publications may be ordered or downloaded from the IBM publications web site at http://www.ibm.com/support/documentation.

A form for reader's comments is provided at the back of this publication. If the form has been removed, address your comments to:

International Business Machines Corporation
Department 55JA
Mail Station P384
2455 South Road
Poughkeepsie, N.Y., 12601-5400
United States of America

FAX (United States and Canada): 1-845-432-9405
FAX (Other Countries): Your International Access Code + 1-845-432-9405
IBMLink™ (United States customers only): IBMUSM10(MHVRCFS)
Internet e-mail: mhvrcfs@us.ibm.com
World Wide Web: http://www.ibm.com/servers/eserver/zseries/zos/webqs.html

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2000, 2017. All rights reserved.

US Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices xxiv
Trademarks xxiv

Preface xxvii
Size and Number Notation. xxviii
Bytes, Characters, and Codes. xxix
Other Publications xxix
Summary of Changes in Twelfth Edition xxx
Summary of Changes in Eleventh Edition xxxiv
Summary of Changes in Tenth Edition xxxvii
Summary of Changes in Ninth Edition. xxxix
Summary of Changes in Eighth Edition. xli
Summary of Changes in Seventh Edition xlii
Summary of Changes in Sixth Edition. xliii
Summary of Changes in Fifth Edition xlv
Summary of Changes in Fourth Edition. xlvii
Summary of Changes in Third Edition. xlix
Summary of Changes in Second Edition. li

Chapter 1, Introduction 1-1
Highlights of Original z/Architecture 1-2
General Instructions for 64-Bit Integers. 1-3
Other New General Instructions 1-3
Floating-Point Instructions. 1-4
Control Instructions 1-5
Trimodal Addressing 1-5
Modal Instructions 1-5
Effects on Bits 0-31 of a General Register . 1-6
Input/Output. 1-6
Additions to z/Architecture. 1-7
ASN-and-LX-Reuse Facility. 1-7
CMPSC-Enhancement Facility 1-7
Compare-and-Swap-and-Store Facility 1-7
Compare-and-Swap-and-Store Facility 2 1-8
Conditional-SSKE Facility 1-8
Configuration-Topology Facility 1-8
Configuration-z/Architecture-Architectural-Mode Facility 1-8
Constrained-Transactional-Execution Facility . 1-8
DAT-Enhancement Facility 1. 1-8
DAT-Enhancement Facility 2. 1-8
Decimal-Floating-Point Facility 1-9
Decimal-Floating-Point Packed-Conversion Facility 1-9
Decimal-Floating-Point-Rounding Facility 1-9
Decimal-Floating-Point Zoned-Conversion Facility 1-9
Distinct-Operands Facility 1-9
Enhanced-DAT Facility 1. 1-9
Enhanced-DAT Facility 2 1-10
Enhanced-Monitor Facility 1-11
Entropy Encoding Compression Facility 1-11
ESA/390-Compatibility-Mode Facility 1-11
ETF2-Enhancement Facility 1-11
ETF3-Enhancement Facility 1-11
Execute-Extensions Facility 1-11
Execution-Hint Facility 1-11
Extended-Immediate Facility. 1-12
Extended-I/O-Measurement-Block Facility . . . 1-12
Extended-I/O-Measurement-Word Facility . . . 1-12
Extended-Translation Facility 2. 1-12
Extended-Translation Facility 3. 1-13
Extract-CPU-Time Facility. 1-13
Fast-BCR-Serialization Facility 1-13
Fibre-Channel Extensions (FCX) 1-13
FCX-Bidirectional-Data-Transfer Facility 1-14
Floating-Point Extension Facility. 1-14
Floating-Point-Support-Sign- Handling Facility 1-15
FPR-GR-Transfer Facility 1-15
General-Instructions-Extension Facility 1-15
Guarded-Storage Facility 1-15
HFP Multiply-and-Add/Subtract Facility 1-16
HFP-Unnormalized-Extensions Facility 1-16
High-Word Facility. 1-16
IEEE-Exception-Simulation Facility. 1-16
Insert-Reference-Bits-Multiple Facility 1-16
Instruction-Execution-Protection Facility. 1-16
Interlocked-Access Facility 1 1-17
Interlocked-Access Facility 2 1-17
IPTE-Range Facility 1-17
List-Directed Initial Program Load. 1-17
Load-and-Trap Facility 1-17
Load-and-Zero-Rightmost-Byte Facility 1-17
Load/Store-on-Condition Facility 1 1-18
Load/Store-on-Condition Facility 2 1-18
Local-TLB-Clearing Facility. 1-18
Long-Displacement Facility. 1-18
Message-Security Assist. 1-18
Message-Security-Assist Extension 1. 1-19
Message-Security-Assist Extension 2. 1-19
Message-Security-Assist Extension 3. 1-19
Message-Security-Assist Extension 4. 1-20
Message-Security-Assist Extension 5. 1-21
Message-Security-Assist Extension 6. 1-21
Message-Security-Assist Extension 7. 1-21
Message-Security-Assist Extension 8. 1-22
Miscellaneous-Instruction-Extensions Facility 1 1-22
Miscellaneous-Instruction-Extensions Facility 2 1-22
Modified CCW Indirect Data Addressing Facility. 1-22
Move-With-Optional-Specifications Facility. . . 1-23

© Copyright IBM Corp. 2000, 2017

Multiple-Epoch Facility 1-23
Multiple-Subchannel-Set Facility 1-23
Multithreading Facility 1-23
Nonquiescing Key-Setting Facility 1-23
Parsing-Enhancement Facility 1-24
PER-3 Facility 1-24
PER Zero-Address-Detection Facility 1-24
PFPO Facility 1-24
Population-Count Facility 1-24
Processor-Assist Facility 1-24
Reset-Reference-Bits-Multiple Facility 1-25
Restore-Subchannel Facility 1-25
Server-Time-Protocol Facility 1-25
Side-Effect-Access Facility 1-25
Store-Clock-Fast Facility 1-25
Store-Facility-List-Extended Facility 1-25
Test-Pending-External-Interruption Facility . . 1-25
TOD-Clock-Steering Facility 1-26
Transactional-Execution Facility 1-26
Vector-Enhancements Facility 1 1-26
Vector Facility for z/Architecture 1-26
Vector Packed-Decimal Facility 1-26
Warning-Track Interruption Facility 1-27
The ESA/390 Base 1-27
The ESA/370 and 370-XA Base 1-32
System Program 1-34
Compatibility 1-34
Compatibility among z/Architecture Systems . 1-34
Compatibility between z/Architecture and ESA/390 1-34
Control-Program Compatibility 1-34
Problem-State Compatibility 1-35
Availability 1-35

Chapter 2, Organization 2-1
Main Storage 2-2
Expanded Storage 2-2
CPU 2-2
CPU Types 2-3
Multithreading 2-3
PSW 2-3
General Registers 2-4
Floating-Point Registers 2-4
Floating-Point-Control Register 2-5
Vector Registers 2-5
Control Registers 2-6
Access Registers 2-6
Cryptographic Facility 2-7
External Time Reference 2-7
I/O 2-7
Channel Subsystem 2-7
Channel Paths 2-7
I/O Devices and Control Units 2-8
Operator Facilities 2-8

Chapter 3, Storage 3-1
Storage Addressing 3-2
Information Formats 3-3
Integral Boundaries 3-3
Address Types and Formats 3-4
Address Types 3-4
Absolute Address 3-4
Real Address 3-4
Virtual Address 3-5
Primary Virtual Address 3-5
Secondary Virtual Address 3-5
AR-Specified Virtual Address 3-5
Home Virtual Address 3-5
Logical Address 3-5
Instruction Address 3-5
Effective Address 3-5
Address Size and Wraparound 3-6
Address Wraparound 3-7
Storage Key 3-9
Protection 3-10
Key-Controlled Protection 3-11
Storage-Protection-Override Control 3-12
Fetch-Protection-Override Control 3-13
Access-List-Controlled Protection 3-13
DAT Protection 3-13
Low-Address Protection 3-14
Instruction-Execution Protection 3-14
Suppression on Protection 3-15
Reference Recording 3-19
Change Recording 3-20
Prefixing 3-21
Prefixing in the z/Architecture Architectural Mode 3-22
Prefixing in the ESA/390-Compatibility Mode . 3-22
Common Prefixing Attributes 3-22
Address Spaces 3-23
Changing to Different Address Spaces 3-23
Address-Space Number 3-23
ASN-Second-Table-Entry Sequence Number . 3-24
ASN-Second-Table-Entry Instance Number and ASN Reuse 3-25
ASN Translation 3-30
ASN-Translation Controls 3-30
Control Register 14 3-30
ASN-Translation Tables 3-31
ASN-First-Table Entries 3-31
ASN-Second-Table Entries 3-31
ASN-Translation Process 3-33
ASN-First-Table Lookup 3-33
ASN-Second-Table Lookup 3-34

Recognition of Exceptions during ASN Translation . . . 3-35
ASN Authorization . . . 3-35
ASN-Authorization Controls . . . 3-35
Control Register 4 . . . 3-35
ASN-Second-Table Entry . . . 3-35
Authority-Table Entries . . . 3-36
ASN-Authorization Process . . . 3-36
Authority-Table Lookup . . . 3-36
Recognition of Exceptions during ASN Authorization . . . 3-38
Dynamic Address Translation . . . 3-38
Translation Control . . . 3-40
Translation Modes . . . 3-40
Control Register 0 . . . 3-41
Control Register 1 . . . 3-42
Control Register 7 . . . 3-44
Control Register 13 . . . 3-45
Translation Tables . . . 3-45
Region-Table Entries . . . 3-46
Segment-Table Entries . . . 3-49
Page-Table Entries . . . 3-51
Translation Process . . . 3-52
Inspection of Real-Space Control . . . 3-54
Inspection of Designation-Type Control . . . 3-54
Lookup in a Table Designated by an Address-Space-Control Element . . . 3-57
Lookup in a Table Designated by a Region-Table Entry . . . 3-59
Page-Table Lookup . . . 3-61
Formation of the Real and Absolute Addresses . . . 3-62
Recognition of Exceptions during Translation . . . 3-62
Translation-Lookaside Buffer . . . 3-62
TLB Structure . . . 3-63
Formation of TLB Entries . . . 3-63
Use of TLB Entries . . . 3-65
Modification of Translation Tables . . . 3-67
Address Summary . . . 3-70
Addresses Translated . . . 3-70
Handling of Addresses . . . 3-71
Assigned Storage Locations . . . 3-73
Assigned Storage Locations in the z/Architecture Architectural Mode . . . 3-73
Assigned Storage Locations in the ESA/390-Compatibility Mode . . . 3-83

Chapter 4, Control . . . 4-1
CPU States . . . 4-2
Stopped State . . . 4-2
Operating State . . . 4-3
Load State . . . 4-3
Check-Stop State . . . 4-3
Program-Status Word . . . 4-4
Program-Status-Word Format . . . 4-5
Short PSW Format . . . 4-7
Control Registers . . . 4-8
Tracing . . . 4-12
Implicit Tracing . . . 4-12
Branch Tracing . . . 4-12
ASN Tracing . . . 4-13
Mode Tracing . . . 4-13
Explicit Tracing . . . 4-14
Control-Register Allocation . . . 4-15
Trace Entries . . . 4-15
Trace Operation . . . 4-25
Program-Event Recording . . . 4-26
PER Instruction-Fetching Nullification . . . 4-26
Control-Register Allocation and Address-Space-Control Element . . . 4-26
PER Operation . . . 4-28
Identification of Cause . . . 4-29
Priority of Indication . . . 4-33
Storage-Area Designation . . . 4-34
PER Events . . . 4-35
Successful Branching . . . 4-35
Instruction Fetching . . . 4-36
Storage Alteration . . . 4-36
Store Using Real Address . . . 4-37
Zero-Address Detection . . . 4-37
Transaction End . . . 4-39
Indication of PER Events Concurrently with Other Interruption Conditions . . . 4-39
Indication of PER Events and Guarded-Storage Events . . . 4-42
Breaking-Event-Address Recording . . . 4-43
Breaking-Event-Address Register . . . 4-44
Execution-Break Instructions . . . 4-44
Timing . . . 4-46
Time-of-Day Clock and Epoch Index . . . 4-46
Format . . . 4-46
States . . . 4-47
Changes in Clock State . . . 4-48
Setting and Inspecting the Clock . . . 4-49
TOD Programmable Register . . . 4-51
TOD-Clock Synchronization . . . 4-53
Timing Mode . . . 4-53
Timing State . . . 4-53
STP Clock Source State . . . 4-54
TOD-Clock Steering . . . 4-54
Offset-Based TOD-Clock Steering Overview . . . 4-55
Hardware-Based TOD-Clock-Steering Overview . . . 4-55
TOD-Offset-Update Events . . . 4-56
Episodes . . . 4-56

TOD-Clock-Steering Registers 4-56
UTC Information Block (UIB) 4-57
Clock Comparator 4-59
CPU Timer 4-62
Guarded-Storage Facility 4-64
Guarded-Storage-Facility Registers 4-64
Control Register 2 4-64
Guarded-Storage-Designation (GSD) Register 4-65
Guarded-Storage-Section-Mask (GSSM) Register 4-65
Guarded-Storage-Event Parameter-List-Address (GSEPLA) Register 4-66
Guarded-Storage Control Block (GSCB) 4-66
Guarded-Storage-Event Parameter List (GSEPL) 4-66
Guarded-Storage Facility Operation 4-69
Guarded-Storage-Event Detection 4-69
Guarded-Storage-Event Processing 4-70
Externally Initiated Functions 4-73
Resets 4-73
. CPU Reset 4-77
Initial CPU Reset 4-78
Subsystem Reset 4-79
Clear Reset 4-79
Power-On Reset 4-80
Initial Program Loading 4-81
CCW-Type IPL 4-81
Store Status 4-81
Multiprocessing 4-82
Shared Main Storage 4-83
CPU-Address Identification 4-83
CPU-Address Expansion 4-83
CPU-Address Contraction 4-84
CPU Signaling and Response 4-84
Signal-Processor Orders 4-84
Sense 4-85
External Call 4-85
Emergency Signal 4-85
Start 4-85
Stop 4-85
Restart 4-85
Stop and Store Status 4-85
Initial CPU Reset 4-85
CPU Reset 4-85
Set Prefix 4-85
Store Status at Address 4-86
Store Extended Status at Address 4-87
Set Architecture 4-89
Conditional Emergency Signal 4-90
Sense Running Status 4-91
Set Multithreading 4-91
Store Additional Status at Address 4-92
Conditions Determining Response 4-94
Conditions Precluding Interpretation of the Order Code 4-94
Status Bits 4-94
Facility Indications 4-98

Chapter 5, Program Execution 5-1
Instructions 5-3
Operands 5-3
Instruction Formats 5-3
Register Operands 5-7
Immediate Operands 5-8
Storage Operands 5-9
Address Generation 5-10
Trimodal Addressing 5-10
Sequential Instruction-Address Generation . . . 5-10
Operand-Address Generation 5-11
Formation of the Intermediate Value 5-11
Formation of the Operand Address 5-12
Branch-Address Generation 5-12
Formation of the Intermediate Value 5-12
Formation of the Branch Address 5-13
Instruction Execution and Sequencing 5-14
Decision Making 5-14
Loop Control 5-14
Subroutine Linkage without the Linkage Stack 5-14
Simple Branch Instructions 5-14
Other Linkage Instructions 5-17
Interruptions 5-22
Types of Instruction Ending 5-22
Completion 5-22
Suppression 5-22
Nullification 5-23
Termination 5-23
Interruptible Instructions 5-24
Point of Interruption 5-24
Unit of Operation 5-24
Execution of Interruptible Instructions 5-24
Condition-Code Alternative to Interruptibility 5-25
Exceptions to Nullification and Suppression . . 5-26
Modification of DAT-Table Entries 5-27
Trial Execution for Editing Instructions and Translate Instruction 5-27
Authorization Mechanisms 5-27
Mode Requirements 5-28
Extraction-Authority Control 5-28
PSW-Key Mask 5-28
Secondary-Space Control 5-29
Subsystem-Linkage Control 5-29
ASN-Translation Control 5-29
Authorization Index 5-29
Instructions and Controls Related to ASN-and-LX Reuse 5-30

PC-Number Translation 5-33
PC-Number Translation Control 5-34
Control Register 0 5-34
Control Register 5 5-35
PC-Number Translation Tables 5-36
Linkage-Table Entries 5-36
Linkage-First-Table Entries 5-36
Linkage-Second-Table Entries 5-36
Entry-Table Entries 5-37
Table Summary 5-38
PC-Number-Translation Process 5-38
Obtaining the Linkage-Table or Linkage-First-Table Designation 5-42
Linkage-Table Lookup 5-42
Linkage-First-Table Lookup 5-42
Linkage-Second-Table Lookup 5-43
Linkage-Second-Table-Entry-Sequence-Number Comparison 5-43
Entry-Table Lookup 5-43
Recognition of Exceptions during PC-Number Translation 5-44
Home Address Space 5-44
Access-Register Introduction 5-45
Summary 5-45
Access-Register Functions 5-46
Access-Register-Specified Address Spaces 5-46
Access-Register Instructions 5-52
Access-Register Translation 5-53
Access-Register-Translation Control 5-53
Control Register 2 5-53
Control Register 5 5-54
Control Register 8 5-54
Access Registers 5-54
Access-Register-Translation Tables 5-55
Dispatchable-Unit Control Table and Access-List Designations 5-55
Access-List Entries 5-56
ASN-Second-Table Entries 5-57
Access-Register-Translation Process 5-59
Selecting the Access-List-Entry Token . . . 5-62
Obtaining the Primary or Secondary Address-Space-Control Element 5-62
Checking the First Byte of the ALET 5-62
Obtaining the Effective Access-List Designation 5-62
Access-List Lookup 5-62
Locating the ASN-Second-Table Entry . . . 5-63
Authorizing the Use of the Access-List Entry 5-63
Checking for Access-List-Controlled Protection 5-64
Obtaining the Address-Space-Control Element from the ASN-Second-Table Entry 5-64
Recognition of Exceptions during Access-Register Translation 5-64
ART-Lookaside Buffer 5-64
ALB Structure 5-64
Formation of ALB Entries 5-64
Use of ALB Entries 5-65
Modification of ART Tables 5-66
Subspace Groups 5-66
Subspace-Group Tables 5-66
Subspace-Group Dispatchable-Unit Control Table 5-66
Subspace-Group ASN-Second-Table Entries 5-68
Subspace-Replacement Operations 5-70
Linkage-Stack Introduction 5-70
Summary 5-71
Linkage-Stack Functions 5-71
Transferring Program Control 5-71
Branching Using the Linkage Stack 5-73
Adding and Retrieving Information 5-74
Testing Authorization 5-74
Program-Problem Analysis 5-75
Linkage-Stack Entry-Table Entries 5-75
Linkage-Stack Operations 5-76
Linkage-Stack-Operations Control 5-78
Control Register 0 5-78
Control Register 15 5-78
Linkage Stack 5-79
Entry Descriptors 5-79
Header Entries 5-80
Trailer Entries 5-81
State Entries 5-81
Stacking Process 5-84
Locating Space for a New Entry 5-84
Forming the New Entry 5-85
Updating the Current Entry 5-86
Updating Control Register 15 5-86
Recognition of Exceptions during the Stacking Process 5-86
Unstacking Process 5-86
Locating the Current Entry and Processing a Header Entry 5-87
Checking for a State Entry 5-88
Restoring Information 5-88
Updating the Preceding Entry 5-89
Updating Control Register 15 5-89
Recognition of Exceptions during the Unstacking Process 5-89
Transactional-Execution Facility 5-89
Transactional-Execution Terminology 5-89
Transactional-Execution Facility Controls 5-92
Control Register Bits 5-92
Transaction-Diagnostic-Block Address (TDBA) 5-93

Transaction-Abort PSW (TAPSW) 5-93
Transaction Nesting Depth (TND) 5-93
Transaction Diagnostic Block (TDB) 5-93
Transactional-Execution Facility Instructions . 5-97
Restricted Instructions 5-97
Transactional-Execution Facility Operation . . 5-99
Transaction Initiation 5-99
Execution in the Transactional-Execution Mode 5-99
Normal Transaction Ending 5-99
Transaction Abort Conditions 5-99
Transaction Abort Processing 5-102
Program-Interruption Filtering on a Transaction Abort 5-103
Priority of Abort Conditions 5-107
Constrained Transaction 5-107
Monitor-Event Counting 5-109
ESA/390-Compatibility-Mode Facility 5-111
Sequence of Storage References 5-113
Conceptual Sequence 5-113
Overlapped Operation of Instruction Execution 5-114
Divisible Instruction Execution 5-115
Interlocks for Virtual-Storage References . . 5-115
Interlocks between Instructions 5-116
Interlocks within a Single Instruction 5-116
Instruction Fetching 5-118
ART-Table and DAT-Table Fetches 5-119
Storage-Key Accesses 5-120
Storage-Operand References 5-123
Storage-Operand Fetch References 5-123
Storage-Operand Store References 5-123
Storage-Operand Update References . . . 5-124
Storage-Operand Consistency 5-125
Single-Access References 5-125
Multiple-Access References 5-125
Block-Concurrent References 5-126
Relation between Operand Accesses 5-129
Storage Operand References in the Transactional-Execution Mode 5-129
Other Storage References 5-129
Relation between Storage-Key Accesses . . 5-130
Serialization 5-130
CPU Serialization 5-130
Specific-Operand Serialization 5-132
Channel-Program Serialization 5-133
Quiescing 5-133

Chapter 6, Interruptions 6-1
Interruption Action 6-2
Interruption Code 6-5
Enabling and Disabling 6-6
Handling of Floating Interruption Conditions . . 6-6
Instruction-Length Code 6-7
Zero ILC 6-7
ILC on Instruction-Fetching Exceptions 6-8
Exceptions Associated with the PSW 6-9
Early Exception Recognition 6-9
Late Exception Recognition 6-10
External Interruption 6-10
Clock Comparator 6-12
CPU Timer 6-12
Emergency Signal 6-12
External Call 6-13
Interrupt Key 6-13
Malfunction Alert 6-13
Measurement Alert 6-13
Service Signal 6-14
Timing Alert 6-14
ETR-Timing-Alert Condition 6-14
STP-Timing-Alert Condition 6-14
Warning Track 6-15
I/O Interruption 6-15
Machine-Check Interruption 6-16
Program Interruption 6-16
Data-Exception Code (DXC) 6-17
Priority of Program Interruptions for Data Exceptions 6-17
Vector-Exception Code 6-20
Program-Interruption Conditions 6-20
Addressing Exception 6-20
AFX-Translation Exception 6-22
ALEN-Translation Exception 6-22
ALE-Sequence Exception 6-22
ALET-Specification Exception 6-22
ASCE-Type Exception 6-22
ASTE-Instance Exception 6-23
ASTE-Sequence Exception 6-23
ASTE-Validity Exception 6-24
ASX-Translation Exception 6-24
Crypto-Operation Exception 6-24
Data Exception 6-25
Decimal-Divide Exception 6-26
Decimal-Overflow Exception 6-26
Execute Exception 6-26
EX-Translation Exception 6-26
Extended-Authority Exception 6-27
Fixed-Point-Divide Exception 6-27
Fixed-Point-Overflow Exception 6-27
HFP-Divide Exception 6-28
HFP-Exponent-Overflow Exception 6-28
HFP-Exponent-Underflow Exception 6-28
HFP-Significance Exception 6-28
HFP-Square-Root Exception 6-29
LFX-Translation Exception 6-29
LSTE-Sequence Exception 6-29
LSX-Translation Exception 6-29

LX-Translation Exception. 6-30
Monitor Event. 6-30
Operand Exception 6-31
Operation Exception 6-31
Page-Translation Exception. 6-32
PC-Translation-Specification Exception . . 6-32
PER Event 6-32
Primary-Authority Exception 6-33
Privileged-Operation Exception 6-33
Protection Exception 6-34
Region-First-Translation Exception 6-35
Region-Second-Translation Exception . . . 6-36
Region-Third-Translation Exception 6-36
Secondary-Authority Exception 6-37
Segment-Translation Exception. 6-37
Space-Switch Event. 6-37
Special-Operation Exception 6-38
Specification Exception 6-40
Stack-Empty Exception 6-44
Stack-Full Exception 6-44
Stack-Operation Exception 6-44
Stack-Specification Exception 6-45
Stack-Type Exception 6-45
Trace-Table Exception. 6-45
Transaction-Constraint Exception 6-45
Transactional-Execution-Aborted Event . . 6-46
Translation-Specification Exception. 6-46
Vector Processing Exception. 6-47
Collective Program-Interruption Names. 6-47
Recognition of Access Exceptions. 6-47
Multiple Program-Interruption Conditions . . . 6-50
Access Exceptions. 6-53
ASN-Translation Exceptions 6-56
Subspace-Replacement Exceptions 6-56
Trace Exceptions. 6-56
Restart Interruption 6-56
Supervisor-Call Interruption. 6-57
Priority of Interruptions 6-57

Chapter 7, General Instructions 7-1

Data Format. 7-3
Binary-Integer Representation. 7-4
Binary Arithmetic 7-5
Signed Binary Arithmetic 7-5
Addition and Subtraction 7-5
Fixed-Point Overflow 7-5
Unsigned Binary Arithmetic 7-6
Signed and Logical Comparison 7-7
Instructions 7-7
ADD 7-26
ADD IMMEDIATE 7-27
ADD HALFWORD 7-28
ADD HALFWORD IMMEDIATE. 7-28
ADD HIGH 7-29
ADD IMMEDIATE HIGH 7-29
ADD LOGICAL 7-30
ADD LOGICAL IMMEDIATE. 7-30
ADD LOGICAL HIGH 7-31
ADD LOGICAL WITH CARRY 7-31
ADD LOGICAL WITH SIGNED IMMEDIATE . 7-32
ADD LOGICAL WITH SIGNED IMMEDIATE HIGH. 7-33
AND 7-33
AND IMMEDIATE 7-34
BRANCH AND LINK 7-35
BRANCH AND SAVE 7-36
BRANCH AND SAVE AND SET MODE 7-37
BRANCH AND SET MODE 7-38
BRANCH INDIRECT ON CONDITION 7-39
BRANCH ON CONDITION 7-40
BRANCH ON COUNT 7-41
BRANCH ON INDEX HIGH 7-42
BRANCH ON INDEX LOW OR EQUAL 7-42
BRANCH PREDICTION PRELOAD 7-43
BRANCH PREDICTION RELATIVE PRELOAD. 7-43
BRANCH RELATIVE AND SAVE 7-46
BRANCH RELATIVE AND SAVE LONG 7-46
BRANCH RELATIVE ON CONDITION. 7-46
BRANCH RELATIVE ON CONDITION LONG 7-46
BRANCH RELATIVE ON COUNT 7-47
BRANCH RELATIVE ON COUNT HIGH 7-47
BRANCH RELATIVE ON INDEX HIGH 7-48
BRANCH RELATIVE ON INDEX LOW OR EQUAL 7-48
CHECKSUM 7-49
CIPHER MESSAGE 7-53
CIPHER MESSAGE WITH CHAINING. 7-53
CIPHER MESSAGE WITH AUTHENTICATION 7-78
CIPHER MESSAGE WITH CIPHER FEEDBACK. 7-92
CIPHER MESSAGE WITH COUNTER. 7-107
CIPHER MESSAGE WITH OUTPUT FEEDBACK. 7-120
COMPARE 7-134
COMPARE IMMEDIATE. 7-134
COMPARE RELATIVE LONG 7-135
COMPARE AND BRANCH. 7-135
COMPARE AND BRANCH RELATIVE. 7-135
COMPARE IMMEDIATE AND BRANCH . . . 7-136
COMPARE IMMEDIATE AND BRANCH RELATIVE. 7-136
COMPARE AND FORM CODEWORD. 7-137
COMPARE AND SWAP 7-144
COMPARE DOUBLE AND SWAP 7-144
COMPARE AND SWAP AND STORE 7-146

COMPARE AND TRAP 7-149
COMPARE IMMEDIATE AND TRAP. 7-149
COMPARE HALFWORD 7-150
COMPARE HALFWORD IMMEDIATE 7-150
COMPARE HALFWORD RELATIVE LONG 7-150
COMPARE HIGH. 7-151
COMPARE IMMEDIATE HIGH 7-151
COMPARE LOGICAL 7-152
COMPARE LOGICAL IMMEDIATE 7-152
COMPARE LOGICAL RELATIVE LONG. . . 7-153
COMPARE LOGICAL AND BRANCH 7-154
COMPARE LOGICAL AND BRANCH RELATIVE 7-154
COMPARE LOGICAL IMMEDIATE AND BRANCH 7-154
COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE 7-154
COMPARE LOGICAL AND TRAP 7-155
COMPARE LOGICAL IMMEDIATE AND TRAP 7-156
COMPARE LOGICAL CHARACTERS UNDER MASK. 7-157
COMPARE LOGICAL HIGH 7-157
COMPARE LOGICAL IMMEDIATE HIGH . . 7-158
COMPARE LOGICAL LONG 7-158
COMPARE LOGICAL LONG EXTENDED. . 7-160
COMPARE LOGICAL LONG UNICODE . . . 7-163
COMPARE LOGICAL STRING 7-166
COMPARE UNTIL SUBSTRING EQUAL . . 7-167
COMPRESSION CALL 7-170
COMPUTE INTERMEDIATE MESSAGE DIGEST 7-188
COMPUTE LAST MESSAGE DIGEST 7-201
COMPUTE MESSAGE AUTHENTICATION CODE. 7-219
CONVERT TO BINARY 7-230
CONVERT TO DECIMAL. 7-231
CONVERT UTF-16 TO UTF-32 7-231
CONVERT UTF-16 TO UTF-8 7-234
CONVERT UNICODE TO UTF-8. 7-234
CONVERT UTF-32 TO UTF-16 7-238
CONVERT UTF-32 TO UTF-8 7-241
CONVERT UTF-8 TO UTF-16 7-244
CONVERT UTF-8 TO UNICODE. 7-244
CONVERT UTF-8 TO UTF-32 7-248
COPY ACCESS 7-252
DIVIDE 7-252
DIVIDE LOGICAL. 7-253
DIVIDE SINGLE. 7-254
EXCLUSIVE OR. 7-254
EXCLUSIVE OR IMMEDIATE 7-256
EXECUTE 7-256
EXECUTE RELATIVE LONG. 7-256
EXTRACT ACCESS. 7-257
EXTRACT CPU ATTRIBUTE 7-257
EXTRACT CPU TIME 7-260
EXTRACT PSW 7-261
EXTRACT TRANSACTION NESTING DEPTH 7-261
FIND LEFTMOST ONE. 7-262
INSERT CHARACTER 7-262
INSERT CHARACTERS UNDER MASK. . . .7-262
INSERT IMMEDIATE 7-263
INSERT PROGRAM MASK. 7-264
LOAD. 7-264
LOAD IMMEDIATE 7-264
LOAD RELATIVE LONG. 7-264
LOAD ACCESS MULTIPLE 7-265
LOAD ADDRESS 7-266
LOAD ADDRESS EXTENDED 7-266
LOAD ADDRESS RELATIVE LONG. 7-267
LOAD AND ADD 7-268
LOAD AND ADD LOGICAL. 7-268
LOAD AND AND 7-269
LOAD AND EXCLUSIVE OR. 7-269
LOAD AND OR 7-270
LOAD AND TEST 7-270
LOAD AND TRAP 7-271
LOAD AND ZERO RIGHTMOST BYTE 7-271
LOAD BYTE 7-272
LOAD BYTE HIGH 7-272
LOAD COMPLEMENT 7-272
LOAD COUNT TO BLOCK BOUNDARY . . .7-273
LOAD GUARDED 7-274
LOAD LOGICAL AND SHIFT GUARDED . .7-274
LOAD GUARDED STORAGE CONTROLS .7-275
LOAD HALFWORD. 7-276
LOAD HALFWORD IMMEDIATE 7-276
LOAD HALFWORD RELATIVE LONG 7-276
LOAD HALFWORD HIGH. 7-277
LOAD HALFWORD IMMEDIATE ON CONDITION 7-277
LOAD HALFWORD HIGH IMMEDIATE ON CONDITION 7-277
LOAD HIGH. 7-278
LOAD HIGH AND TRAP 7-278
LOAD LOGICAL 7-278
LOAD LOGICAL RELATIVE LONG. 7-278
LOAD LOGICAL AND TRAP. 7-279
LOAD LOGICAL AND ZERO RIGHTMOST BYTE. 7-279
LOAD LOGICAL CHARACTER. 7-279
LOAD LOGICAL CHARACTER HIGH. 7-280
LOAD LOGICAL HALFWORD. 7-280
LOAD LOGICAL HALFWORD RELATIVE LONG 7-280
LOAD LOGICAL HALFWORD HIGH. 7-281
LOAD LOGICAL IMMEDIATE. 7-281

LOAD LOGICAL THIRTY ONE BITS. 7-281
LOAD LOGICAL THIRTY ONE BITS AND TRAP 7-282
LOAD MULTIPLE 7-282
LOAD MULTIPLE DISJOINT. 7-283
LOAD MULTIPLE HIGH 7-283
LOAD NEGATIVE 7-283
LOAD ON CONDITION 7-284
LOAD HIGH ON CONDITION 7-284
LOAD PAIR DISJOINT 7-285
LOAD PAIR FROM QUADWORD 7-286
LOAD POSITIVE 7-287
LOAD REVERSED 7-287
MONITOR CALL 7-288
MOVE 7-289
MOVE INVERSE 7-290
MOVE LONG. 7-290
MOVE LONG EXTENDED 7-294
MOVE LONG UNICODE 7-297
MOVE NUMERICS 7-301
MOVE STRING 7-301
MOVE WITH OFFSET. 7-302
MOVE ZONES. 7-303
MULTIPLY 7-304
MULTIPLY HALFWORD 7-305
MULTIPLY HALFWORD IMMEDIATE. 7-305
MULTIPLY LOGICAL. 7-306
MULTIPLY SINGLE. 7-307
MULTIPLY SINGLE IMMEDIATE 7-307
NEXT INSTRUCTION ACCESS INTENT . . 7-308
NONTRANSACTIONAL STORE 7-310
OR 7-310
OR IMMEDIATE 7-312
PACK. 7-312
PACK ASCII. 7-313
PACK UNICODE 7-314
PERFORM CRYPTOGRAPHIC COMPUTATION 7-315
PERFORM LOCKED OPERATION. 7-331
PERFORM PROCESSOR ASSIST 7-345
PERFORM RANDOM NUMBER OPERATION 7-346
POPULATION COUNT. 7-359
PREFETCH DATA. 7-359
PREFETCH DATA RELATIVE LONG 7-359
ROTATE LEFT SINGLE LOGICAL 7-361
ROTATE THEN AND SELECTED BITS . . . 7-361
ROTATE THEN EXCLUSIVE OR SELECTED BITS. 7-362
ROTATE THEN OR SELECTED BITS 7-362
ROTATE THEN INSERT SELECTED BITS 7-363
ROTATE THEN INSERT SELECTED BITS HIGH 7-364
ROTATE THEN INSERT SELECTED BITS LOW 7-365
SEARCH STRING 7-366
SEARCH STRING UNICODE. 7-368
SET ACCESS 7-370
SET ADDRESSING MODE 7-370
SET PROGRAM MASK 7-371
SHIFT LEFT DOUBLE 7-371
SHIFT LEFT DOUBLE LOGICAL 7-372
SHIFT LEFT SINGLE 7-372
SHIFT LEFT SINGLE LOGICAL. 7-373
SHIFT RIGHT DOUBLE 7-374
SHIFT RIGHT DOUBLE LOGICAL. 7-374
SHIFT RIGHT SINGLE 7-375
SHIFT RIGHT SINGLE LOGICAL. 7-376
STORE 7-376
STORE RELATIVE LONG 7-377
STORE ACCESS MULTIPLE 7-377
STORE CHARACTER 7-378
STORE CHARACTER HIGH 7-378
STORE CHARACTERS UNDER MASK. . . . 7-378
STORE CLOCK 7-379
STORE CLOCK FAST 7-379
STORE CLOCK EXTENDED 7-380
STORE FACILITY LIST EXTENDED 7-382
STORE GUARDED STORAGE CONTROLS 7-383
STORE HALFWORD 7-383
STORE HALFWORD RELATIVE LONG . . . 7-384
STORE HALFWORD HIGH 7-384
STORE HIGH 7-384
STORE MULTIPLE. 7-385
STORE MULTIPLE HIGH. 7-385
STORE ON CONDITION 7-385
STORE HIGH ON CONDITION 7-386
STORE PAIR TO QUADWORD 7-386
STORE REVERSED. 7-387
SUBTRACT. 7-387
SUBTRACT HALFWORD 7-388
SUBTRACT HIGH. 7-389
SUBTRACT LOGICAL 7-389
SUBTRACT LOGICAL IMMEDIATE. 7-390
SUBTRACT LOGICAL HIGH 7-391
SUBTRACT LOGICAL WITH BORROW . . . 7-391
SUPERVISOR CALL 7-392
TEST ADDRESSING MODE 7-392
TEST AND SET 7-392
TEST UNDER MASK (TEST UNDER MASK HIGH, TEST UNDER MASK LOW) 7-393
TRANSACTION ABORT. 7-394
TRANSACTION BEGIN (TBEGIN) 7-395
TRANSACTION BEGIN (TBEGINC). 7-399
TRANSACTION END 7-401
TRANSLATE. 7-402
TRANSLATE AND TEST 7-403

TRANSLATE AND TEST EXTENDED. 7-403
TRANSLATE AND TEST REVERSE EXTENDED 7-404
TRANSLATE AND TEST REVERSE. 7-408
TRANSLATE EXTENDED 7-409
TRANSLATE ONE TO ONE 7-411
TRANSLATE ONE TO TWO 7-411
TRANSLATE TWO TO ONE 7-411
TRANSLATE TWO TO TWO. 7-411
UNPACK 7-416
UNPACK ASCII 7-417
UNPACK UNICODE. 7-418
UPDATE TREE 7-419
Protection of Cryptographic Keys 7-424
Protection of DES Keys 7-425
Protection of AES Keys 7-427

Chapter 8, Decimal Instructions 8-1
Decimal-Number Formats 8-1
Zoned Format 8-1
Packed-Decimal Formats. 8-1
Signed-Packed-Decimal Format 8-2
Unsigned-Packed-Decimal Format 8-2
Decimal Codes. 8-2
Decimal Operations 8-3
Decimal-Arithmetic Instructions 8-3
Editing Instructions. 8-4
Execution of Decimal Instructions 8-4
Other Instructions for Decimal Operands. 8-4
General-Operand Data Exception 8-5
Instructions. 8-5
ADD DECIMAL. 8-6
COMPARE DECIMAL 8-7
DIVIDE DECIMAL 8-7
EDIT. 8-8
EDIT AND MARK. 8-11
MULTIPLY DECIMAL. 8-12
SHIFT AND ROUND DECIMAL. 8-12
SUBTRACT DECIMAL. 8-13
TEST DECIMAL. 8-14
ZERO AND ADD 8-14

Chapter 9, Floating-Point Overview and Support Instructions. 9-1
Sign Bit. 9-2
Finite Floating-Point Numbers 9-2
Infinities 9-2
Not-A-Number (NaN) 9-2
Signaling and Quiet NaNs 9-2
Payload 9-2
Propagation of NaNs 9-3
Default QNaN. 9-3
Floating-Point Number Representations 9-3
Hexadecimal-Floating-Point (HFP) 9-3
Binary Floating-Point (BFP). 9-4
Decimal Floating-Point (DFP) 9-4
Canonical DFP Data 9-4
Comparison of Floating-Point Number Representations 9-4
Floating-Point Number Ranges 9-4
Equivalent Floating-Point Number Representations 9-5
Effective Width. 9-7
Floating-Point Data in Storage 9-8
Registers And Controls 9-8
Floating-Point Registers 9-8
Additional Floating-Point (AFP) Registers. . .9-9
Valid Floating-Point-Register Designations. .9-9
Floating-Point-Control (FPC) Register. 9-9
IEEE Masks and Flags 9-10
FPC DXC Byte. 9-10
Operations on the FPC Register 9-10
AFP-Register-Control Bit. 9-11
IEEE Computational Operations 9-11
Intermediate Values 9-12
Precise Intermediate Value 9-12
Precision-Rounded Value 9-12
Denormalized Value 9-12
Functionally-Rounded Value 9-12
Rounded Intermediate Value. 9-12
Scaled Value 9-12
Scale Factor (Ψ) 9-12
Unsigned Scaling Exponent (α). 9-12
Signed Scaling Exponent (Ω) 9-13
IEEE Rounding 9-13
Permissible Set 9-13
Selection of Candidates. 9-13
Ties 9-13
Voting Digit and Common-Rounding-Point View. 9-13
Rounding Methods 9-14
Explicit Rounding Methods 9-17
Summary of Rounding Action 9-17
IEEE Exceptions 9-18
Concurrent IEEE Exceptions. 9-18
IEEE Invalid Operation 9-19
IEEE Division-By-Zero. 9-19
IEEE Overflow 9-19
IEEE Underflow 9-20
IEEE Inexact 9-20
Quantum Exception. 9-21
Suppression of Certain IEEE Exceptions 9-23
IEEE Same-Radix Format Conversion 9-23
IEEE Comparison 9-24
Condition Codes for IEEE Instructions 9-24
Instructions 9-24
CONVERT BFP TO HFP. 9-27

CONVERT HFP TO BFP 9-28
COPY SIGN. 9-30
EXTRACT FPC 9-30
LOAD. 9-31
LOAD COMPLEMENT. 9-31
LOAD FPC. 9-31
LOAD FPC AND SIGNAL 9-32
LOAD FPR FROM GR. 9-34
LOAD GR FROM FPR. 9-34
LOAD NEGATIVE 9-34
LOAD POSITIVE 9-34
LOAD ZERO 9-35
PERFORM FLOATING-POINT OPERATION 9-35
SET BFP ROUNDING MODE 9-47
SET DFP ROUNDING MODE 9-47
SET FPC 9-47
SET FPC AND SIGNAL. 9-48
STORE 9-48
STORE FPC 9-49
Summary of All Floating-Point Instructions 9-49
Impacts on ESA/390 and ESA/390-Compatibility Mode 9-52
Impacts of the Decimal-Floating-Point Facility 9-52
Impacts of the Floating-Point Extension Facility 9-53

Chapter 10, Control Instructions . . . 10-1
BRANCH AND SET AUTHORITY 10-7
BRANCH AND STACK 10-11
BRANCH IN SUBSPACE GROUP 10-13
COMPARE AND REPLACE DAT TABLE ENTRY 10-17
COMPARE AND SWAP AND PURGE 10-21
DIAGNOSE 10-23
EXTRACT AND SET EXTENDED AUTHORITY 10-24
EXTRACT PRIMARY ASN 10-24
EXTRACT PRIMARY ASN AND INSTANCE 10-24
EXTRACT SECONDARY ASN 10-24
EXTRACT SECONDARY ASN AND INSTANCE 10-25
EXTRACT STACKED REGISTERS 10-25
EXTRACT STACKED STATE 10-26
INSERT ADDRESS SPACE CONTROL . . . 10-29
INSERT PSW KEY 10-30
INSERT REFERENCE BITS MULTIPLE. . . 10-30
INSERT STORAGE KEY EXTENDED 10-30
INSERT VIRTUAL STORAGE KEY. 10-31
INVALIDATE DAT TABLE ENTRY 10-32
INVALIDATE PAGE TABLE ENTRY 10-37
LOAD ADDRESS SPACE PARAMETERS . 10-41
LOAD CONTROL 10-50
LOAD PAGE TABLE ENTRY ADDRESS . . 10-50
LOAD PSW 10-54
LOAD PSW EXTENDED 10-55
LOAD REAL ADDRESS 10-56
LOAD USING REAL ADDRESS 10-60
MODIFY STACKED STATE 10-61
MOVE PAGE 10-62
MOVE TO PRIMARY 10-64
MOVE TO SECONDARY 10-64
MOVE WITH DESTINATION KEY 10-66
MOVE WITH KEY 10-67
MOVE WITH OPTIONAL SPECIFICATIONS 10-68
MOVE WITH SOURCE KEY. 10-71
PAGE IN 10-72
PAGE OUT 10-73
PERFORM CRYPTOGRAPHIC KEY MANAGEMENT OPERATION 10-74
PERFORM FRAME MANAGEMENT FUNCTION 10-77
PERFORM TIMING FACILITY FUNCTION . 10-81
PERFORM TOPOLOGY FUNCTION 10-89
PROGRAM CALL 10-90
PROGRAM RETURN 10-103
PROGRAM TRANSFER 10-107
PROGRAM TRANSFER WITH INSTANCE 10-107
PURGE ALB 10-116
PURGE TLB 10-116
RESET REFERENCE BIT EXTENDED . . . 10-116
RESET REFERENCE BITS MULTIPLE . . . 10-117
RESUME PROGRAM 10-117
SET ADDRESS SPACE CONTROL. 10-120
SET ADDRESS SPACE CONTROL FAST. 10-120
SET CLOCK 10-121
SET CLOCK COMPARATOR. 10-122
SET CLOCK PROGRAMMABLE FIELD. . . 10-123
SET CPU TIMER 10-123
SET PREFIX. 10-123
SET PSW KEY FROM ADDRESS 10-124
SET SECONDARY ASN. 10-125
SET SECONDARY ASN WITH INSTANCE 10-125
SET STORAGE KEY EXTENDED 10-130
SET SYSTEM MASK 10-133
SIGNAL PROCESSOR. 10-133
STORE CLOCK COMPARATOR 10-135
STORE CONTROL. 10-135
STORE CPU ADDRESS. 10-136
STORE CPU ID 10-136
STORE CPU TIMER. 10-138
STORE FACILITY LIST 10-138
STORE PREFIX 10-139
STORE REAL ADDRESS. 10-139
STORE SYSTEM INFORMATION 10-140
STORE THEN AND SYSTEM MASK 10-164
STORE THEN OR SYSTEM MASK 10-164
STORE USING REAL ADDRESS 10-165

TEST ACCESS 10-165
TEST BLOCK. 10-167
TEST PENDING EXTERNAL INTERRUPTION 10-169
TEST PROTECTION 10-170
TRACE. 10-173
TRAP 10-174

Chapter 11, Machine-Check Handling. 11-1
Machine-Check Detection 11-2
Correction of Machine Malfunctions. 11-2
Error Checking and Correction. 11-2
CPU Retry 11-2
Effects of CPU Retry 11-3
Checkpoint Synchronization. 11-3
Handling of Machine Checks during Checkpoint Synchronization. 11-4
Checkpoint-Synchronization Operations . . 11-4
Checkpoint-Synchronization Action 11-4
Channel-Subsystem Recovery. 11-5
Unit Deletion. 11-5
Handling of Machine Checks 11-5
Validation 11-6
Invalid CBC in Storage. 11-7
Programmed Validation of Storage 11-7
Invalid CBC in Storage Keys 11-7
Invalid CBC in Registers 11-9
Check-Stop State. 11-9
System Check Stop 11-10
Machine-Check Interruption. 11-10
Exigent Conditions 11-10
Repressible Conditions 11-10
Interruption Action 11-11
Interruption Action in the z/Architecture Architectural Mode 11-11
Interruption Action in the ESA/390-Compatibility Mode. 11-12
Interruption Action: Common Actions. . . . 11-13
Point of Interruption 11-14
Machine-Check-Interruption Code 11-14
Subclass. 11-15
System Damage. 11-15
Instruction-Processing Damage 11-15
System Recovery 11-15
Timing-Facility Damage 11-16
External Damage 11-16
Degradation 11-16
Warning 11-17
Channel Report Pending 11-17
Service-Processor Damage 11-17
Channel-Subsystem Damage 11-17
Subclass Modifiers 11-17
Backed Up 11-17
Delayed Access Exception 11-17
Ancillary Report 11-18
Synchronous Machine-Check-Interruption Conditions 11-18
Processing Backup 11-18
Processing Damage 11-19
Storage Errors 11-19
Storage Error Uncorrected 11-19
Storage Error Corrected 11-19
Storage-Key Error Uncorrected 11-19
Storage Degradation 11-19
Indirect Storage Error 11-20
Machine-Check Interruption-Code Validity Bits 11-20
PSW-MWP Validity 11-20
PSW Mask and Key Validity 11-20
PSW Program-Mask and Condition-Code Validity 11-21
PSW-Instruction-Address Validity 11-21
Failing-Storage-Address Validity 11-21
Vector Register Validity 11-21
External-Damage-Code Validity 11-21
Floating-Point-Register Validity 11-21
General-Register Validity. 11-21
Control-Register Validity 11-21
Storage Logical Validity 11-21
Access-Register Validity 11-21
Guarded-Storage-Registers Validity 11-22
TOD-Programmable-Register Validity 11-22
Floating-Point-Control-Register Validity . .11-22
CPU-Timer Validity 11-22
Clock-Comparator Validity. 11-22
Machine-Check Extended Interruption Information 11-22
Register-Save Areas 11-23
External-Damage Code. 11-23
Failing-Storage Address 11-24
Machine-Check Extended Save Area (MCESA) 11-24
Machine-Check Extended Save Area in the z/Architecture Architectural Mode . . .11-24
Machine-Check Extended Save Area in the ESA/390 Compatibility Mode. 11-25
Handling of Machine-Check Conditions 11-25
Floating Interruption Conditions 11-25
Floating Machine-Check-Interruption Conditions 11-26
Floating I/O Interruptions. 11-26
Machine-Check Masking. 11-26
Channel-Report-Pending Subclass Mask .11-26
Recovery Subclass Mask 11-26
Degradation Subclass Mask 11-27
External-Damage Subclass Mask 11-27

Warning Subclass Mask 11-27
Machine-Check Logout 11-27
Summary of Machine-Check Masking 11-27

Chapter 12, Operator Facilities 12-1
Manual Operation 12-1
Basic Operator Facilities 12-1
Address-Compare Controls 12-1
Alter-and-Display Controls 12-2
Architectural-Mode Indicator 12-2
Architectural-Mode-Selection Controls 12-3
Check-Stop Indicator 12-3
CPUs-per-Core Indicator 12-3
IML Controls 12-3
Interrupt Key 12-3
Load Indicator 12-3
Load-Clear Key 12-4
Load-Clear-List-Directed Key 12-4
Load-Normal Key 12-4
Load-with-Dump Key 12-4
Load-Unit-Address Controls 12-4
Manual Indicator 12-4
Power Controls 12-4
Rate Control 12-5
Restart Key 12-5
Start Key 12-5
Stop Key 12-5
Store-Status Key 12-5
System-Reset-Clear Key 12-6
System-Reset-Normal Key 12-6
Test Indicator 12-6
TOD-Clock Control 12-6
Wait Indicator 12-6
Multiprocessing Configurations 12-6
Multithreading Considerations 12-7

Chapter 13, I/O Overview 13-1
Input/Output (I/O) 13-1
The Channel Subsystem 13-1
Subchannel Sets 13-2
Subchannels 13-2
Attachment of Input/Output Devices 13-3
Channel Paths 13-3
Control Units 13-4
I/O Devices 13-5
I/O Addressing 13-5
Subchannel-Set Identifier 13-5
Channel-Path Identifier 13-5
Subchannel Number 13-5
Device Number 13-6
Device Identifier 13-6
Fibre-Channel Extensions 13-6
I/O-Command Words 13-7
Transport Command Word (TCW) 13-7
Channel Program Organization 13-7
CCW Channel Program 13-7
TCW Channel Program 13-7
Execution of I/O Operations 13-8
Start-Function Initiation 13-8
Subchannel Operation Modes 13-9
Path Management 13-9
Channel-Program Execution 13-9
Conclusion of I/O Operations 13-10
Chaining When Using a CCW Channel Program 13-11
Chaining When Using a TCW Channel Program 13-11
Premature Conclusion of I/O Operations . 13-12
I/O Interruptions 13-12

Chapter 14, I/O Instructions 14-1
I/O-Instruction Formats 14-1
I/O-Instruction Execution 14-2
Serialization 14-2
Operand Access 14-2
Condition Code 14-2
Program Exceptions 14-2
Instructions 14-2
CANCEL SUBCHANNEL 14-3
CLEAR SUBCHANNEL 14-5
HALT SUBCHANNEL 14-6
MODIFY SUBCHANNEL 14-7
RESET CHANNEL PATH 14-9
RESUME SUBCHANNEL 14-10
SET ADDRESS LIMIT 14-12
SET CHANNEL MONITOR 14-13
START SUBCHANNEL 14-15
STORE CHANNEL PATH STATUS 14-16
STORE CHANNEL REPORT WORD 14-17
STORE SUBCHANNEL 14-18
TEST PENDING INTERRUPTION 14-19
I/O-Interruption Code 14-20
TEST SUBCHANNEL 14-21

Chapter 15, Basic I/O Functions 15-1
Control of Basic I/O Functions 15-1
Subchannel-Information Block 15-2
Path-Management-Control Word 15-2
Subchannel-Status Word 15-9
Model-Dependent Area/Measurement Block Address 15-9
Summary of Modifiable Fields 15-10
Channel-Path Allegiance 15-12
Working Allegiance 15-12
Working Allegiance for Subchannels Operating in Command Mode 15-12

Working Allegiance for Subchannels Operating in Transport Mode 15-12
Active Allegiance 15-13
Dedicated Allegiance 15-13
Channel-Path Availability 15-13
Control-Unit Type 15-14
Clear Function 15-14
Clear-Function Path Management 15-14
Clear-Function Subchannel Modification . . . 15-15
Clear-Function Signaling and Completion . . 15-15
Halt Function 15-16
Halt-Function Path Management 15-16
Halt-Function Signaling and Completion . . . 15-17
Start Function and Resume Function 15-20
Start-Function and Resume-Function Path Management 15-20
Interrogate Function 15-22
Interrogate-Function Path Management . . . 15-22
Interrogate TCCB Transportation and Completion 15-22
Execution of I/O Operations 15-23
Blocking of Data 15-24
Operation-Request Block 15-24
Command-Mode ORB 15-25
Transport-Mode ORB 15-30
Channel-Command Word 15-31
Transport Control Word 15-33
Transport-Command-Control Block 15-36
Transport-Command-Area Header 15-37
Transport-Command Area 15-37
Device-Command Word 15-38
Transport-Command-Area Trailer 15-40
Transport-Command DCW 15-42
Transfer-CBC-Offset-Block DCW 15-43
CBC-Offset Block 15-44
Transfer-TCA-Extension DCW 15-44
Transport-Command-Area Extension 15-45
Interrogate TCCB 15-46
Interrogate DCW 15-46
Interrogate Data 15-46
Transport Status Block 15-47
Transport-Status Header (TSH) 15-48
I/O-Status TSA 15-49
Device-Detected-Program-Check TSA . . 15-50
Interrogate TSA 15-56
Command Code 15-57
Designation of Storage Area 15-58
CCW Channel Program Chaining 15-60
Data Chaining 15-61
Command Chaining 15-63
TCW Channel Program Chaining 15-63
Skipping 15-64
Program-Controlled Interruption 15-64
Indirect-Storage Designator (ISD) 15-66
CCW Indirect Data Addressing 15-66
Modified CCW Indirect Data Addressing . . . 15-68
Transport Indirect Data Addressing 15-70
Suspension of CCW Channel-Program Execution 15-73
Commands and Flags for CCWs 15-75
Branching in CCW Channel Programs 15-76
Transfer in Channel 15-77
Command Retry 15-77
Concluding I/O Operations before Initiation . . . 15-77
Concluding I/O Operations during Initiation 15-78
Immediate Conclusion of Command-Mode I/O Operations 15-78
Concluding I/O Operations During Data Transfer 15-79
Channel-Path-Reset Function 15-80
Channel-Path-Reset-Function Signaling 15-81
Channel-Path-Reset-Function-Completion Signaling 15-81

Chapter 16, I/O Interruptions 16-1
Interruption Conditions 16-2
Intermediate Interruption Condition 16-4
Primary Interruption Condition 16-4
Secondary Interruption Condition 16-5
Alert Interruption Condition 16-5
Priority of Interruptions 16-5
Interruption Action 16-6
Interruption-Response Block 16-6
IRB Format 16-6
Subchannel-Status Word 16-7
Command-Mode SCSW 16-8
Subchannel Key 16-9
Suspend Control (S) 16-9
Extended-Status-Word Format (L) 16-9
Deferred Condition Code (CC) 16-9
CCW Format (F) 16-11
Prefetch (P) 16-11
Initial-Status-Interruption Control (I) 16-12
Address-Limit-Checking Control (A) 16-12
IRB-Format Control (X) 16-12
Suppress-Suspended Interruption (U) 16-12
Subchannel-Control Field 16-12
Zero Condition Code (Z) 16-12
Extended Control (E) 16-12
Path Not Operational (N) 16-13
Function Control (FC) 16-13
Activity Control (AC) 16-14
Status Control (SC) 16-17
CCW-Address Field 16-19
Device-Status Field 16-24
Subchannel-Status Field 16-24
Program-Controlled Interruption 16-24

Incorrect Length 16-24
Program Check 16-25
Protection Check 16-28
Channel-Data Check 16-28
Channel-Control Check 16-29
Interface-Control Check 16-29
Chaining Check 16-30
Count Field 16-31
Transport-Mode SCSW 16-34
Subchannel Key 16-35
Reserved 16-35
Extended-Status-Word Format (L) 16-35
Deferred Condition Code (CC) 16-35
Format (FMT) 16-35
IRB-Format Control (X) 16-35
Interrogate Complete (Q) 16-35
Subchannel-Control Field 16-35
Extended Control (E) 16-35
Path Not Operational (N) 16-35
Function Control (FC) 16-35
Activity Control (AC) 16-36
Status Control (SC) 16-36
TCW Address Field 16-36
Device-Status Field 16-39
Subchannel-Status Field 16-39
Incorrect Length 16-39
Program Check 16-40
Protection Check 16-42
Channel-Data Check 16-42
Channel-Control Check 16-43
Interface-Control Check 16-43
Channel-Subsystem Retry Failed 16-44
FCX-Status Field 16-44
Subchannel-Extended-Status Field 16-44
Extended-Status Word 16-47
Extended-Status Format 0 16-47
Subchannel Logout 16-47
Extended-Report Word 16-51
Failing-Storage Address 16-53
Extended-Subchannel-Logout Descriptor (ESLD) 16-53
Secondary-CCW Address 16-53
Extended-Status Format 1 16-53
Extended-Status Format 2 16-54
Extended-Status Format 3 16-55
Extended-Control Word 16-56
Extended-Measurement Word 16-56

Chapter 17, I/O Support Functions. . 17-1

Channel-Subsystem Monitoring 17-1
Channel-Subsystem Timing 17-2
Channel-Subsystem Timer 17-2
Measurement-Block Update 17-2
Measurement Block 17-3
Measurement-Block Format 17-7
Measurement-Block Origin 17-7
Measurement-Block Address 17-7
Measurement-Block Key 17-8
Measurement-Block Index 17-8
Measurement-Block-Update Mode 17-8
Measurement-Block-Format Control 17-8
Measurement-Block-Update Enable 17-9
Control-Unit-Queuing Measurement 17-9
Control-Unit-Defer Time 17-9
Device-Active-Only Measurement 17-9
Initial-Command-Response Measurement . 17-9
Time-Interval-Measurement Accuracy 17-9
Device-Connect-Time Measurement 17-10
Device-Connect-Time-Measurement Mode 17-10
Device-Connect-Time-Measurement Enable 17-10
Extended Measurement Word 17-11
Extended-Measurement-Word Enable . . . 17-11
Signals and Resets 17-11
Signals 17-11
Halt Signal 17-12
Clear Signal 17-12
Reset Signal 17-12
Resets 17-13
Channel-Path Reset 17-13
I/O-System Reset 17-13
Externally Initiated Functions 17-16
Initial Program Loading 17-16
CCW-type IPL 17-17
List-Directed IPL 17-19
Reconfiguration of the I/O System 17-23
Status Verification 17-23
Address-Limit Checking 17-23
Configuration Alert 17-24
Incorrect-Length-Indication Suppression 17-24
Concurrent Sense 17-24
Channel-Subsystem Recovery 17-24
Channel Report 17-25
Channel-Report Word 17-26
Restore-Subchannel Facility 17-28
Extended-Subchannel-Logout Facility 17-29
Channel-Subsystem-I/O-Priority Facility 17-29
Number of Channel-Subsystem-Priority Levels 17-30
Multiple-Subchannel-Set Facility 17-30

Chapter 18, Hexadecimal-Floating-Point Instructions. 18-1

HFP Arithmetic 18-1
HFP Number Representation 18-1

Normalization 18-2
HFP Data Formats 18-2
HFP Short Format 18-4
HFP Long Format 18-4
HFP Extended Format 18-4
Instructions 18-5
ADD NORMALIZED 18-8
ADD UNNORMALIZED 18-9
COMPARE 18-10
CONVERT FROM FIXED 18-11
CONVERT TO FIXED 18-11
DIVIDE 18-12
HALVE 18-13
LOAD AND TEST 18-13
LOAD COMPLEMENT 18-14
LOAD FP INTEGER 18-15
LOAD LENGTHENED 18-15
LOAD NEGATIVE 18-16
LOAD POSITIVE 18-16
LOAD ROUNDED 18-17
MULTIPLY 18-17
MULTIPLY AND ADD 18-19
MULTIPLY AND SUBTRACT 18-19
MULTIPLY AND ADD UNNORMALIZED . . . 18-20
MULTIPLY UNNORMALIZED 18-22
SQUARE ROOT 18-23
SUBTRACT NORMALIZED 18-24
SUBTRACT UNNORMALIZED 18-25

Chapter 19, Binary-Floating-Point Instructions 19-1

Binary-Floating-Point Facility 19-1
Floating-Point-Control (FPC) Register 19-2
BFP Arithmetic 19-2
BFP Data Formats 19-2
BFP Short Format 19-2
BFP Long Format 19-2
BFP Extended Format 19-2
Biased Exponent 19-2
Significand 19-3
Values of Nonzero Numbers 19-3
Classes of BFP Data 19-4
Zeros 19-4
Subnormal Numbers 19-4
Normal Numbers 19-4
Infinities 19-4
Signaling and Quiet NaNs 19-4
BFP-Format Conversion 19-5
BFP Rounding 19-5
BFP Comparison 19-5
Remainder 19-5
IEEE Exceptions 19-7
Summary of Rounding And Range Actions . . 19-8
Result Figures 19-9
Data-Exception Codes (DXC) and Abbreviations 19-10
Instructions 19-11
ADD 19-15
COMPARE 19-17
COMPARE AND SIGNAL 19-18
CONVERT FROM FIXED 19-19
CONVERT FROM LOGICAL 19-21
CONVERT TO FIXED 19-22
CONVERT TO LOGICAL 19-25
DIVIDE 19-27
DIVIDE TO INTEGER 19-28
LOAD AND TEST 19-31
LOAD COMPLEMENT 19-31
LOAD FP INTEGER 19-32
LOAD LENGTHENED 19-33
LOAD NEGATIVE 19-34
LOAD POSITIVE 19-35
LOAD ROUNDED 19-35
MULTIPLY 19-37
MULTIPLY AND ADD 19-38
MULTIPLY AND SUBTRACT 19-38
SQUARE ROOT 19-40
SUBTRACT 19-40
TEST DATA CLASS 19-41

Chapter 20, Decimal-Floating-Point Instructions 20-1

Decimal-Floating-Point Facility 20-1
DFP Arithmetic 20-2
Finite Floating-Point Number 20-2
Cohort 20-2
Quantum 20-2
Preferred Quantum 20-2
Scaled Preferred Quantum 20-3
Delivered Quantum 20-3
Special Quantum-Handling Operations 20-3
DFP Data Formats 20-3
DFP Short Format 20-3
DFP Long Format 20-3
DFP Extended Format 20-4
Sign 20-4
Combination 20-4
Encoded Trailing Significand 20-5
Values of Finite Numbers 20-5
Significand 20-5
DFP Significant Digits 20-5
Canonical Declets 20-6
DFP Canonical Data 20-6
Classes of DFP Data 20-6
Zeros 20-6
Subnormal Numbers 20-6

Normal Numbers 20-7
Infinities 20-7
Signaling and Quiet NaNs 20-7
Canonicalization 20-7
DFP-Format Conversion 20-7
DFP Rounding 20-7
DFP Comparison 20-7
DFP Formatting Instructions 20-8
Signed-Packed-Decimal Format 20-8
Unsigned-Packed-Decimal Format 20-8
Zoned-Decimal Format 20-9
IEEE Exceptions 20-9
Summary of Preferred Quantum 20-10
Summary of Rounding And Range Actions . 20-10
Result Figures 20-15
Data-Exception Codes (DXC) and Abbreviations 20-15
Instructions 20-16
ADD 20-19
COMPARE 20-22
COMPARE AND SIGNAL 20-23
COMPARE BIASED EXPONENT 20-23
CONVERT FROM FIXED 20-24
CONVERT FROM LOGICAL 20-25
CONVERT FROM PACKED 20-26
CONVERT FROM SIGNED PACKED 20-28
CONVERT FROM UNSIGNED PACKED . . 20-28
CONVERT FROM ZONED 20-29
CONVERT TO FIXED 20-29
CONVERT TO LOGICAL 20-32
CONVERT TO PACKED 20-33
CONVERT TO SIGNED PACKED 20-35
CONVERT TO UNSIGNED PACKED 20-35
CONVERT TO ZONED 20-36
DIVIDE 20-37
EXTRACT BIASED EXPONENT 20-39
EXTRACT SIGNIFICANCE 20-39
INSERT BIASED EXPONENT 20-40
LOAD AND TEST 20-41
LOAD FP INTEGER 20-42
LOAD LENGTHENED 20-45
LOAD ROUNDED 20-46
MULTIPLY 20-47
QUANTIZE 20-49
REROUND 20-52
SHIFT SIGNIFICAND LEFT 20-54
SHIFT SIGNIFICAND RIGHT 20-54
SUBTRACT 20-55
TEST DATA CLASS 20-56
TEST DATA GROUP 20-57
Densely Packed Decimal (DPD) 20-58
Decimal-to-DPD Mapping 20-58
DPD-to-Decimal Mapping 20-58

Chapter 21, Vector Overview and Support Instructions 21-1
Overview 21-1
Vector Registers and Controls 21-1
Vector Enablement Control 21-1
Vector Storage Accesses 21-2
Saturating Arithmetic 21-2
Instructions 21-2
VECTOR BIT PERMUTE 21-4
VECTOR GATHER ELEMENT 21-4
VECTOR GENERATE BYTE MASK 21-5
VECTOR GENERATE MASK 21-5
VECTOR LOAD 21-6
VECTOR LOAD AND REPLICATE 21-6
VECTOR LOAD ELEMENT 21-6
VECTOR LOAD ELEMENT IMMEDIATE 21-7
VECTOR LOAD GR FROM VR ELEMENT . . 21-7
VECTOR LOAD LOGICAL ELEMENT AND ZERO 21-8
VECTOR LOAD MULTIPLE 21-9
VECTOR LOAD RIGHTMOST WITH LENGTH 21-9
VECTOR LOAD TO BLOCK BOUNDARY . . 21-10
VECTOR LOAD VR ELEMENT FROM GR . 21-11
VECTOR LOAD VR FROM GRS DISJOINT 21-11
VECTOR LOAD WITH LENGTH 21-11
VECTOR MERGE HIGH 21-12
VECTOR MERGE LOW 21-12
VECTOR PACK 21-13
VECTOR PACK SATURATE 21-13
VECTOR PACK LOGICAL SATURATE 21-14
VECTOR PERMUTE 21-15
VECTOR PERMUTE DOUBLEWORD IMMEDIATE 21-15
VECTOR REPLICATE 21-15
VECTOR REPLICATE IMMEDIATE 21-16
VECTOR SCATTER ELEMENT 21-16
VECTOR SELECT 21-17
VECTOR SIGN EXTEND TO DOUBLEWORD 21-17
VECTOR STORE 21-18
VECTOR STORE ELEMENT 21-18
VECTOR STORE MULTIPLE 21-19
VECTOR STORE RIGHTMOST WITH LENGTH 21-19
VECTOR STORE WITH LENGTH 21-20
VECTOR UNPACK HIGH 21-20
VECTOR UNPACK LOGICAL HIGH 21-21
VECTOR UNPACK LOW 21-21
VECTOR UNPACK LOGICAL LOW 21-22

Chapter 22, Vector Integer Instructions 22-1
Instructions 22-1
VECTOR ADD 22-3
VECTOR ADD COMPUTE CARRY 22-4
VECTOR ADD WITH CARRY 22-4
VECTOR ADD WITH CARRY COMPUTE CARRY 22-5
VECTOR AND 22-5
VECTOR AND WITH COMPLEMENT 22-5
VECTOR AVERAGE 22-6
VECTOR AVERAGE LOGICAL 22-6
VECTOR CHECKSUM 22-6
VECTOR ELEMENT COMPARE 22-7
VECTOR ELEMENT COMPARE LOGICAL . 22-7
VECTOR COMPARE EQUAL 22-7
VECTOR COMPARE HIGH 22-8
VECTOR COMPARE HIGH LOGICAL 22-9
VECTOR COUNT LEADING ZEROS 22-10
VECTOR COUNT TRAILING ZEROS 22-10
VECTOR EXCLUSIVE OR 22-11
VECTOR GALOIS FIELD MULTIPLY SUM . 22-11
VECTOR GALOIS FIELD MULTIPLY SUM AND ACCUMULATE 22-12
VECTOR LOAD COMPLEMENT 22-12
VECTOR LOAD POSITIVE 22-12
VECTOR MAXIMUM 22-13
VECTOR MAXIMUM LOGICAL 22-13
VECTOR MINIMUM 22-13
VECTOR MINIMUM LOGICAL 22-14
VECTOR MULTIPLY AND ADD LOW 22-14
VECTOR MULTIPLY AND ADD HIGH 22-15
VECTOR MULTIPLY AND ADD LOGICAL HIGH 22-15
VECTOR MULTIPLY AND ADD EVEN 22-15
VECTOR MULTIPLY AND ADD LOGICAL EVEN 22-15
VECTOR MULTIPLY AND ADD ODD 22-16
VECTOR MULTIPLY AND ADD LOGICAL ODD 22-16
VECTOR MULTIPLY HIGH 22-16
VECTOR MULTIPLY LOGICAL HIGH 22-17
VECTOR MULTIPLY LOW 22-17
VECTOR MULTIPLY EVEN 22-18
VECTOR MULTIPLY LOGICAL EVEN 22-18
VECTOR MULTIPLY ODD 22-18
VECTOR MULTIPLY LOGICAL ODD 22-18
VECTOR MULTIPLY SUM LOGICAL 22-19
VECTOR NAND 22-20
VECTOR NOR 22-20
VECTOR NOT EXCLUSIVE OR 22-20
VECTOR OR 22-20
VECTOR OR WITH COMPLEMENT 22-21
VECTOR POPULATION COUNT 22-21
VECTOR ELEMENT ROTATE LEFT LOGICAL 22-21
VECTOR ELEMENT ROTATE AND INSERT UNDER MASK 22-22
VECTOR ELEMENT SHIFT LEFT 22-23
VECTOR ELEMENT SHIFT RIGHT ARITHMETIC 22-23
VECTOR ELEMENT SHIFT RIGHT LOGICAL 22-24
VECTOR SHIFT LEFT 22-25
VECTOR SHIFT LEFT BY BYTE 22-25
VECTOR SHIFT LEFT DOUBLE BY BYTE .22-25
VECTOR SHIFT RIGHT ARITHMETIC 22-25
VECTOR SHIFT RIGHT ARITHMETIC BY BYTE 22-26
VECTOR SHIFT RIGHT LOGICAL 22-26
VECTOR SHIFT RIGHT LOGICAL BY BYTE 22-26
VECTOR SUBTRACT 22-26
VECTOR SUBTRACT COMPUTE BORROW INDICATION 22-27
VECTOR SUBTRACT WITH BORROW INDICATION 22-27
VECTOR SUBTRACT WITH BORROW COMPUTE BORROW INDICATION 22-28
VECTOR SUM ACROSS DOUBLEWORD . .22-28
VECTOR SUM ACROSS QUADWORD 22-29
VECTOR SUM ACROSS WORD 22-29
VECTOR TEST UNDER MASK 22-30

Chapter 23, Vector String Instructions 23-1
Vector String Facility 23-1
Instructions 23-1
VECTOR FIND ANY ELEMENT EQUAL 23-1
VECTOR FIND ELEMENT EQUAL 23-3
VECTOR FIND ELEMENT NOT EQUAL 23-4
VECTOR ISOLATE STRING 23-5
VECTOR STRING RANGE COMPARE 23-6

Chapter 24, Vector Floating-Point Instructions 24-1
IEEE Exception Handling 24-1
Result Figures 24-1
Instructions 24-2
VECTOR FP ADD 24-3
VECTOR FP COMPARE SCALAR 24-7
VECTOR FP COMPARE AND SIGNAL SCALAR 24-8
VECTOR FP COMPARE EQUAL 24-9
VECTOR FP COMPARE HIGH 24-11
VECTOR FP COMPARE HIGH OR EQUAL .24-13

VECTOR FP CONVERT FROM FIXED 64-BIT 24-15
VECTOR FP CONVERT FROM LOGICAL 64-BIT 24-16
VECTOR FP CONVERT TO FIXED 64-BIT 24-17
VECTOR FP CONVERT TO LOGICAL 64-BIT 24-19
VECTOR FP DIVIDE 24-21
VECTOR LOAD FP INTEGER 24-23
VECTOR FP LOAD LENGTHENED 24-25
VECTOR FP LOAD ROUNDED 24-26
VECTOR FP MAXIMUM 24-27
VECTOR FP MINIMUM 24-33
VECTOR FP MULTIPLY 24-39
VECTOR FP MULTIPLY AND ADD 24-41
VECTOR FP MULTIPLY AND SUBTRACT. 24-41
VECTOR FP NEGATIVE MULTIPLY AND ADD 24-41
VECTOR FP NEGATIVE MULTIPLY AND SUBTRACT 24-41
VECTOR FP PERFORM SIGN OPERATION 24-43
VECTOR FP SQUARE ROOT. 24-44
VECTOR FP SUBTRACT 24-45
VECTOR FP TEST DATA CLASS IMMEDIATE. 24-46

Chapter 25, Vector Decimal Instructions 25-1
Vector-Packed-Decimal Facility. 25-1
Vector Decimal Control 25-1
Vector Decimal Registers 25-1
Decimal Digits and Signs. 25-2
Instructions 25-2
VECTOR ADD DECIMAL 25-3
VECTOR COMPARE DECIMAL 25-4
VECTOR CONVERT TO BINARY 25-5
VECTOR CONVERT TO DECIMAL 25-6
VECTOR DIVIDE DECIMAL 25-8
VECTOR LOAD IMMEDIATE DECIMAL . . . 25-9
VECTOR MULTIPLY DECIMAL 25-10
VECTOR MULTIPLY AND SHIFT DECIMAL 25-11
VECTOR PACK ZONED 25-12
VECTOR PERFORM SIGN OPERATION DECIMAL 25-13
VECTOR REMAINDER DECIMAL 25-14
VECTOR SHIFT AND DIVIDE DECIMAL . . 25-16
VECTOR SHIFT AND ROUND DECIMAL . 25-17
VECTOR SUBTRACT DECIMAL 25-18
VECTOR TEST DECIMAL 25-20
VECTOR UNPACK ZONED 25-20

Appendix A, Number Representation and Instruction-Use Examples A-1
Number Representation A-2
Binary Integers A-2
Signed Binary Integers A-2
Unsigned Binary Integers A-3
Decimal Integers A-4
Hexadecimal-Floating-Point Numbers A-5
Conversion Example. A-6
Instruction-Use Examples. A-7
Machine Format A-7
Assembler-Language Format A-7
Addressing Mode in Examples A-7
General Instructions A-7
ADD HALFWORD (AH) A-8
AND (N, NC, NI, NR) A-8
NI Example A-8
Linkage Instructions (BAL, BALR, BAS, BASR, BASSM, BSM) A-8
Other BALR and BASR Examples A-10
BRANCH AND STACK (BAKR) A-10
BAKR Example 1 . . . . . .A-11
BAKR Example 2 . . . . . .A-11
BAKR Example 3 . . . . . .A-11
BRANCH ON CONDITION (BC, BCR) A-12
BRANCH ON COUNT (BCT, BCTR) A-12
BRANCH ON INDEX HIGH (BXH) A-12
BXH Example 1. . . . . .A-12
BXH Example 2. . . . . .A-13
BRANCH ON INDEX LOW OR EQUAL (BXLE) A-13
BXLE Example 1. . . . . .A-14
BXLE Example 2. . . . . .A-14
COMPARE AND FORM CODEWORD (CFC) A-14
COMPARE HALFWORD (CH) A-14
COMPARE LOGICAL (CL, CLC, CLI, CLR) . .A-15
CLC Example A-15
CLI Example A-15
CLR Example A-16
COMPARE LOGICAL CHARACTERS UNDER MASK (CLM). A-16
COMPARE LOGICAL LONG (CLCL) A-16
COMPARE LOGICAL STRING (CLST) A-18
CONVERT TO BINARY (CVB) A-18
CONVERT TO DECIMAL (CVD) A-19
DIVIDE (D, DR). A-19
EXCLUSIVE OR (X, XC, XI, XR) A-20
XC Example A-20
XI Example A-21
EXECUTE (EX). A-21
FIND LEFTMOST ONE (FLOGR). A-22
INSERT CHARACTERS UNDER MASK (ICM). A-23

LOAD (L, LR) A-23
LOAD ADDRESS (LA) A-23
LOAD HALFWORD (LH) A-24
MOVE (MVC, MVI) A-24
MVC Example A-24
MVI Example A-25
MOVE INVERSE (MVCIN) A-25
MOVE LONG (MVCL) A-26
MOVE NUMERICS (MVN) A-26
MOVE STRING (MVST) A-27
MOVE WITH OFFSET (MVO) A-27
MOVE ZONES (MVZ) A-28
MULTIPLY (M, MR) A-28
MULTIPLY HALFWORD (MH) A-29
OR (O, OC, OI, OR) A-29
OI Example A-29
PACK (PACK) A-29
ROTATE THEN EXCLUSIVE OR SELECTED BITS A-30
ROTATE THEN INSERT SELECTED BITS . . A-30
ROTATE THEN OR SELECTED BITS A-30
SEARCH STRING (SRST) A-31
SRST Example 1 . . . . . A-31
SRST Example 2 . . . . . A-31
SHIFT LEFT DOUBLE (SLDA) A-31
SHIFT LEFT SINGLE (SLA) A-32
STORE CHARACTERS UNDER MASK (STCM) A-32
STORE MULTIPLE (STM) A-32
TEST UNDER MASK (TM) A-33
TRANSLATE (TR) A-33
TRANSLATE AND TEST (TRT) A-34
UNPACK (UNPK) A-35
UPDATE TREE (UPT) A-36
Decimal Instructions A-36
ADD DECIMAL (AP) A-36
COMPARE DECIMAL (CP) A-36
DIVIDE DECIMAL (DP) A-36
EDIT (ED) A-37
EDIT AND MARK (EDMK) A-38
MULTIPLY DECIMAL (MP) A-39
SHIFT AND ROUND DECIMAL (SRP) A-39
Decimal Left Shift A-39
Decimal Right Shift A-39
Decimal Right Shift and Round A-40
Multiplying by a Variable Power of 10 . . . . A-40
ZERO AND ADD (ZAP) A-40
Hexadecimal-Floating-Point Instructions A-41
ADD NORMALIZED (AD, ADR, AE, AER, AXR) A-41
ADD UNNORMALIZED (AU, AUR, AW, AWR) A-41
COMPARE (CD, CDR, CE, CER) A-42
DIVIDE (DD, DDR, DE, DER) A-42
HALVE (HDR, HER) A-43
MULTIPLY (MD, MDR, MDE, MDER, MXD, MXDR, MXR) A-43
Hexadecimal-Floating-Point-Number Conversion A-44
Fixed Point to Hexadecimal Floating Point A-44
Hexadecimal Floating Point to Fixed Point A-44
Multiprogramming and Multiprocessing Examples A-45
Example of a Program Failure Using OR Immediate A-45
Conditional Swapping Instructions (CS, CDS) A-46
Setting a Single Bit A-46
Updating Counters A-47
Bypassing Post and Wait A-47
Lock/Unlock A-47
Lock/Unlock with LIFO Queuing for Contentions A-48
Lock/Unlock with FIFO Queuing for Contentions A-49
Free-Pool Manipulation A-50
PERFORM LOCKED OPERATION (PLO) . . A-51
Sorting Instructions A-53
Tree Format A-53
Example of Use of Sort Instructions A-55

Appendix B, Lists of Instructions . . . B-1
Instructions Arranged by Name B-5
Instructions Arranged by Mnemonic B-29
Instructions Arranged by Operation Code B-53

Appendix C, Condition-Code Settings C-1

Appendix D, Compression Call Facility D-1
Introduction to Compression Call Facility D-1
Compression and Expansion Dictionaries D-1
Compression-Dictionary Entries D-1
Compression Process D-2
Child and Sibling Characters D-2
Child and Extension-Character Combinations. D-3
Restriction on Identical Child and Sibling Characters D-3
Expansion-Dictionary Entries D-6
Expansion Process D-6
Compressed-Data Symbol Size D-7
Symbol Translation D-8
Order Preservation D-8
Entropy Encoding D-9
Results of Dictionary Errors D-10
Dictionary Formats D-10
Notation D-10
Compression Dictionary D-11

Format-0 Sibling Descriptor D-11
Format-1 Sibling Descriptor D-12
Expansion Dictionary D-12
Character Entry D-12
Format-1 Sibling Descriptor D-13

Appendix G, Table of Powers of 2 . . . G-1

Appendix H, Hexadecimal Tables. . . . H-1

Appendix I, EBCDIC and ISO-8 Codes .I-1
Control Character Representations I-2
Formatting Character Representations I-2
Additional ISO-8 Control Character Representations I-2

Index X-1

Notices

References in this publication to IBM® products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY, 10504-1785 USA.

Trademarks

The following terms are trademarks or registered trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX/ESA
BookMaster
CICS
DB2
Enterprise Systems Architecture/370
Enterprise Systems Architecture/390
Enterprise Systems Connection Architecture
ESA/370
ESA/390
FICON
IBM
IBM z13
IBM z13s
IBM z14
IBM zSystems
IBMLink
ibm.com
MVS/ESA
OS/390
Processor Resource/Systems Manager
PR/SM
Sysplex Timer
System z
System z9
System z10
System/370
VM/ESA
z/Architecture
zEnterprise
z/OS
zPDT
z Systems
z/VM
z9
z10
z13
z14

ANSI is a registered trademark of the American National Standards Institute in the United States, other countries, or both.

© Copyright IBM Corp. 2000, 2017

IEEE is a trademark of the Institute of Electrical and Electronics Engineers, Inc. in the United States, other countries, or both.

Java and all Java-based marks are trademarks or registered trademarks of Oracle America Inc. in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries or both.

Unicode is a registered trademark of Unicode, Incorporated in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other trademarks and registered trademarks are the properties of their respective companies.

Preface

This publication provides, for reference purposes, a detailed z/Architecture™ description.

The publication applies only to systems operating as defined by z/Architecture. For systems operating in accordance with the Enterprise Systems Architecture/390® (ESA/390™) definition, Reference [1.] on page xxix should be consulted.

The publication describes each function at the level of detail needed to prepare an assembler-language program that relies on that function. It does not, however, describe the notation and conventions that must be employed in preparing such a program, for which the user must instead refer to the appropriate assembler-language publication.

The information in this publication is provided principally for use by assembler-language programmers, although anyone concerned with the functional details of z/Architecture will find it useful.

This publication is written as a reference and should not be considered an introduction or a textbook. It assumes the user has a basic knowledge of data-processing systems.

All facilities discussed in this publication are not necessarily available on every model. Furthermore, in some instances the definitions have been structured to allow for some degree of extendibility, and therefore certain capabilities may be described or implied that are not offered on any model. Examples of such capabilities are the use of a 16-bit field in the subsystem-identification word to identify the subchannel number, the size of the CPU address, and the number of CPUs sharing main storage. The allowance for this type of extendibility should not be construed as implying any intention by IBM to provide such capabilities. For information about the characteristics and availability of facilities on a specific model, see the functional characteristics publication for that model.

Largely because this publication is arranged for reference, certain words and phrases appear, of necessity, earlier in the publication than the principal discussions explaining them. The reader who encounters a problem because of this arrangement should refer to the index, which indicates the location of the key description.

The information presented in this publication is grouped in 25 chapters and several appendixes:

Chapter 1, Introduction, highlights the major facilities of z/Architecture.

Chapter 2, Organization, describes the major groupings within the system — main storage, expanded storage, the central processing unit (CPU), the external time reference (ETR), and input/output — with some attention given to the composition and characteristics of those groupings.

Chapter 3, Storage, explains the information formats, the addressing of storage, and the facilities for storage protection. It also deals with dynamic address translation (DAT), which, coupled with special programming support, makes the use of a virtual storage possible.

Chapter 4, Control, describes the facilities for the switching of system status, for special externally initiated operations, for debugging, and for timing. It deals specifically with CPU states, control modes, the program-status word (PSW), control registers, tracing, program-event recording, timing facilities, resets, store status, and initial program loading.

Chapter 5, Program Execution, explains the role of instructions in program execution, looks in detail at instruction formats, and describes briefly the use of the program-status word (PSW), of branching, and of interruptions. It contains the principal description of the advanced address-space facilities that were introduced in ESA/370™. It also details the aspects of program execution on one CPU as observed by other CPUs and by channel programs.

Chapter 6, Interruptions, details the mechanism that permits the CPU to change its state as a result of conditions external to the system, within the system, or within the CPU itself. Six classes of interruptions are identified and described: machine-check interruptions, program interruptions, supervisor-call interruptions, external interruptions, input/output interruptions, and restart interruptions.

Chapter 7, General Instructions, contains detailed descriptions of logical and binary-integer data formats and of most of the unprivileged instructions.

© Copyright IBM Corp. 2000, 2017

Other unprivileged instructions are described in chapters 8-10 and 18-20.

Chapter 8, Decimal Instructions, describes in detail decimal data formats and the decimal instructions.

Chapter 9, Floating-Point Overview and Support Instructions, includes an introduction to the floating-point operations, detailed descriptions of those instructions common to binary-floating-point, decimal-floating-point, and hexadecimal-floating-point operations, and summaries of all floating-point instructions.

Chapter 10, Control Instructions, contains detailed descriptions of all of the semiprivileged and privileged instructions except for the I/O instructions.

Chapter 11, Machine-Check Handling, describes the mechanisms for detecting, correcting, and reporting machine malfunctions.

Chapter 12, Operator Facilities, describes the basic manual functions and controls available for operating and controlling the system.

Chapters 13-17 of this publication provide a detailed definition of the functions performed by the channel subsystem and the logical interface between the CPU and the channel subsystem.

Chapter 13, I/O Overview, provides a brief description of the basic components and operation of the channel subsystem.

Chapter 14, I/O Instructions, contains the description of the I/O instructions.

Chapter 15, Basic I/O Functions, describes the basic I/O functions performed by the channel subsystem, including the initiation, control, and conclusion of I/O operations.

Chapter 16, I/O Interruptions, covers I/O interruptions and interruption conditions.

Chapter 17, I/O Support Functions, describes such functions as channel-subsystem usage monitoring, resets, initial-program loading, reconfiguration, and channel-subsystem recovery.

Chapter 18, Hexadecimal-Floating-Point Instructions, contains detailed descriptions of the hexadecimal-floating-point (HFP) data formats and the HFP instructions.

Chapter 19, Binary-Floating-Point Instructions, contains detailed descriptions of the binary-floating-point (BFP) data formats and the BFP instructions.

Chapter 20, Decimal-Floating-Point Instructions, contains detailed descriptions of the decimal-floating-point (DFP) data formats and the DFP instructions.

Chapter 21, Vector Overview and Support Instructions, describes the vector facility support instructions.

Chapter 22, Vector Integer Instructions, describes the vector facility integer instructions.

Chapter 23, Vector String Instructions, describes the vector facility string instructions.

Chapter 24, Vector Floating-Point Instructions, describes the vector facility floating-point instructions.

Chapter 25, Vector Decimal Instructions, describes the vector decimal instructions.

The *Appendices* include:

- Appendix A: Information about number representation and instruction-use examples
- Appendix B: Lists of the instructions arranged in several sequences
- Appendix C: A summary of the condition-code settings
- Appendix D: Additional information on the compression-call (CMPSC) facility, including processes, dictionary formats, entropy encoding, and order preservation.
- Appendix G: A table of the powers of 2
- Appendix H: Tabular information helpful in dealing with hexadecimal numbers
- Appendix I: A table of EBCDIC and other codes.

Size and Number Notation

In this publication, the letters K, M, G, T, P, and E denote the multipliers 2^{10}, 2^{20}, 2^{30}, 2^{40}, 2^{50}, and 2^{60}, respectively. Although the letters are borrowed from the decimal system and stand for kilo (10^3), mega (10^6), giga (10^9), tera (10^{12}), peta (10^{15}), and exa (10^{18}), they do not have the decimal meaning but

instead represent the power of 2 closest to the corresponding power of 10. Their meaning in this publication is as follows:

Symbol	Value
K (kilo)	$1{,}024 = 2^{10}$
M (mega)	$1{,}048{,}576 = 2^{20}$
G (giga)	$1{,}073{,}741{,}824 = 2^{30}$
T (tera)	$1{,}099{,}511{,}627{,}776 = 2^{40}$
P (peta)	$1{,}125{,}899{,}906{,}842{,}624 = 2^{50}$
E (exa)	$1{,}152{,}921{,}504{,}606{,}846{,}976 = 2^{60}$

The following are some examples of the use of K, M, G, T, and E:

2,048 is expressed as 2K.
4,096 is expressed as 4K.
65,536 is expressed as 64K (not 65K).
2^{24} is expressed as 16M.
2^{31} is expressed as 2G.
2^{42} is expressed as 4T.
2^{53} is expressed as 8P.
2^{64} is expressed as 16E.

When the words “thousand” and “million” are used, no special power-of-2 meaning is assigned to them.

All numbers in this publication are in decimal unless they are explicitly noted as being in binary or hexadecimal (hex).

Bytes, Characters, and Codes

Although the System/360 architecture was originally designed to support the Extended Binary-Coded-Decimal Interchange Code (EBCDIC), the instructions and data formats of the architecture are for the most part independent of the external code which is to be processed by the machine. For most instructions, all 256 possible combinations of bit patterns for a particular byte can be processed, independent of the character which the bit pattern is intended to represent. For instructions which use the zoned format, and for those few instructions which are dependent on a particular external code, the various TRANSLATE instructions may be used to convert data from one code to another code. Thus, a machine operating in accordance with z/Architecture can process EBCDIC, ASCII, or any other code which can be represented in eight or fewer bits per character.

In this publication, unless otherwise specified, the value given for a byte is the value obtained by considering the bits of the byte to represent a binary code. Thus, when a byte is said to contain a zero, the value 00000000 binary, or 00 hex, is meant, and not the value for an EBCDIC character “0,” which would be F0 hex.

Other Publications

1. The IBM ESA/390 architectural mode is described in *IBM Enterprise Systems Architecture/390 Principles of Operation*, SA22-7201.
2. The parallel-I/O interface is described in the publication *IBM System/360 and System/370 I/O Interface Channel to Control Unit Original Equipment Manufacturers’ Information*, GA22-6974.
3. The parallel-I/O channel-to-channel adapter is described in the publication *IBM Enterprise Systems Architecture/390 Channel-to-Channel Adapter for the System/360 and System/370 I/O Interface*, SA22-7091.
4. The Enterprise Systems Connection Architecture® (ESCON®) I/O interface, referred to in this publication along with the FICON I/O interface as the serial-I/O interface, is described in the publication *IBM Enterprise Systems Architecture/390 ESCON I/O Interface*, SA22-7202.
5. The FICON I/O interface is described in the ANSI® standards document *Fibre Channel - Single-Byte Command Code Sets-2 (FC-SB-2)*.
6. The channel-to-channel adapter for the serial-I/O interface is described in the publication *IBM Enterprise Systems Architecture/390 ESCON Channel-to-Channel-Adapter*, SA22-7203.
7. The commands, status, and sense data that are common to all I/O devices that comply with z/Architecture are described in the publication *IBM Enterprise Systems Architecture/390 Common I/O-Device Commands and Self Description*, SA22-7204.
8. The compression facility is described in the publication *IBM Enterprise Systems Architecture/390 Data Compression*, SA22-7208. The z/Architec-

ture form of the COMPRESSION CALL instruction is described in this publication.

9. The interpretive-execution facility is described in the publication *IBM 370-XA Interpretive Execution*, SA22-7095.

10. The load-program-parameter and CPU-measurement facilities are described in the publication *The Load-Program-Parameter and CPU-Measurement Facilities*, SA23-2260.

11. The store-hypervisor-information facility is described in the publication *z/VM CP Programming Services* (SC24-6179).

12. The IBM Enterprise Systems Architecture/Extended Configuration (ESA/XC) virtual-machine architecture is described in *Enterprise Systems Architecture/Extended Configuration Principles of Operation*, SC24-6192.

13. The data-encryption algorithm is described in the publication *Data Encryption Standard*, Federal Information Processing Standards (FIPS) publication 46-3, October 25, 1999.

14. The advanced encryption standard is described in *Advanced Encryption Standard*, FIPS publication 197, November 26, 2001.

15. The secure hash algorithm is described in the publication *Secure Hash Standard*, FIPS publication 180-4, March, 2012.

16. The cipher-feedback (CFB), output-feedback (OFB), and counter (CTR) modes of encryption and decryption are described in *Recommendation for Block Cipher Modes of Operation*, National Institute of Standards and Technology (NIST) Special Publication 800-38A, December, 2001.

17. The Galois/counter mode (GCM) multiplication operation is described in *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, NIST special publication 800-38D, November, 2007.

18. Pseudorandom number generation is described in *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, NIST special publication 800-90A, January, 2012.

19. Entropy sources for random number generation are described in *Recommendation for the Entropy Sources Used for Random Bit Generation*, NIST DRAFT Special Publication 800-90B, August, 2012.

20. Binary floating point is described in *IEEE Standard for Floating-Point Arithmetic*, Institute of Electrical and Electronics Engineers publication IEEE 754-2008.

21. The SHA-3 standard is described in *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, FIPS publication 202, August, 2015.

22. The extensible-markup language (XML) is described in *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, published by the World Wide Web Consortium.

Summary of Changes in Twelfth Edition

The twelfth edition of this publication differs from the previous edition principally by containing the definitions of the following facilities:

- Configuration-z/Architecture-architectural-mode (CZAM) facility
- ESA/390-compatibility-mode (ESA/390CM facility)
- Guarded-storage facility
- Insert-reference-bits-multiple facility
- Instruction-execution-protection facility
- Message-security-assist extension 6
- Message-security-assist extension 7
- Message-security-assist extension 8
- Miscellaneous-instruction-extensions facility 2
- Multiple-epoch facility
- Side-effect-access facility
- Test-pending-external-interruption facility
- Vector-enhancements facility 1
- Vector packed-decimal facility

The twelfth edition also contains the following significant changes relative to the previous edition:

- In the Preface (this section):
  - A summary of Chapter 25 is added.
  - The list of other publications is extended to include additional documents referenced by this document.

- In Chapter 1, “Introduction,” a summary of the new facilities is presented.
- In Chapter 2, “Organization”:
  - The description of CPU registers is extended to include the guarded-storage-facility registers and the epoch index.
  - The description of general and control registers is amended to account for the ESA/390-compatibility-mode facility.
- In Chapter 3, “Storage”:
  - The sections “Address Types,” “Low-Address Protection,” “Prefixing,” “Address Spaces,” “ASN Translation,” “ASN Authorization,” “Dynamic Address Translation,” and “Assigned Storage Locations” are amended to account for the ESA/390 compatibility mode.
  - The sections “Storage Keys” and “Address Summary” are amended to account for the INSERT REFERENCE BITS MULTIPLE facility.
  - The sections “Protection,” “Suppression on Protection,” “Dynamic Address Translation,” and “Assigned Storage Locations” are amended to account for instruction-execution protection. As a result, the section on “Suppression on Protection” is complete rewritten, introducing the side-effect-access facility.
  - Clarification is made to the “Change Recording” section.
- In Chapter 4, “Control”:
  - Amendments are made to the sections “Load State,” “Resets,” “Initial Program Loading,” “Store Status,” and “Signal-Processor Orders” to account for the configuration-z/Architecture-architectural-mode facility.
  - Amendments are made to the sections “Program Status Word,” “Control Registers,” “Tracing,” “Program Event Recording,” “Resets,” “Store Status,” and “Signal-Processor Orders” to account for the ESA/390-compatibility-mode facility.
  - Other amendments and clarifications are made to the “Program Event Recording” section.
  - Amendments are made to the section “Tracing” and “Timing” to account for the multiple-epoch facility. Additionally, a new leap second is defined in the section “Timing.”
  - The section “Guarded-Storage Facility” is added.
  - The section “Facility Indications” is amended to describe new facility-indication bits.
- In Chapter 5, “Program Execution”:
  - Amendments are made to the sections “Instruction Formats” and “Block-Concurrent References” to account for the vector-enhancements facility 1 and the vector-packed-decimal facility.
  - The section “Formation of the Intermediate Value” is amended to account for the BRANCH INDIRECT ON CONDITION instruction.
  - The sections “Condition-Code Alternative to Interruptibility” and “Multiple-Access References” are amended to account for the message-security-assist extension 8.
  - Clarification is made to the section “Exceptions to Nullification and Suppression.”
  - The section “Transactional-Execution Facility” is updated to account for the guarded-storage facility. Other clarifications and corrections are also included in the description of the transactional-execution facility.
  - Amendments are made to the section “Monitor-Event Counting.”
  - The ESA/390 compatibility mode is described in a new section.
  - Amendments are made to the section “Relatino between Storage-Key Accesses” to account for the insert-reference-bits-multiple facility.
- In Chapter 6, “Interruptions”:
  - Multiple sections are amended to account for the ESA/390-compatibility-mode facility.
  - Various subsections of the section “Program-Interruption Conditions” are amended to account for the guarded-storage, instruction-execution-protection, message-security-

assist extension 6, message-security-assist extension 8, and vector-packed-decimal facilities.

  - Various other clarifications are made to the sections “Instruction-Length Code” and “Exceptions Associated with the PSW.”

- In Chapter 7, “General Instructions”:

  - The instructions of the guarded-storage facility, miscellaneous-instruction-extensions facility 2, and message-security-assist extension 8 are described. (Of necessity, the original miscellaneous-instruction-extensions facility is now called the miscellaneous-instruction-extensions facility 1.)

  - New functions are defined for the KIMD, KLMD, and PRNO instructions in support of the message-security-assist extensions 6 and 7.

  - Numerous other clarifications are made to the descriptions of each of the message-security-assist instructions and to the section “Protection of Cryptographic Keys. All of the figures in these sections have been redrawn.

  - The descriptions of multiple instructions are amended to account for the ESA/390-compatibility-mode facility.

  - Clarification is made to the description of the COMPARE AND FORM CODEWORD instruction.

  - Clarification is made to the description of the CONVERT UTF-16 TO UTF-32, CONVERT UTF-16 TO UTF-8, CONVERT UTF-32 TO UTF-16, and CONVERT UTF-32 TO UTF-8 instructions.

  - The description of the EXTRACT CPU ATTRIBUTE instruction is extended to provide CPU attributes.

  - A programming note is added to all instructions having a query function, advising of the proper use of the function.

  - The descriptions of STORE CLOCK, STORE CLOCK EXTENDED, and STORE CLOCK FAST are amended to account for the multiple-epoch facility and the addition of a new leap second.

- In Chapter 8, “Decimal Instructions”:

  - Clarification is made to the introductory material and to the sections “Decimal Codes,” “Other Instructions for Decimal Operands,” and “General-Operand Data Exception.”

  - In the description of EDIT and EDIT AND MARK, clarification is made to the setting of condition code 2.

  - In the description of TEST DECIMAL, clarification is made to the description of condition codes 1 and 2.

- In Chapter 9, “Floating-Point Overview and Support Instructions”:

  - Clarification is made to the section “Hexadecimal-Floating-Point (HFP).”

  - Clarification is made to the section “IEEE Computational Operations” and to the description of the PERFORM FLOATING POINT OPERATION instruction with regards to subnormal numbers and exception handling.

  - The section “Impacts on ESA/390” is extended to account for the ESA/390-compatibility-mode facility.

  - Item 1 in the section “Impacts of the Floating-Point Extension Facility” is corrected.

- In Chapter 10, “Control Instructions”:

  - Descriptions of the INSERT REFERENCE BITS MULTIPLE, and TEST PENDING EXTERNAL INTERRUPTION instructions are added.

  - The descriptions of BRANCH AND SET AUTHORITY, EXTRACT STACKED STATE, INVALIDATE PAGE TABLE ENTRY, LOAD ADDRESS SPACE PARAMETERS, LOAD CONTROL, LOAD PSW, LOAD PSW EXTENDED, LOAD REAL ADDRESS, RESUME PROGRAM, SET PREFIX, SET SYSTEM MASK, STORE PREFIX, STORE REAL ADDRESS, STORE THEN OR SYSTEM MASK, and TEST ACCESS instructions are amended to account for the ESA/390-compatibility-mode facility

  - Various clarifications are made to the description of the PERFORM FRAME MANAGEMENT FUNCTION instruction.

  - New function codes are described for the PERFORM TIMING FACILITY FUNCTION instruction in support of the multiple-epoch facility. Additionally, the 27th leap second is described.
  - A programming note is added to the description of the SET CLOCK instruction in support of the multiple-epoch facility.
  - The description of the STORE CPU ID instruction is complete rewritten.
  - Various changes are made to the description of the STORE SYSTEM INFORMATION instruction, including a new field in the SYSIB 1.1.1, corrections to the descriptions of the adjustment factors in the SYSIB 1.2.2, the addition of the virtual-server ID and virtual-server name in the SYSIB 2.2.2, and the addition of the extended-VM-name field in the SYSIB 3.2.2.
  - The description of the TEST PROTECTION instruction is amended to account for the instruction-execution-protection facility.
  - The description of the TRACE instruction is amended to account for the multiple-epoch facility.
- In Chapter 11, “Machine-Check Handling”:
  - The section “Effects of CPU Retry” is amended to account for the addition of the message-security-assist extension 8.
  - The sections “Validation” and “Invalid CBC in Registers” are amended, simplifying the description of register validation.
  - The sections “Interruption Action,” “Machine-Check-Interruption Code,” “Machine-Check Extended Interruption Information,” and “Machined-Check Extended Save Area (MCESA)” are amended to account for the addition of the guarded-storage facility and the ESA/390-compatibility-mode facility.
- In Chapter 12, “Operator Facilities,” various sections are amended to account for the addition of the ESA/390-compatibility-mode facility.
- In Chapter 14, “I/O Instructions”:
  - Clarification is made to the Figure 14-1, “Summary of I/O Instructions.”
- In Chapter 15, “Basic I/O Functions”:
  - In the section “Command-Mode ORB”, a correction is made to the description of the prefetch control.
  - In the section “CCW Channel Program Chaining”, a correction is made to the figure “Subchannel Chaining Action.”
- In Chapter 16, “I/O Interruptions, in the section “Extended-Status Word,” the illustration of the subchannel logout is corrected.
- In Chapter 17, “I/O Support Functions”:
  - In the section “Externally Initiated Functions,” the descriptions of CCW-type IPL and List-Directed IPL are amended to account for the ESA/390-compatibility-mode (ESA/390CM) and configuration-z/Architecture-architectural-mode (CZAM) facilities, and clarification is made to the contents of the ORB used by a CCW-type IPL.
- In Chapter 19, “Binary-Floating-Point Instructions”:
  - Terminology has been updated to conform to that of Reference [20.] on page xxx.
  - In the figure “Summary of BFP Instructions,” clarification is made to the applicability of the symbols Xi, Xo, Xu, Xx, and Xz (they are data exceptions, as opposed to the same symbols in Chapter 24, where they are vector-processing exceptions).
- In Chapter 20, “Decimal-Floating-Point Instructions,” in the figure “Summary of DFP Instructions”:
  - The instruction format is corrected for CONVERT FROM ZONED and CONVERT TO ZONED.
  - Clarification is made to the applicability of the symbols Xi, Xo, Xu, Xx, and Xz (they are data exceptions, as opposed to the same symbols in Chapter 24, where they are vector-processing exceptions).
- In Chapter 21, “Vector Overview and Support Instructions”:
  - The instructions and instruction enhancements added by the vector-enhancements facility 1 are described. The instructions

added by the vector-packed-decimal facility are described.

  - An optional alignment-hint operand is added to the VECTOR LOAD, VECTOR LOAD MULTIPLE, VECTOR STORE, and VECTOR STORE MULTIPLE.

- In Chapter 22, “Vector Integer Instructions”:
  - The instructions added by the vector-enhancements facility 1 are described.
  - The description of the VECTOR POPULATION COUNT instruction is extended to accommodate multiple element sizes.
- In Chapter 24, “Vector Floating-Point Instructions”:
  - In the section “IEEE Exception Handling,” clarification is made to the contents of the VXC.
  - The mnemonics for VECTOR FP LOAD LENGTHENED and VECTOR FP LOAD ROUNDED are changed to VFLL and VFLR, respectively. The former mnemonics VLDE and VLED, respectively, are deprecated but retained for compatibility purposes.
  - The instructions and instruction enhancements added by the vector-enhancements facility 1 are described.
- Chapter 25, “Vector Decimal Instructions” is added.
- Appendix D, “Compression Call Facility” is added.

Summary of Changes in Eleventh Edition

The eleventh edition of this publication differs from the previous edition principally by containing the definitions of the following facilities:

- Decimal-floating-point packed-conversion facility
- FCX-bidirectional-data-transfer facility
- Load-and-zero-rightmost-byte facility
- Load/store-on-condition facility 2
- Message-security-assist extension 5
- Multithreading facility
- Vector facility for z/Architecture

The eleventh edition also contains the following significant changes relative to the previous edition:

- In the Preface:
  - Summaries of Chapters 21-24 are added.
  - The list of other publications is extended to include additional documents referenced by this document.
- In Chapter 1, “Introduction,” a summary of the new facilities is presented.
- In Chapter 2, “Organization”:
  - New sections are added, describing CPU types, multithreading, and vector registers.
  - Clarification is made as to the contents of bit positions 0-31 of general registers while the CPU is in the ESA/390 mode.
- In Chapter 3, “Storage”:
  - Clarification is made as to the applicability of key-controlled protection’s to the program-interruption TDB.
  - Clarification is made as to the applicability of DAT to certain instructions and events.
  - Bits 62-63 of the format-1 RTTE and STE are made available for programming.
  - The vector-exception code and machine-check-extended-save-area address are added to the assigned-storage locations.
  - Because of their pervasive use by z/OS, three additional assigned-storage locations are made available for use by programming.
- In Chapter 4, “Operation”:
  - Additional constraints are placed on the use of CPU address-compare controls is program-event recording is used.
  - Additional bits are defined in control registers 0 and 2.
  - Applicability of VRV-format instructions (used by the vector facility) are added to the description of PER zero-address detection.
  - In the section “Timing”:

    - Numerous clarifications are made to the section “Setting and Inspecting the Clock.”
    - Leap second 26 is documented.
    - Descriptions of hardware-based and offset-based TOD-clock steering are added.
    - Description of the UTC-information block (UIB) is added.
    - Clarification is made to the description of the CPU timer.
  - The descriptions of various reset operations are updated to account for the addition of the multithreading and vector facilities.
  - Clarification is made to the description of CCW-based IPL.
  - In the section “Multiprocessing,” the process of CPU-address expansion and contraction is described.
  - In the section “CPU Signaling and Response,” various changes are described in support of the multithreading facility:
    - The SIGP set-multithreading order is described.
    - Various changes are made to other SIGP orders in support of multithreading.
  - New facility indications are described.
- In Chapter 5, “Program Execution”:
  - Instruction formats are added, most in support of vector-facility for z/Architecture instructions.
  - Clarification is made to the description of instruction nullification and suppression.
  - The section “Condition-Code Alternative to Interruptibility” is updated to account for new instructions.
  - Clarification is added to the description of the TBEGIN-specified TDB.
  - The TDB is updated to include the vector-exception code (VXC).
  - A programming note is added, describing the proper use of branch instructions following a TRANSACTION BEGIN.
  - The section “Multiple-Access References” is updated to account for added instructions.
  - The section “Block-Concurrent References” is updated to account for the vector-facility for z/Architecture instructions.
  - A programming note is added to the section “Storage-Key Accesses”, describing behavior of the nonquiescing key-setting operation.
- In Chapter 6, “Interruptions”:
  - The vector-processing interruption is described.
  - STP timing-alert interruptions are defined to be floating interruptions.
  - Clarification is made to the section “Exceptions Associated with the PSW.”
  - Clarification is made to the description of external interruptions.
  - Clarification is made to the description of the warning-track interruption.
  - Because it is used for other instructions besides decimal instructions, data-exception code 0 is renamed *general-operand data exception* (it was formerly *decimal-operand data exception*).
  - Descriptions of the vector-exception code, vector-instruction data exception code, and the vector-processing exception are added.
  - The descriptions of various program-interruption codes are updated to account for new instructions.
- In Chapter 7, “General Instructions”:
  - A programming note is added to the BRANCH PREDICTION PRELOAD and BRANCH PREDICTION RELATIVE PRELOAD instructions.
  - The CIPHER MESSAGE WITH CFB and CIPHER MESSAGE WITH OFB instructions are renamed to the more descriptive CIPHER MESSAGE WITH CIPHER FEEDBACK and CIPHER MESSAGE WITH OUTPUT FEEDBACK, respectively. The

mnemonics for these instructions are unchanged.

– Instructions for the following facilities are described:
  - Load-and-zero-rightmost-byte facility
  - Load/store-on-condition facility 2 (note, the former load/store-on-condition facility is renamed load-store-on-condition facility 1)
  - Message-security-assist extension 5
  - Vector facility for z/Architecture (the LOAD COUNT TO BLOCK BOUNDARY is the only general instruction in this facility; the remaining instructions appear in Chapters 21-24)

– Numerous clarifications and corrections are made to the descriptions of the message-security instructions: CIPHER MESSAGE, CIPHER MESSAGE WITH CHAINING, CIPHER MESSAGE WITH CIPHER FEEDBACK, CIPHER MESSAGE WITH COUNTER, CIPHER MESSAGE WITH OUTPUT FEEDBACK, COMPUTE INTERMEDIATE MESSAGE DIGEST, COMPUTE LAST MESSAGE DIGEST, COMPUTE MESSAGE AUTHENTICATION CODE, and PERFORM CRYPTOGRAPHIC COMPUTATION. Additional clarification is made to the section "Protection of Cryptographic Keys."

– Clarification is made to the description of access exception for instructions of the interlocked-access facility 1. Also, additional extended mnemonics are described for the instructions of the facility.

– A programming note is added to the description of PERFORM LOCKED OPERATION.

– Programming notes are added to the SHIFT LEFT SINGLE LOGICAL and SHIFT RIGHT SINGLE LOGICAL instructions.

– Clarification is added to programming note 5 for STORE CLOCK and STORE CLOCK FAST.

– Leap second 26 is documented.

– Various changes and clarifications are made to the description of TRANSACTION BEGIN.

- In Chapter 8, "General Instructions":
  – The description of the signed-packed-decimal format is amended to accommodate the addition of the DFP-packed-conversion facility.
  – The term *general-operand data exception* replaces the former *decimal-operand data exception*.
- In Chapter 9, "Floating-Point Overview and Support Instructions":
  – References to the ANSI/IEEE binary-floating-point standard are updated.
  – Numerous clarifications are made to the description of the PERFORM FLOATING POINT OPERATION instruction and affiliated text.
- In Chapter 10, "Control Instructions":
  – In the description of BRANCH AND SET AUTHORITY, clarification is made to the discussion of special conditions.
  – In the description of LOAD PAGE TABLE ENTRY ADDRESS, a correction is made to programming notes 2 and 3.
  – Clarification is provided in the description of the LOAD REAL ADDRESS instruction.
  – In the description of PERFORM TIMING FACILITY FUNCTION:
    - New functions are added.
    - Numerous other clarifications are made to the instruction description.
    - Leap second 26 is documented.
  – The description of SIGNAL PROCESSOR is changed to accommodate the vector facility for z/Architecture and multithreading facility.
  – The description of STORE SYSTEM INFORMATION is updated to accommodate the multithreading facility.
  – The description of TEST PROTECTION is updated to account for the EDAT-2 facility.
- In Chapter 11, "Machine-Check Handling":

  - Various sections are updated to accommodate the addition of the vector facility for z/Architecture.
  - The vector-register-validity bit is added to the MCIC.
  - The machine-check extended save area is described.
- In Chapter 12, “Operator Facilities”:
  - Various constraints are put on the use of address-compare controls with using the program-event-recording (PER) facility.
  - Alter-and-display controls accommodate the vector registers.
  - A CPUs-per-core indicator is added in support of the multithreading facility.
  - The section “Multithreading Considerations” is added.
- In Chapter 13, “I/O Overview,” the bidirectional-data-transfer facility and transfer-control-area extension (TCAX) are described.
- In Chapter 15, “Basic I/O Functions”:
  - Description of the bidirectional-data-transfer operations is added to the various transport-mode I/O control structures.
  - Description of the transport-control-area extension (TCAX) is added.
  - Description of the transfer-TCA-Extension (TTE) DCW is added.
- In Chapter 16, “I/O Interruptions,” program-check conditions for bidirectional-data-transfer operations are added.
- In Chapter 19, “Binary-Floating-Point Instructions,” references to the ANSI/IEEE binary-floating-point standard are updated.
- In Chapter 20, “Decimal-Floating-Point Instructions”:
  - The instructions of the DFP-packed-conversion facility are described.
  - The term *general-operand data exception* replaces the former *decimal-operand data exception*.
  - In the description of the CONVERT TO ZONED instruction, clarification is made to the description of the zone control.
- Chapters 21-24 are added, describing the instructions of the vector facility for z/Architecture.
- Numerous other minor clarifications and corrections are included.

Summary of Changes in Tenth Edition

The tenth edition of this publication differs from the previous edition principally by containing the definitions of the following facilities:

- Decimal-floating-point zoned-conversion facility
- Enhanced-DAT facility 2
- Execution-hint facility
- Interlocked-access facility 2
- Load-and-trap facility
- Local-TLB-clearing facility
- Miscellaneous-instruction-extensions facility 1
- PER zero-address-detection facility
- Processor-assist facility
- Side-effect-access facility
- Transactional-execution facility
- Warning-track-interruption facility

The tenth edition contains the following significant changes relative to the previous edition:

- In Chapter 1, “Introduction,” a summary of the new facilities is presented.
- In Chapter 3, “Storage:”
  - Changes introduced by the enhanced-DAT facility 2 are added.
  - Changes introduced by the execution-hint facility are added.
  - Changes introduced by the side-effect-access facility are described.
  - Changes introduced by the transactional-execution facility are added.
- In Chapter 4, “Control:”

  - Changes introduced by the enhanced-DAT facility 2 are added.
  - Changes introduced by the execution-hint facility are added.
  - Changes introduced by the PER zero-address-detection facility are added.
  - Changes introduced by the transactional-execution facility are added.
  - Leap second 25 is documented.
  - Facility indications for the new facilities are added.
- In Chapter 5, “Program Execution:”
  - New instruction formats are defined.
  - Changes introduced by the enhanced-DAT facility 2 are added.
  - Changes introduced by the execution-hint facility are added.
  - Changes introduced by the interlocked-access facility 2 are added.
  - Changes introduced by the transactional-execution facility are added.
  - Clarification is made to the access-register-translation process.
  - Clarification is made to the handling of PER events in the enhanced-monitor counting array.
- In Chapter 6, “Interruptions:”
  - Changes introduced by the load-and-trap facility are added.
  - Changes introduced by the transactional-execution facility are added.
  - Changes introduced by the warning-track-interruption facility are added.
  - Clarification is made to the description of the measurement-alert external interruption.
- In Chapter 7, “General Instructions:”
  - Changes introduced by the execution-hint facility are added.
  - Changes introduced by the interlocked-access facility 2 are added. The original interlocked-access facility is now called the interlocked-access facility 1.
  - Changes introduced by the load-and-trap facility are added.
  - Changes introduced by the miscellaneous-instruction-extension facility are added.
  - Changes introduced by the processor-assist facility are added.
  - Changes introduced by the transactional-execution facility are added.
  - Changes introduced by the PER zero-address-detection facility are added.
  - The EXTRACT CACHE ATTRIBUTE instruction is renamed EXTRACT CPU ATTRIBUTE.
- In Chapter 8, “Decimal Instructions:”
  - Changes introduced by the decimal-floating-point zoned-conversion facility are added.
  - Changes introduced by the transactional-execution facility are added.
- In Chapter 9, “Floating-Point Overview and Support Instructions:”
  - Changes introduced by the transactional-execution facility are added.
  - Miscellaneous corrections are made to Figure 9-21.
  - Miscellaneous clarifications and corrections are made to the description of PERFORM FLOATING POINT OPERATION.
- In Chapter 10, “Control Instructions:”
  - Changes introduced by the enhanced-DAT facility 2 are added.
  - Changes introduced by the local-TLB-clearing facility are added.
  - Changes introduced by the transactional-execution facility are added.
  - Changes introduced by the PER zero-address-detection facility are added.
  - Clarification is made to the description of EXTRACT STACKED STATE.

  - Clarification is made to the description of STORE CPU ID and STORE SYSTEM INFORMATION.

- In Chapter 12, “Operator Facilities,” clarification is made to the description of the load-unit-address controls.
- In Chapter 13, “I/O Overview:”
  - Clarification is made to the description of the device number.
- In Chapter 14, “I/O Instructions:”
  - Changes introduced by the transactional-execution facility are added.
- In Chapter 15, “Basic I/O Functions:”
  - Various clarifications are made to the description of the output-count field of the transport-control word.
  - Various clarifications and changes are made to the device-command word, including the description of the suppress-length-indication flag.
  - The transport-command DCW is described.
  - Clarifications are made to the description of the Transfer-CBC-offset-block DCW.
- In Chapter 16, “I/O Interruptions:”
  - The incorrect-length subchannel-status indication is described.
- In Chapter 17, “I/O Support Functions:”
  - The interrupt-delay-time and I/O-priority-delay-time fields are described in the measurement block.
- In Chapter 18, “Hexadecimal-Floating-Point Instructions:”
  - Changes introduced by the transactional-execution facility are added.
  - Clarifications are made to the description of CONVERT TO FIXED.
- In Chapter 19, “Binary-Floating-Point Instructions:”
  - Changes introduced by the transactional-execution facility are added.
  - Clarifications and corrections are made to the description of CONVERT TO FIXED.
  - Clarifications and corrections are made to the description of CONVERT TO LOGICAL.
- In Chapter 20, “Decimal-Floating-Point Instructions:”
  - Changes introduced by the decimal-floating-point zoned-conversion facility are added.
  - Changes introduced by the transactional-execution facility are added.

The tenth edition also contains numerous minor corrections and clarifications.

Summary of Changes in Ninth Edition

The ninth edition of this publication differs from the previous edition principally by containing the definitions of the following facilities:

- CMPSC-enhancement facility
- Distinct-operands facility
- Enhanced-monitor facility
- Fast-BCR-serialization facility
- Floating-point extension facility
- High-word facility
- Interlocked-access facility 1
- IPTE-range facility
- Load/store-on-condition facility
- Message-security-assist extension 3
- Message-security-assist extension 4
- Nonquiescing key-setting facility
- Population-count facility
- The Reset-reference-bits-multiple facility

The ninth edition contains the following significant changes relative to the previous edition:

- In Chapter 3, “Storage:”
  - Changes introduced by the reset-reference-bits-multiple are added to various sections.
  - Changes introduced by the enhanced-monitor facility are added.
  - Changes introduced by the access-exception-fetch/store-indication facility are added.

  - Clarification is added to the description of the translation-exception identification.

- In Chapter 4, “Control:”
  - Changes introduced by the enhanced-monitor facility are added.
  - Clarification is made to the description of the PER ASCE identification (AI) and PER access identification (PAID).
  - Changes introduces by the high-word facility are added.
  - Descriptions of the new facility indications are added.
- In Chapter 5, “Program Execution:”
  - A significant editorial change has been made to the description of the instruction formats. Instruction formats having variations in assembler syntax or field content are designated by an alphabetic suffix which is also reflected in the instruction descriptions in chapters 7-10, 14, and 18-24.
  - Changes introduced by the enhanced-monitor facility are added.
  - Clarification is made to the description of suppression.
  - Changes introduced by the message-security assist extensions 3 and 4 are added.
  - Changes introduced by the nonquiescing-key-setting facility are added, including the description of quiescing.
  - Changes introduced by the interlocked-access facility are added, including the description of an interlocked-fetch reference and specific-operand serialization.
- In Chapter 6, “Interruptions:”
  - Changes introduced by the enhanced-monitor facility are added.
  - Changes introduced by the floating-point extension facility are added.
  - Clarification is provided regarding the handling of data exceptions resulting from the compare-and-trap instructions.
- In Chapter 7, “General Instructions:”
  - Clarification is made as to the content of the chapter.
  - Changes introduced by the distinct-operands facility are added.
  - Changes introduced by the high-word facility are added.
  - Changes introduced by the interlocked-access facility are added.
  - Changes introduced by the load/store-on-condition facility are added.
  - Changes introduced by the message-security-assist extension 3 are added, including a new section on the protection of cryptographic keys.
  - Changes introduced by the message-security-assist extension 4 are added.
  - Changes introduced by the population-count facility are added.
  - Changes introduced by the instruction-format clarifications are added.
  - Clarifications are made to the description of COMPARE AND SWAP AND STORE.
  - Changes introduced by the CMPSC-enhancement facility are added.
  - Extended assembler mnemonics are defined for the ROTATE THEN AND SELECTED BITS, ROTATE THEN OR SELECTED BITS, and ROTATE THEN EXCLUSIVE OR SELECTED BITS instructions.
  - Clarification is made to the descriptions of the SEARCH STRING and SEARCH STRING UNICODE instructions.
- In Chapter 8, “Decimal Instructions:”
  - Changes introduced by the instruction-format clarifications are added.
  - Clarification is made to the description of the EDIT and EDIT AND MARK instructions.
- In Chapter 9, “Floating-Point Overview and Support Instructions:”
  - Changes introduced by the instruction-format clarifications are added.

  - Changes introduced by the floating-point-extension facility are added.

- In Chapter 10, “Control Instructions:”
  - Clarification is made as to the content of the chapter; specifically, the chapter also contains the description of certain nonprivileged instructions.
  - Changes introduced by the instruction-format clarifications are added.
  - Changes introduced by the IPTE-range facility are added.
  - Changes introduced by the message-security-assist extension 3 are added.
  - Changes introduced by the nonquiescing key-setting facility are added.
  - Changes introduced by the reset-reference-bits-multiple facility are added.
  - Changes are made to the STORE SYSTEM INFORMATION instruction with regards to topology information returned.
- In Chapter 14, “Basic I/O Functions,” the CBC-offset block is added for the fibre-channel-extensions facility.
- In Chapter 18, “Hexadecimal-Floating Instructions,” changes introduced by the instruction-format clarifications are added.
- In Chapter 19, “Binary-Floating-Point Instructions:”
  - Changes introduced by the instruction-format clarifications are added.
  - Changes introduced by the floating-point-extension facility are added.
- In Chapter 20, “Decimal-Floating-Point Instructions:”
  - Changes introduced by the instruction-format clarifications are added.
  - Changes introduced by the floating-point-extension facility are added.

The ninth edition also contains numerous minor corrections and clarifications.

Summary of Changes in Eighth Edition

The eighth edition of this publication differs from the previous edition principally by containing the definitions of the following facility:

- Fibre-channel-extensions (FCX) facility

The eighth edition contains the following significant changes relative to the previous edition:

- In Chapter 1, “Introduction,” a summary of the fibre-channel-extensions facility is added.
- In Chapter 4, “Control,” the clock setting for the leap second on January 1, 2009 is added to the section “Timing.”
- In Chapter 7, “General Instructions,” the clock setting for the leap second on January 1, 2009 is added to the description of STORE CLOCK EXTENDED.
- In Chapter 13, “I/O Overview,” changes introduced by the fibre-channel-extensions facility are added to various sections.
- In Chapter 14, “I/O Instructions,” changes introduced by the fibre-channel-extensions facility are added to various sections.
- In Chapter 15, “Basic I/O Functions,” changes introduced by the fibre-channel-extensions facility are added to various sections.
- In Chapter 16, “I/O Interruptions,” changes introduced by the fibre-channel-extensions facility are added to various sections.
- In Chapter 17, “I/O Support Functions,” changes introduced by the fibre-channel-extensions facility are added to various sections.

The eighth edition also contains numerous minor corrections and clarifications.

Summary of Changes in Seventh Edition

The seventh edition of this publication differs from the previous edition principally by containing the definitions of the following facilities:

- Compare-and-swap-and-store facility 2
- Configuration-topology facility
- Enhanced-DAT facility
- Execute-extensions facility
- General-instructions-extension facility
- Message-security-assist extension 2
- Move-with-optional-specifications facility
- Parsing-enhancement facility
- Restore-subchannel facility

The seventh edition contains the following significant changes relative to the previous edition:

- In Chapter 3, “Storage:”
  - Changes introduced by the enhanced-DAT facility are added to various sections.
  - Changes introduced by the enhanced suppression-on-protection function are added
  - Changes introduced by the execute-extensions facility are added to various sections.
  - Changes introduced by the general-instructions-extension facility are added to the section “Assigned Storage Locations.”
  - Changes introduced by the move-with-optional-specifications facility are added to the section “Protection.”
- In Chapter 4, “Control:”
  - Changes introduced by the enhanced-DAT facility are added to various sections.
  - Changes introduced by the execute-extensions facility are added to various sections.
  - Changes introduced by the general-instructions-extension facility are added to various sections.
  - New facility bits for the compare-and-swap-and-store facility 2, configuration-topology facility, enhanced-DAT facility, execute-extensions facility, general-instructions-extension facility, move-with-optional-specifications facility, and parsing-enhancement facility are added to the section “Facility Indications.”
- In Chapter 5, “Program Execution:”
  - In the section “Instruction Formats,” the RIS, RRS, and SIL formats are added. Four new versions of the RIE instruction format and one new version of the RXY format are added.
  - Changes introduced by the enhanced-DAT facility are added to various sections.
  - Changes introduced by the execute-extensions facility are added to various sections.
  - Changes introduced by the general-instructions-extension facility are added to various sections.
  - Changes introduced by the move-with-optional-specifications facility are added to various sections.
  - Changes introduced by the parsing-enhancement facility are added to the section “Multiple-Access References.”
- In Chapter 6, “Interruptions:”
  - Changes introduced by the compare-and-swap-and-store facility 2 are added to the section “Specification Exception.”
  - Changes introduced by the enhanced-DAT facility are added to various sections.
  - Changes introduced by the execute-extensions facility are added to various sections.
  - Changes introduced by the general-instructions-extension facility are added to various sections.
  - Changes introduced by the move-with-optional-specifications facility are added to various sections.
  - Changes introduced by the parsing-enhancement facility are added to section “Specification Exception.”
- In Chapter 7, “General Instructions:”
  - For various instruction descriptions in which multiple instruction formats are present,

headings have been added to distinguish one format from another.

  - Changes introduced by the compare-and-swap-and-store facility 2 are added to the description of the COMPARE AND SWAP AND STORE instruction.

  - Changes introduced by the message-security-assist extension 2 are added to the descriptions of the CIPHER MESSAGE, CIPHER MESSAGE WITH CHAINING, COMPUTE INTERMEDIATE MESSAGE DIGEST and COMPUTE LAST MESSAGE DIGEST instructions.

  - Programming notes are added to the description of the UTF conversion instructions (CU12, CU14, CU21, CU24, CU41, and CU42) indicating that the instructions support big-endian encoding only.

  - Changes introduced by the execute-extensions facility are added to various sections, including the description of the EXECUTE RELATIVE LONG instruction.

  - Changes introduced by the general-instructions-extension facility are added to various sections, including the descriptions of the facility's 72 new instructions.

  - The description of the MOVE LONG UNICODE instruction is amended such that an odd length specification is permitted.

  - Changes introduced by the parsing-enhancement facility are added to various sections, including the descriptions of the TRANSLATE AND TEST EXTENDED and TRANSLATE AND TEST REVERSE EXTENDED instructions.

- In Chapter 10, "Control Instructions:"

  - Changes introduced by the move-with-optional-specifications facility are added to various sections, including the description of the MOVE WITH OPTIONAL SPECIFICATIONS instruction.

  - Changes introduced by the enhanced-DAT facility are added to various sections, including the description of the PERFORM FRAME MANAGEMENT FUNCTION instruction.

  - Changes introduced by the configuration-topology facility are added to various sections, including the description of the PERFORM TOPOLOGY FUNCTION instruction and enhancements to the STORE SYSTEM INFORMATION instruction.

  - Changes introduced by the execute-extensions facility are added to various sections.

  - The STORE SYSTEM INFORMATION instruction is enhanced to include the following:

    - Additional information returned in SYSIB 1.1.1.

    - The definition of SYSIB 15.1.2 in support of the configuration-topology facility.

- In Chapter 11, "Machine-Check Handling:"

  - Changes introduced by the enhanced-DAT facility are added to various sections.

  - Changes introduced by the execute-extensions facility are added to various sections.

- In Chapter 13, "I/O Overview," changes introduces by the multiple-subchannel set facility are described.

- In Chapter 14, "I/O Instructions," changes introduces by the multiple-subchannel set facility are described.

- In Chapter 17, "I/O Support Functions," changes introduces by the multiple-subchannel set facility are described.

The seventh edition also contains numerous minor corrections and clarifications.

Summary of Changes in Sixth Edition

The sixth edition of this publication differs from the previous edition principally by containing the definitions of the following facilities:

- Compare-and-swap-and-store facility
- Conditional-SSKE facility
- Decimal-floating-point facility
- Decimal-floating-point-rounding facility
- Extract-CPU-time facility

- Floating-point-support-sign-handling facility
- FPR-GR-transfer facility
- IEEE-exception-simulation facility
- PFPO facility

The sixth edition contains the following significant changes relative to the previous edition:

- In Chapter 3, “Storage”:
  - Changes introduced by the conditional-SSKE facility are added to various sections.
  - In the “Assigned Storage Locations” section, the descriptions of the various fields now include hexadecimal as well as decimal locations.
  - Clarification is made to the definition of the exception access identification and PER access identification at locations 160 and 161, respectively.
- In Chapter 4, “Control”:
  - Changes introduced by the extract-CPU-time facility are added to the section “CPU Timer.”
  - New facility bits for the compare-and-swap-and-store facility, decimal-floating-point facility, decimal-floating-point performance, extract-CPU-time facility, floating-point-support-enhancement facilities (FPR-GR-transfer, FPS-sign-handling, and DFP-rounding), and PFPO (PERFORM FLOATING-POINT OPERATION) facility are added to the section “Facility Indications.”
- In Chapter 5, “Program Execution”:
  - In the section “Instruction Formats”, the RRR and SSF formats are added.
  - Changes introduced by the compare-and-swap-and-store facility are added to various sections.
  - Changes introduced by the extract-CPU-time facility are added to the “Consistency Specification” section.
  - Previous restrictions on storing into the instruction stream in the access-register or home address-space-control modes are removed.
- In Chapter 6, “Interruptions”:
  - Changes introduced by the simulated IEEE exception (IXS) are added to the various sections.
  - Changes introduced by the compare-and-swap-and-store facility are added to various sections.
  - The descriptions of the special-operation and specification program interruptions are updated and rearranged.
  - Clarification is added to the section “Multiple Program-Interruption Conditions.”
- In Chapter 7, “General Instructions”:
  - The descriptions of the COMPARE AND SWAP AND STORE and EXTRACT CPU TIME instructions are added.
  - Programming notes for COMPARE LOGICAL LONG UNICODE and MOVE LONG UNICODE are amended to account for the long-displacement facility.
  - A programming note clarifying the setting of the condition code is added to AND IMMEDIATE, EXCLUSIVE OR IMMEDIATE, and OR IMMEDIATE.
- In Chapter 9, “Floating-Point Overview and Support Instructions”:
  - The concepts of “views” and “quantum” are introduced.
  - The name of the instruction SET ROUNDING MODE is changed to SET BFP ROUNDING MODE and is moved from Chapter 19 to Chapter 9.
  - The following instructions operating on the floating-point-control (FPC) register are moved from Chapter 19 to Chapter 9: EXTRACT FPC, LOAD FPC, SET FPC, and STORE FPC.
  - The FPR-GR-transfer facility instructions, LOAD FPR FROM GR and LOAD GR FROM FPR, are added.
  - The four instructions making up the floating-point-support-sign-handling facility are added: COPY SIGN, LOAD COMPLEMENT, LOAD NEGATIVE, and LOAD POSITIVE.

  - The decimal-floating-point-rounding facility is added. This includes a 3-bit DFP rounding mode field in the floating-point control (FPC) register and the instruction SET DFP ROUNDING MODE.
  - The IEEE-exception-simulation facility is added, including the instructions SET FPC AND SIGNAL and LOAD FPC AND SIGNAL.
  - The instruction PERFORM FLOATING-POINT OPERATION (PFPO) is added.
- In Chapter 10, "Control Instructions", changes introduced by the conditional-SSKE facility are added to the description of SET STORAGE KEY EXTENDED.
- In Chapter 11, "Machine-Check Handling", changes introduced by the conditional-SSKE facility are added.
- In Chapter 12, the operator rate control is now defined to be model dependent.
- In Chapter 14, clarification is added to the section "Modified CCW Indirect Data Addressing".
- In Chapter 18, "Hexadecimal-Floating-Point Instructions": The names of two of the rounding methods used by the instruction CONVERT TO FIXED are changed from "biased round to nearest" and "round to nearest" to "round to nearest with ties away from 0" and "round to nearest with ties to even", respectively.
- In Chapter 19, "Binary-Floating-Point Instructions": This chapter is updated extensively to be consistent with Chapter 20. This includes the following:
  - The terms "normal" and "subnormal", referring to BFP data classes, replace the obsolete terms "normalized", and "denormalized", respectively.
  - The name of the instruction SET ROUNDING MODE is changed to SET BFP ROUNDING MODE and is moved from Chapter 19 to Chapter 9.
  - The following instructions operating on the floating-point-control (FPC) register are moved from Chapter 19 to Chapter 9: EXTRACT FPC, LOAD FPC, SET FPC, and STORE FPC.
  - The section "Floating-Point-Control (FPC) Register" is moved to Chapter 9.
  - The sections "BFP Rounding", "BFP Comparison", "Condition Codes for BFP Instructions", and "IEEE Exception Conditions" are renamed and moved to Chapter 9.
- In Chapter 20, "Decimal-Floating-Point Instructions": This chapter is new, and describes the decimal-floating-point (DFP) facility.

The sixth edition also contains numerous minor corrections and clarifications.

Summary of Changes in Fifth Edition

The fifth edition of this publication differs from the previous edition principally by containing the definitions of the following facilities:

- DAT-enhancement facility 2
- ETF2-enhancement facility
- ETF3-enhancement facility
- Extended-immediate facility
- HFP-unnormalized-extensions facility
- Message-security-assist extension 1
- Modified-CCW-indirect-data-addressing facility
- PER-3 facility
- Server-time-protocol facility
- Store-clock-fast facility
- Store-facility-list-extended facility:
- TOD-clock-steering facility

The fifth edition contains minor clarifications and corrections and also the following significant changes relative to the previous edition:

- In Chapter 3, "Storage":
  - In the "Information Formats" section, the length of a storage-operand field that is implied by the instruction now includes a 16-byte operand.
  - Changes introduced by the modified-CCW-indirect-data-addressing facility are added to various sections.
  - Changes introduced by the DAT-enhancement facility 2 are added to various sections.

  - Changes introduced by the PER-3 facility are added to the section “Assigned Storage Locations.”
  - Changes introduced by the store-facility-list-extended facility are added to section “Assigned Storage Locations.”
- In Chapter 4, “Control”:
  - Changes introduced by the store-clock-fast facility are added to various sections.
  - Changes introduced by the PER-3 facility, including breaking-event-address recording and PER instruction-fetch nullification, are added to various sections.
  - Changes introduced by the server-time protocol (STP) facility are added to various sections.
  - Changes introduced by the TOD-clock-steering facility are added to various sections.
  - The conditional-emergency-signal and sense-running-status orders are added to the section “Signal-Processor Orders.”
  - The location in which the PSW that is preserved as a result of switching from the z/Architecture to the ESA/390 architectural mode is formally named the *Captured z/Architecture PSW register*.
  - The description of all facility-indication bits has been moved to a new section at the end of the chapter.
- In Chapter 5, “Program Execution”:
  - The instructions of the message-security assist are added to the section “Condition Code Alternative to Interruptibility.”
  - Changes introduced by the DAT-enhancement facility 2 are added to various sections.
  - Changes introduced by the store-clock-fast facility are added to various sections.
  - Changes introduced by the modified-CCW-indirect-data-addressing facility are added to the section “Channel-Program Serialization.”
- In Chapter 6, “Interruptions”:
  - Changes introduced by the PER-3 facility are added to the various sections.
  - Changes introduced by the server-time protocol (STP) facility are added to various sections.
  - Changes introduced by the TOD-clock-steering facility are added to various sections.
  - Changes introduced by the DAT-enhancement facility 2 are added to various sections.
  - Changes introduced by the extended-immediate facility are added to the description of the specification exception program interruption.
- In Chapter 7, “General Instructions”:
  - A note is added to the “Instructions” section indicating that, for certain new or modified instructions, an operand may be optional.
  - Descriptions of thirty-four new instructions introduced by the extended-immediate facility are added.
  - A description of the STORE CLOCK FAST instruction is added.
  - A description of the STORE FACILITY LIST EXTENDED instruction is added.
  - Changes introduced by the TOD-clock-steering facility are added to the STORE CLOCK, STORE CLOCK FAST, and STORE CLOCK EXTENDED instructions.
  - Descriptions of new functions introduced by the message-security-assist extension 1 are added to the CIPHER MESSAGE, CIPHER MESSAGE WITH CHAINING, COMPUTE INTERMEDIATE MESSAGE DIGEST, and COMPUTE LAST MESSAGE DIGEST instructions.
  - Changes introduced by the ETF-2 enhancement facility are added to the TRANSLATE ONE TO ONE, TRANSLATE ONE TO TWO, TRANSLATE TWO TO ONE, and TRANSLATE TWO TO TWO instructions.
  - Changes introduced by the ETF-3 enhancement are added to the CONVERT UTF-16 TO UTF-32, CONVERT UTF-16 TO UTF-8, CONVERT UTF-8 TO UTF-16, and CONVERT UTF-8 TO UTF-32 instructions.
- In Chapter 10, “Control Instructions”:

  - A description of the LOAD PAGE-TABLE-ENTRY ADDRESS instruction, introduced by the DAT-enhancement facility 2, is added.
  - A description of the PERFORM TIMING FACILITY FUNCTION instruction, introduced by the TOD-clock-steering facility, is added.
  - Changes introduced by the TOD-clock-steering facility are added to the description of the SET CLOCK instruction.
  - The description of all facility-indication bits are moved to Chapter 4 (they are removed from the description of the STORE FACILITY LIST instruction).
  - A model-dependent field is defined in the SYSIB 2.2.2 operand stored by the STORE SYSTEM INFORMATION instruction.
  - Changes introduced by the store-clock-fast facility are added to the description of the TRACE instruction.
- In Chapter 11, “Machine-Check Handling”:
  - Changes introduced by the store-clock-fast facility are added to the section “CPU Retry.”
  - Changes introduced by the modified-CCW-indirect-data-addressing facility are added to the section “Invalid CBC in Storage”.
  - Changes introduced by the TOD-clock-steering facility are added to the section “Timing-Facility Damage.”
  - Changes introduced by the server-time-protocol facility are added to the section “External-Damage Code.”
- In Chapter 13, “I/O Overview”, changes introduced by the modified-CCW-indirect-data-address facility are added to the section “Channel-Program Execution”.
- In Chapter 15, “Basic I/O Functions”:
  - Changes introduced by the modified-CCW-indirect-data-addressing facility are added which include:
    - The modified-indirect-data-address word (MIDAW) is defined.
    - Bit 25 of word 1 of the operation-request block (ORB) is defined as the modified-CCW-indirect-data-addressing control.
    - Bit 7 of the flags field in the channel-command word (CCW) is defined as the modified-indirect-data-addressing flag.
- In Chapter 16, “I/O Interruptions“:
  - Changes introduced by the modified-CCW-indirect-data-address facility are added to the sections “Subchannel-Status Word” and “Extended-Status Word”.
  - Changes are introduced to the section “Interface-Control Check” to support retrying a CCW-type IPL when an interface-control check is detected during the execution of the IPL channel program.
- In Chapter 17, “I/O Support Functions”:
  - Changes are introduced to the section “CCW-Type IPL” to support retrying a CCW-type IPL when an interface-control check is detected during the execution of the IPL channel program.
  - Changes are introduced to the section “List-Directed IPL” to support storing of the subsystem-identification word (SID) for the IPL-device during list-directed IPL, if a subchannel is associated with the IPL-device.
- In Chapter 18, “Hexadecimal-Floating-Point Instructions”, descriptions of the twelve instructions introduced by the HFP-unnormalized-extensions facility are added.
- In Appendix I, character representations of EBCDIC code pages 81C, 94C, 500, and 1047, IBM-PC, and BookMaster symbols are removed. Only the commonly used EBCDIC code page 037 and ISO-8 characters are shown.

Summary of Changes in Fourth Edition

The fourth edition of this publication differs from the previous edition principally by containing the definitions of the extended-translation facility 3 and the ASN-and-LX-reuse facility. The fourth edition contains minor clarifications and corrections and also the following significant changes relative to the previous edition:

- In Chapter 3, “Storage”:

  - Changes introduced by the ASN-and-LX-reuse facility are added, including changes to the “Address Spaces” and “ASN Translation” sections.
  - A programming note is added to the “Dynamic Address Translation” section, describing implications in using common segments.
  - The ATMID and AI fields are corrected in the “Assigned Storage Locations” figure.
- In Chapter 4, “Control”:
  - Changes introduced by the ASN-and-LX-reuse facility are added, including changes to the “Trace” section.
  - The list-directed IPL function is added to the “Initial Program Load” section.
  - In the “Trace” section, serialization requirements for instructions that implicitly store into the trace table or linkage stack are relaxed.
- In Chapter 5, “Program Execution”:
  - A new RI and SS instruction format are included.
  - Changes introduced by the ASN-and-LX-reuse facility are added, including changes to the “Authorization Mechanisms”, “PC-Number Translation”, and “Linkage-Stack Operations” sections.
  - The instructions of the extended-translation facility 3 (except TRANSLATE AND TEST REVERSED) are added to the sections “Condition-Code Alternative to Interruptibility” and “Multiple-Access References.”
  - Serialization requirements for instructions that implicitly store into the trace table or linkage stack are relaxed.
- In Chapter 6, “Interruptions”:
  - Changes introduced by the ASN-and-LX-reuse facility include the new LFX translation, LSX translation, LSTE sequence, and ASTE instance exceptions.
  - An additional condition for TRAP is added to the list of instructions that can cause a special-operation exception to be recognized.
  - SEARCH STRING UNICODE, a part of the extended-translation facility 3, is added to the list of instructions that can cause a specification exception to be recognized.
- In Chapter 7, “General Instructions”:
  - Six new instructions provided by the extended-translation facility 3 are added. The instructions CONVERT UNICODE TO UTF-8 (CUUTF) and CONVERT UTF-8 TO UNICODE (CUTFU) are renamed to CONVERT UTF-16 TO UTF-8 (CU21) and CONVERT UTF-8 TO UTF-16 (CU12), respectively. The old mnemonics continue to be recognized.
  - The instruction-format illustrations for STAMY and STMY are corrected.
- In Chapter 10, “Control Instructions”:
  - Changes introduced by the ASN-and-LX-reuse facility include the following:
    - Four new instructions provided by the facility are added.
    - The definitions of BRANCH AND STACK, BRANCH IN SUBSPACE GROUP, EXTRACT PRIMARY ASN, EXTRACT SECONDARY ASN, LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, and SET SECONDARY ASN are updated to account for the facility.
  - New facility bits for the ASN-and-LX-reuse facility and the extended-translation facility 3 are added to STORE FACILITY LIST.
  - New fields are added to the system-information block (SYSIB) returned by STORE SYSTEM INFORMATION.
  - Serialization requirements for instructions that implicitly store into the trace table or linkage stack are relaxed, including PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER (WITH INSTANCE), SET SECONDARY ASN (WITH INSTANCE), and TRACE.
  - Corrections and clarifications are made to TRAP.
- In Chapter 12, “Operator Facilities”: Changes introduced by the list-directed IPL function are

added, including the load-clear-list-directed key and the load-with-dump key.

- In Chapter 14, “I/O Instructions”, Clarifications are added to programming note 1 of the CANCEL SUBCHANNEL instruction description.
- In Chapter 16, “I/O Interruptions”:
  - Bit 1 of the extended-report word (ERW) is defined as the request-logging-only (L) bit.
  - Bit 2 of the ERW is defined as the extended-subchannel-logout-pending (E) bit.
  - Words 2-3 of the format-0 extended-status word (ESW) are defined as the failing-storage address when the failing-storage-address-validity (F) bit, bit 6 of the ERW, is one and as the extended-subchannel-logout descriptor (ESLD) when the extended-subchannel-logout-pending (E) bit, bit 2 of the ERW, is one. The E-bit is always zero when the F-bit is one, and the F-bit is always zero when the E-bit is one.
- In Chapter 17, “I/O Support Functions”, list-directed IPL is added to support initial-program loading (IPL) from devices that are not accessed by CCWs. IPL from devices that are accessed by CCWs is designated CCW-type IPL.
- In Appendix C, “Condition-Code Settings”: The instructions of the extended-translation facility 3 are added.

Summary of Changes in Third Edition

The third edition of this publication differs from the previous edition principally by containing the definitions of the DAT-enhancement, HFP-multiply-add/subtract, and long-displacement facilities and the message-security assist. The third edition contains minor clarifications and corrections and also the following significant changes relative to the previous edition:

- In Chapter 3, “Storage”:
  - Clarifications are added to the description of dynamic-address-translation process.
  - The primary address-space-control element (ASCE) in control register 1 is an attaching ASCE even when the CPU is in the home-space mode, and the home ASCE in control register 13 is an attaching ASCE even when the CPU is in the secondary-space mode.
  - The illustration of the PER code in Figure 3-19 is corrected.
- In Chapter 4, “Control”:
  - The relationships between ETR time (TOD-clock time), UTC, and International Atomic Time are described in programming note 3 on page 4-51.
  - Code 0 of the SIGNAL PROCESSOR set-architecture order, and also a CPU reset due to activation of the load normal key, are changed to save the current z/Architecture PSW when switching to the ESA/390 architectural mode. Also, code 2 of the order is added, and this restores, for CPUs other than the one executing SIGNAL PROCESSOR, the saved PSW when switching to the z/Architecture architectural mode, provided that the saved PSW has not been set to all zeros by certain resets.
- In Chapter 5, “Program Execution”:
  - The I format, the RI format with a M_1 operand, and the SS format with the I_3 operand are added.
  - The RSY, RXY, and SIY instruction formats are added, and the RSE format is deleted. (All instructions that were of format RSE are now referred to as being of format RSY.)
  - The formation of an operand address using the 20-bit signed displacement of instructions of formats RSY, RXY, and SIY is described.
  - The results when a PER instruction-fetching event occurs along with certain exceptions or exception conditions are clarified. See “Indication of PER Events Concurrently with Other Interruption Conditions” on page 4-39
  - The fetch of the address-space-control element from the ASN-second-table entry during access-register translation is double-word concurrent as observed by other CPUs.

  - The change bit is not necessarily set to one currently with the related storage reference, as observed by other CPUs; it may be set to one before or after the reference, within certain limits. See “Storage-Key Accesses” on page 5-120 for a detailed description of when the change bit is set.
  - The five instructions of the message-security assist are added to the list of instructions having multiple-access references.

- In Chapter 6, “Interruptions,” the list of conditions causing a specification exception to be recognized is extended to include those caused by the message-security assist instructions.
- In Chapter 7, “General Instructions”:
  - Thirty-nine instructions provided by the long-displacement facility are added. With the exception of the new LOAD BYTE instruction, the instructions added by the long-displacement facility have names and functions that are the same as existing instructions (but the mnemonics and opcodes are new). The new instructions are of formats RSY, RXY, and SIY and have a 20-bit signed displacement instead of a 12-bit unsigned displacement.
  - All previously existing format-RSE and format-RXE instructions are changed to be of formats RSY and RXY, respectively, by use of a previously unused byte in the instructions. These changes are not marked by a bar in the margin.
  - Five instructions provided by the message-security assist are added.
  - The instruction format of SUPERVISOR CALL is changed to I.
- In Chapter 9, “Floating-Point Overview and Support Instructions,” four instructions provided by the long-displacement facility are added. These are the LOAD (long and short) and STORE (long and short) instructions.
- In Chapter 10, “Control Instructions”:
  - The COMPARE AND SWAP AND PURGE (CSPG) and INVALIDATE DAT TABLE ENTRY instructions provided by the DAT-enhancement facility are added. CSPG operates on a doubleword operand in storage.
  - The definition of LOAD ADDRESS SPACE PARAMETERS is clarified.
  - The LOAD REAL ADDRESS (LRAY) instruction provided by the long-displacement facility is added.
  - All previously existing format-RSE instructions are changed to be of format RSY by use of a previously unused byte in the instructions. These changes are not marked by a bar in the margin.
  - The description of the bits set by STORE FACILITY LIST is clarified, and new bits are assigned.
- In Chapter 14, “I/O Instructions”:
  - The definition of MODIFY SUBCHANNEL is modified.
  - The definition of SET CHANNEL MONITOR is modified.
- In Chapter 15, “Basic I/O Functions,” the following changes are made to the subchannel-information-block (SCHIB):
  - Bit 29 of word 6 of the path-management-control word (PMCW) is defined as the measurement-block-format control.
  - Bit 30 of word 6 of the PMCW is defined as the extended-measurement-word-mode enablement bit.
  - The definition of words 10-11 (words 0-1 of the model-dependent area) are changed to contain a measurement-block address, when the extended-I/O-measurement-block facility is installed.
- In Chapter 16, “I/O Interruptions”, the interruption-response block (IRB) is extended to include the extended-measurement word.
- In Chapter 17, “I/O Support Functions”:
  - The requirement that the measurement block be updated when secondary status is accepted is clarified.
  - The extended-measurement-block facility is added.
  - The extended-measurement-word facility is added.

- In Chapter 18, “Hexadecimal-Floating-Point Instructions,” the MULTIPLY AND ADD (four instructions) and MULTIPLY AND SUBTRACT (four instructions) instructions provided by the HFP-multiply-add/subtract facility are added.

The above changes may affect other chapters besides the ones listed. All technical changes to the text or to an illustration are indicated by a vertical line to the left of the change.

Summary of Changes in Second Edition

The second edition of this publication differs from the previous edition mainly by containing clarifications and corrections. The significant changes are as follows:

- In Chapter 1, “Introduction”:
  - Summaries of DIVIDE LOGICAL and MULTIPLY LOGICAL, TEST ADDRESSING MODE, the set-architecture order of SIGNAL PROCESSOR, and STORE FACILITY LIST are added or improved.
  - An extensive summary of the input/output enhancements placed in z/Architecture is added.
- In Chapter 3, “Storage”:
  - Definitions of absolute locations 0-23 are deleted since they pertain only to an ESA/390 initial program load.
  - The definition of real locations 200-203, stored in by STORE FACILITY LIST, is corrected to state that bit 16 indicates the extended-translation facility 2.
- In Chapter 4, “Control,” a description of unassigned fields in the PSW is corrected to state that bit 4 is unassigned and bit 31 is assigned.
- In Chapter 5, “Program Execution,” the RSL format and an RIL format with an M_1 field are added.
- In Chapter 7, “General Instructions”:
  - The definition of BRANCH AND SET MODE is corrected to state that bit 63 of the R_1 general register remains unchanged in the 24-bit or 31-bit addressing mode; the bit is not set to zero.
  - The definitions of PACK ASCII, PACK UNICODE, UNPACK ASCII, and UNPACK UNICODE are clarified.
  - It is clarified that the following instructions perform multiple-access references to their storage operands:
    - CHECKSUM
    - COMPARE AND FORM CODEWORD
    - CONVERT UNICODE TO UTF-8
    - CONVERT UTF-8 TO UNICODE
  - It is clarified that the following instructions do not necessarily process their storage operands left to right as observed by other CPUs: MOVE LONG, MOVE LONG EXTENDED, and MOVE LONG UNICODE. Special padding characters of MOVE LONG and MOVE LONG EXTENDED specify whether left-to-right processing should be performed, as observed by other CPUs, and whether the data being moved should or should not be placed in the cache for availability for subsequent processing.
- In Chapter 10, “Control Instructions,” it is clarified that the following instructions perform multiple-access references to their storage operands:
  - LOAD ADDRESS SPACE PARAMETERS
  - RESUME PROGRAM
  - STORE SYSTEM INFORMATION

Chapters 13-17 contain many clarifying changes, all indicated by a vertical line in the margin, in addition to the significant changes listed below.

- In Chapter 13, “I/O Overview,” statements about the suspend flag in a CCW are clarified to describe the flag being specified as a one and being valid because of a one value of the suspend control in the associated ORB.
- In Chapter 14, “I/O Instructions,” the results of MODIFY SUBCHANNEL when the device-number-valid bit at the designated subchannel is zero are corrected.
- In Chapter 15, “Basic I/O Functions”:
  - It is clarified that unlimited prefetching of data and IDAWs associated with the current

and prefetched CCWs is allowed independent of the value of the prefetch control in the associated ORB.

  - A specified control-unit-priority number is ignored if the channel-subsystem-I/O-priority facility is not operational due to an operator action.

  - It is clarified that address-limit checking applies to data locations and not to locations containing a CCW or IDAW.

- In Chapter 16, “I/O Interruptions,” the form of the address stored in the failing-storage-address field is described in terms of the format-2-IDAW control instead of an addressing mode.

- In Chapter 17, “I/O Support Functions”:

  - The introduction to the channel-subsystem monitoring facilities is clarified.

  - References to the measurement block by the measurement-block-update facility are single-access references and appear to be word concurrent as observed by CPUs. They do not appear to be block concurrent.

  - The description of the channel-subsystem-I/O-priority facility is corrected by including mention of control-unit priority for fibre-channel-attached control units.

The above changes may affect other chapters besides the ones listed.

Chapter 1. Introduction

Highlights of Original z/Architecture 1-2
General Instructions for 64-Bit Integers 1-3
Other New General Instructions 1-3
Floating-Point Instructions 1-4
Control Instructions 1-5
Trimodal Addressing 1-5
Modal Instructions 1-5
Effects on Bits 0-31 of a General Register . 1-6
Input/Output 1-6
Additions to z/Architecture 1-7
ASN-and-LX-Reuse Facility 1-7
CMPSC-Enhancement Facility 1-7
Compare-and-Swap-and-Store Facility 1-7
Compare-and-Swap-and-Store Facility 2 1-8
Conditional-SSKE Facility 1-8
Configuration-Topology Facility 1-8
Configuration-z/Architecture-Architectural -Mode Facility 1-8
Constrained-Transactional-Execution Facility . 1-8
DAT-Enhancement Facility 1 1-8
DAT-Enhancement Facility 2 1-8
Decimal-Floating-Point Facility 1-9
Decimal-Floating-Point Packed-Conversion Facility 1-9
Decimal-Floating-Point-Rounding Facility 1-9
Decimal-Floating-Point Zoned-Conversion Facility 1-9
Distinct-Operands Facility 1-9
Enhanced-DAT Facility 1 1-9
Enhanced-DAT Facility 2 1-10
Enhanced-Monitor Facility 1-11
Entropy Encoding Compression Facility 1-11
ESA/390-Compatibility-Mode Facility 1-11
ETF2-Enhancement Facility 1-11
ETF3-Enhancement Facility 1-11
Execute-Extensions Facility 1-11
Execution-Hint Facility 1-11
Extended-Immediate Facility 1-12
Extended-I/O-Measurement-Block Facility . . 1-12
Extended-I/O-Measurement-Word Facility . . 1-12
Extended-Translation Facility 2 1-12
Extended-Translation Facility 3 1-13
Extract-CPU-Time Facility 1-13
Fast-BCR-Serialization Facility 1-13
Fibre-Channel Extensions (FCX) 1-13
FCX-Bidirectional-Data-Transfer Facility 1-14
Floating-Point Extension Facility 1-14
Floating-Point-Support-Sign- Handling Facility 1-15
FPR-GR-Transfer Facility 1-15
General-Instructions-Extension Facility 1-15
Guarded-Storage Facility 1-15
HFP Multiply-and-Add/Subtract Facility 1-16
HFP-Unnormalized-Extensions Facility 1-16
High-Word Facility 1-16
IEEE-Exception-Simulation Facility 1-16
Insert-Reference-Bits-Multiple Facility 1-16
Instruction-Execution-Protection Facility 1-16
Interlocked-Access Facility 1 1-17
Interlocked-Access Facility 2 1-17
IPTE-Range Facility 1-17
List-Directed Initial Program Load 1-17
Load-and-Trap Facility 1-17
Load-and-Zero-Rightmost-Byte Facility 1-17
Load/Store-on-Condition Facility 1 1-18
Load/Store-on-Condition Facility 2 1-18
Local-TLB-Clearing Facility 1-18
Long-Displacement Facility 1-18
Message-Security Assist 1-18
Message-Security-Assist Extension 1 1-19
Message-Security-Assist Extension 2 1-19
Message-Security-Assist Extension 3 1-19
Message-Security-Assist Extension 4 1-20
Message-Security-Assist Extension 5 1-21
Message-Security-Assist Extension 6 1-21
Message-Security-Assist Extension 7 1-21
Message-Security-Assist Extension 8 1-22
Miscellaneous-Instruction-Extensions Facility 1 1-22
Miscellaneous-Instruction-Extensions Facility 2 1-22
Modified CCW Indirect Data Addressing Facility 1-22
Move-With-Optional-Specifications Facility . . . 1-23
Multiple-Epoch Facility 1-23
Multiple-Subchannel-Set Facility 1-23
Multithreading Facility 1-23
Nonquiescing Key-Setting Facility 1-23
Parsing-Enhancement Facility 1-24
PER-3 Facility 1-24
PER Zero-Address-Detection Facility 1-24
PFPO Facility 1-24
Population-Count Facility 1-24
Processor-Assist Facility 1-24
Reset-Reference-Bits-Multiple Facility 1-25
Restore-Subchannel Facility 1-25
Server-Time-Protocol Facility 1-25
Side-Effect-Access Facility 1-25
Store-Clock-Fast Facility 1-25
Store-Facility-List-Extended Facility 1-25
Test-Pending-External-Interruption Facility . . . 1-25

© Copyright IBM Corp. 2000, 2017

TOD-Clock-Steering Facility. 1-26
Transactional-Execution Facility 1-26
Vector-Enhancements Facility 1 1-26
Vector Facility for z/Architecture 1-26
Vector Packed-Decimal Facility 1-26
Warning-Track Interruption Facility 1-27
The ESA/390 Base 1-27
The ESA/370 and 370-XA Base. 1-32
System Program 1-34
Compatibility 1-34
Compatibility among z/Architecture Systems .1-34
Compatibility between z/Architecture and ESA/390 1-34
Control-Program Compatibility 1-34
Problem-State Compatibility 1-35
Availability 1-35

This publication provides, for reference purposes, a detailed description of z/Architecture.™

The architecture of a system defines its attributes as seen by the programmer, that is, the conceptual structure and functional behavior of the machine, as distinct from the organization of the data flow, the logical design, the physical design, and the performance of any particular implementation. Several dissimilar machine implementations may conform to a single architecture. When the execution of a set of programs on different machine implementations produces the results that are defined by a single architecture, the implementations are considered to be compatible for those programs.

Highlights of Original z/Architecture

z/Architecture is the next step in the evolution from the System/360 to the System/370™, System/370 extended architecture (370-XA), Enterprise Systems Architecture/370™ (ESA/370™), and Enterprise Systems Architecture/390® (ESA/390™). z/Architecture includes all of the facilities of ESA/390 except for the asynchronous-pageout, asynchronous-data-mover, program-call-fast, and ESA/390 vector facilities. z/Architecture also provides significant extensions, as follows:

- Sixty-four-bit general registers and control registers.
- A 64-bit addressing mode, in addition to the 24-bit and 31-bit addressing modes of ESA/390, which are carried forward to z/Architecture.

  Both operand addresses and instruction addresses can be 64-bit addresses. The program-status word (PSW) is expanded to 16 bytes to contain the larger instruction address. The PSW also contains a newly assigned bit that specifies the 64-bit addressing mode.
- Up to three additional levels of dynamic-address-translation (DAT) tables, called region tables, for translating 64-bit virtual addresses.

  A virtual address space may be specified either by a segment-table designation as in ESA/390 or by a region-table designation, and either of these types of designation is called an address-space-control element (ASCE). An ASCE may alternatively be a real-space designation that causes virtual addresses to be treated simply as real addresses without the use of DAT tables.
- An 8 K-byte prefix area for containing larger old and new PSWs and register save areas.
- A SIGNAL PROCESSOR order for switching between the ESA/390 and z/Architecture architectural modes.

  Initial program loading sets the ESA/390 architectural mode. The new SIGNAL PROCESSOR order then can be used to set the z/Architecture mode or to return from z/Architecture to ESA/390. This order causes all CPUs in the configuration always to be in the same architectural mode.
- Many new instructions, many of which operate on 64-bit binary integers

Some of the new instructions that do not operate on 64-bit binary integers have also been added to ESA/390.

All of the ESA/390 instructions, except for those of the four facilities named above, are included in z/Architecture.

The bit positions of the general registers and control registers of z/Architecture are numbered 0-63. An ESA/390 instruction that operates on bit positions

0-31 of a 32-bit register in ESA/390 operates instead on bit positions 32-63 of a 64-bit register in z/Architecture.

z/Architecture was announced in October, 2000. The remainder of this section summarizes the original contents of z/Architecture. Subsequent additions are described in “Additions to z/Architecture” on page 1-7.

General Instructions for 64-Bit Integers

The 32-bit-binary-integer instructions of ESA/390 have new analogs in z/Architecture that operate on 64-bit binary integers. There are two types of analogs:

- Analogs that use two 64-bit binary integers to produce a 64-bit binary integer. For example, the ESA/390 ADD instruction (A for a storage-to-register operation or AR for a register-to-register operation) has the analogs AG (adds 64 bits from storage to the contents of a 64-bit general register) and AGR (adds the contents of a 64-bit general register to the contents of another 64-bit general register). These analogs are distinguished by having “G” in their mnemonics.
- Analogs that use a 64-bit binary integer and a 32-bit binary integer to produce a 64-bit binary integer. The 32-bit integer is either sign-extended or extended on the left with zeros, depending on whether the operation is signed or unsigned, respectively. For example, the ESA/390 ADD (A or AR) instruction has the analogs AGF (adds 32 bits from storage to the contents of a 64-bit general register) and AGFR (adds the contents of bit positions 32-63 of a 64-bit general register to the contents of another 64-bit general register). These analogs are distinguished by having “GF” in their mnemonics.

Other New General Instructions

The other additional or significantly enhanced general instructions of z/Architecture are highlighted as follows:

- ADD LOGICAL WITH CARRY and SUBTRACT LOGICAL WITH BORROW operate on either 32-bit or 64-bit unsigned binary integers and include a carry or borrow, as represented by the leftmost bit of the two-bit condition code in the PSW, in the computation. This can improve the performance of operating on extended-precision integers (integers longer than 64 bits).
- AND IMMEDIATE and OR IMMEDIATE combine a two-byte immediate operand with any of the two bytes on two-byte boundaries in a 64-bit general register.
- BRANCH AND SAVE AND SET MODE and BRANCH AND SET MODE are enhanced so that they set bit 63 of the R_1 general register to one if the current addressing mode is the 64-bit mode, and they set the 64-bit addressing mode if bit 63 of the R_2 general register is one. This allows “pointer-directed” linkages between programs in different addressing modes, including any of the 24-bit, 31-bit, and 64-bit modes.
- BRANCH RELATIVE AND SAVE LONG and BRANCH RELATIVE ON CONDITION LONG are like the BRANCH RELATIVE AND SAVE and BRANCH RELATIVE ON CONDITION instructions of ESA/390 except that the new instructions use a 32-bit immediate field. This increases the target range available through relative branching.
- COMPARE AND FORM CODEWORD is enhanced so that, in the 64-bit addressing mode, the comparison unit is six bytes instead of two and the resulting codeword is eight bytes instead of four. UPDATE TREE is enhanced so that, in the 64-bit addressing mode, a node is 16 bytes instead of eight and the codeword in a node is eight bytes instead of four. This improves the performance of sorting records having long keys.
- DIVIDE LOGICAL uses a 64-bit or 128-bit unsigned binary dividend and a 32-bit or 64-bit unsigned binary divisor, respectively, to produce a 32-bit or 64-bit quotient and remainder, respectively. MULTIPLY LOGICAL uses a 32-bit or 64-bit unsigned binary multiplicand and multiplier to produce a 64-bit or 128-bit product, respectively.
- DIVIDE SINGLE divides a 64-bit dividend by a 32-bit or 64-bit divisor and produces a 64-bit quotient and remainder. MULTIPLY SINGLE is enhanced so it can multiply a 64-bit multiplicand by a 32-bit or 64-bit multiplier and produce a 64-bit product.

- EXTRACT PSW extracts bits 0-63 of the current PSW to allow determination of the current machine state, for example, determination of whether the CPU is in the problem state or the supervisor state.
- INSERT IMMEDIATE inserts a two-byte immediate operand into a 64-bit general register on any of the two-byte boundaries in the register. LOAD LOGICAL IMMEDIATE does the same and also clears the remainder of the register.
- LOAD ADDRESS RELATIVE LONG forms an address relative to the current (unupdated) instruction address by means of a signed 32-bit immediate field.
- LOAD LOGICAL THIRTY ONE BITS places the rightmost 31 bits of either a general register or a word in storage, with 33 zeros appended on the left, in a general register.
- LOAD MULTIPLE DISJOINT loads the leftmost 32 bits of each register in a range of general registers from one area in storage and the rightmost 32 bits of each of those registers from another area in storage. This is for use in place of a LOAD MULTIPLE HIGH instruction and a 32-bit LOAD MULTIPLE instruction when one of the storage areas is addressed by one of the registers loaded.
- LOAD MULTIPLE HIGH and STORE MULTIPLE HIGH load or store the leftmost 32 bits of each register in a range of general registers, allowing augmentation of existing programs that load or store the rightmost 32 bits by means of LOAD MULTIPLE and STORE MULTIPLE. (Sixty-four-bit forms of LOAD MULTIPLE and STORE MULTIPLE also are provided.)
- LOAD PAIR FROM QUADWORD and STORE PAIR TO QUADWORD operate between an even-odd pair of 64-bit general registers and a quadword in storage (16 bytes aligned on a 16-byte boundary). These instructions provide quadword consistency (all bytes appear to be loaded or stored concurrently in a multiple-CPU system). (Only the 64-bit form of COMPARE DOUBLE AND SWAP also provides quadword consistency.)
- LOAD REVERSED and STORE REVERSED load or store a two-byte, four-byte, or eight-byte unit in storage with the left-to-right sequence of the bytes reversed. LOAD REVERSED also can move a four-byte or eight-byte unit between two general registers. These operations allow conversion between “little-endian” and “big-endian” formats.
- PERFORM LOCKED OPERATION is enhanced with two more sets of function codes, with each set providing six different operations. One of the additional sets provides operations on 64-bit operands in 64-bit general registers, and the other provides operations on 128-bit operands in a parameter list.
- ROTATE LEFT SINGLE LOGICAL obtains 32 bits or 64 bits from a general register, rotates them (the leftmost bit replaces the rightmost bit), and places the result in another general register (a nondestructive rotate).
- SET ADDRESSING MODE can set any of the 24-bit, 31-bit, and 64-bit addressing modes.
- SHIFT LEFT SINGLE, SHIFT LEFT SINGLE LOGICAL, SHIFT RIGHT SINGLE, and SHIFT RIGHT SINGLE LOGICAL are enhanced with 64-bit forms that obtain the source operand from one general register and place the result operand in another general register (a nondestructive shift).
- TEST ADDRESSING MODE sets the condition code to indicate whether bits 31 and 32 of the current PSW specify the 24-bit, 31-bit, or 64-bit addressing mode.
- TEST UNDER MASK HIGH and TEST UNDER MASK LOW, which are ESA/390 instructions, are given the alternative name TEST UNDER MASK, and two additional forms are added so that a two-byte immediate operand can be used to test the bits of two bytes located on any of the two-byte boundaries in a 64-bit general register. (The ESA/390 instruction TEST UNDER MASK, which uses a one-byte immediate operand to test a byte in storage, continues to be provided.)

Floating-Point Instructions

The z/Architecture floating-point instructions are the same as in ESA/390 except that instructions are added for converting between 64-bit signed binary integers and either hexadecimal or binary floating-point data. These new instructions have “G” in their mnemonics.

Control Instructions

The new or enhanced control instructions of z/Architecture are highlighted as follows:

- EXTRACT AND SET EXTENDED AUTHORITY is a privileged instruction for changing the extended authorization index in a control register. This enables real-space designations to be used more efficiently by means of access lists.
- EXTRACT STACKED REGISTERS is enhanced to extract optionally all 64 bits of the contents of one or more saved general registers.
- EXTRACT STACKED STATE is enhanced to extract optionally the entire contents of the saved PSW, including a 64-bit instruction address.
- LOAD CONTROL and STORE CONTROL are enhanced for operating optionally on 64-bit control registers.
- LOAD PSW uses an eight-byte storage operand as in ESA/390 and expands this operand to a 16-byte z/Architecture PSW.
- LOAD PSW EXTENDED directly loads a 16-byte PSW.
- LOAD REAL ADDRESS in its ESA/390 form and in the 24-bit or 31-bit addressing mode operates as in ESA/390 if the translation is successful and the obtained real address has a value less than 2G bytes. LOAD REAL ADDRESS in its ESA/390 form and in the 64-bit addressing mode, or in its enhanced z/Architecture form in any addressing mode, loads a 64-bit real address.
- LOAD USING REAL ADDRESS and STORE USING REAL ADDRESS are enhanced to have optionally 64-bit operands.
- SIGNAL PROCESSOR has a new order that can be used to switch all CPUs in the configuration either from the ESA/390 architectural mode to the z/Architecture architectural mode or from z/Architecture to ESA/390. (A system that is to operate using z/Architecture must first be IPLed in the ESA/390 mode.)
- STORE FACILITY LIST is a privileged instruction that stores at real location 200 an indication of whether z/Architecture is installed and of whether it is active. This instruction is added also to ESA/390 and also stores an indication of whether the new z/Architecture instructions that have been added to ESA/390 are available. Real location 200 has previously contained all zeros in most systems and normally can be examined by a problem-state program whether or not STORE FACILITY LIST is installed. The information stored at real location 200 also indicates whether the extended-translation facility 2 is installed.
- STORE REAL ADDRESS is like LOAD REAL ADDRESS except that STORE REAL ADDRESS stores the resulting address instead of placing it in a register.
- TRACE is enhanced to record optionally the contents of 64-bit general registers.

Trimodal Addressing

"Trimodal addressing" refers to the ability to switch between the 24-bit, 31-bit, and 64-bit addressing modes. This switching can be done by means of:

- The old instructions BRANCH AND SAVE AND SET MODE and BRANCH AND SET MODE. Both of these instructions set the 64-bit addressing mode if bit 63 of the R_2 general register is one. If bit 63 is zero, the instructions set the 24-bit or 31-bit addressing mode if bit 32 of the register is zero or one, respectively.
- The new instruction SET ADDRESSING MODE (SAM24, SAM31, and SAM64). The instruction sets the 24-bit, 31-bit, or 64-bit addressing mode as determined by the operation code.

Modal Instructions

Trimodal addressing affects the general instructions only in the manner in which logical storage addresses are handled, except as follows.

- The instructions BRANCH AND LINK, BRANCH AND SAVE, BRANCH AND SAVE AND SET MODE, BRANCH AND SET MODE, and BRANCH RELATIVE AND SAVE place information in bit positions 32-39 of general register R_1 as in ESA/390 in the 24-bit or 31-bit addressing mode or place address bits in those bit positions in the 64-bit addressing mode. The new instruction BRANCH RELATIVE AND SAVE LONG does the same.
- The instructions BRANCH AND SAVE AND SET MODE and BRANCH AND SET MODE place a

one in bit position 63 of general register R_1 in the 64-bit addressing mode. In the 24-bit or 31-bit mode, BRANCH AND SAVE AND SET MODE sets bit 63 to zero, and BRANCH AND SET MODE leaves it unchanged.

- Certain instructions leave bits 0-31 of a general register unchanged in the 24-bit or 31-bit addressing mode but place or update address or length information in them in the 64-bit addressing mode. These are listed in programming note 1 on page 7-8 and are sometimes called modal instructions.

Effects on Bits 0-31 of a General Register

Bits 0-31 of general registers are changed by two types of instructions. The first type is a modal instruction (see the preceding section) when the instruction is executed in the 64-bit addressing mode. The second type is an instruction having, independent of the addressing mode, either a 64-bit result operand in a single general register or a 128-bit result operand in an even-odd general-register pair.

Most of the instructions of the second type are indicated by a "G," either alone or in "GF," in their mnemonics. The other instructions that change or may change bits 0-31 of a general register regardless of the current addressing mode are listed in programming note 2 on page 7-8. All of the instructions of the second type are sometimes referred to as "G-type" instructions.

If a program is not executed in the 64-bit addressing mode and does not contain a G-type instruction, it cannot change bits 0-31 of any general register.

Input/Output

Additional I/O functions and facilities are provided when z/Architecture is installed. They are provided in both the ESA/390 and the z/Architecture architectural mode and are as follows:

- Indirect data addressing is enhanced by the provision of a doubleword format-2 IDAW that is intended to allow operations on data at or above the 2 G-byte absolute-address boundary in z/Architecture. The previously existing IDAW, a word containing a 31-bit address, is now called a format-1 IDAW. The format-2 IDAW contains a 64-bit address. A bit in the operation-request block (ORB) associated with a channel program specifies whether the program uses format-1 or format-2 IDAWs. A further enhancement is the ability of all format-2 IDAWs of a channel program to specify either 2 K-byte or 4 K-byte data blocks, as determined by another bit in the ORB. The use of 4 K-byte blocks improves the efficiency of data transfers.
- The *FICON®-channel facility* provides the capabilities of attaching FICON-I/O-interface and FICON-converted-I/O-interface channel paths and of fully utilizing these channel-path types. FICON channel paths can significantly enhance overall data throughput by providing increased data-transfer rates in comparison to ESCON channel paths and by allowing multiple commands and associated data to be "streamed" to control units, thus further improving performance. The facility supports the following additional control mechanisms:
  - The modification-control bit in the ORB allows the program to optimize the performance of FICON channel paths when dynamically modifying channel programs.
  - The synchronization-control bit in the ORB ensures data integrity along with maximum channel-path performance by delaying the execution of a write command until the completion of an immediately preceding read command when performing unlimited prefetching of CCWs and when the data to be written may be the data read.
  - The streaming-mode-control bit in the ORB allows the program to prevent command streaming in cases that require such prevention.
  - The secondary-CCW-address field in the extended-status word assists in the recovery of channel programs that terminate abnormally when command streaming to a control unit is being performed. The field identifies a CCW that failed at the control unit.
- The *ORB-extension facility* expands the size of the ORB from three words to eight words. This makes fields available for use by the channel-subsystem-I/O-priority facility.
- The *channel-subsystem-I/O-priority facility* allows the program to establish a priority relationship among subchannels that have pending I/O operations. The priority relationship specifies the

order in which I/O operations are initiated by the channel subsystem. Additionally, for fibre-channel-attached control units, the facility allows the program to specify the priority in which I/O operations pending at the control unit are performed.

The input/output enhancements are further highlighted below by describing how they affect the I/O chapters.

- In Chapter 13, “I/O Overview,” FICON and FICON-converted I/O interfaces and the frame-multiplex mode are introduced.
- In Chapter 14, “I/O Instructions”:
  - The CANCEL SUBCHANNEL instruction is described.
  - TEST PENDING INTERRUPTION, when the second-operand address is zero, stores a three-word I/O-interruption code at real locations 184-195. The new third word contains an interruption-identification word that further identifies the source of the I/O interruption.
- In Chapter 15, “Basic I/O Functions”:
  - The ORB is extended to eight words and newly contains a streaming-mode control, modification control, synchronization control, format-2-IDAW control, 2K-IDAW control, ORB-extension control, channel-subsystem priority, and control-unit priority.
  - A doubleword format-2 IDAW and 4 K-byte data blocks optionally designated by format-2 IDAWs are added.
- In Chapter 16, “I/O Interruptions”:
  - A secondary-CCW-address-validity bit and failing-storage-address-format bit are added to the extended-report word.
  - A two-word failing-storage address and a secondary-CCW address are added to the format-0 extended-status word.
- In Chapter 17, “I/O Support Functions”:
  - Control-unit-defer time is added. This has an effect on the device-connect time and device-disconnect time in the measurement block.
  - References to the measurement block by the measurement-block-update facility are single-access references and appear to be word concurrent as observed by CPUs.
  - Device-active-only time is added to the measurement block.
  - The channel-subsystem-I/O-priority facility, providing channel-subsystem priority and control-unit priority, is added.

Additions to z/Architecture

z/Architecture was announced in October, 2000. Any extension added subsequently is summarized below and has the date of its announcement at the end of its summary.

ASN-and-LX-Reuse Facility

The *ASN-and-LX-reuse facility* may be available on a model implementing z/Architecture. The facility provides the means by which an address-space number (ASN) that is used in certain space-switching linkage instructions may be safely reused. The facility also adds a 32-bit program-call (PC) number, and it also provides the means by which the linkage index that is used in PC-number translation may be safely reused. The facility provides the following instructions:

- EXTRACT PRIMARY ASN AND INSTANCE
- EXTRACT SECONDARY ASN AND INSTANCE
- PROGRAM TRANSFER WITH INSTANCE
- SET SECONDARY ASN WITH INSTANCE

(May, 2004)

CMPSC-Enhancement Facility

The *CMPSC-enhancement facility* may be available on a model implementing z/Architecture. The facility provides performance improvements for the COMPRESSION CALL instruction. (August, 2010)

Compare-and-Swap-and-Store Facility

The *compare-and-swap-and-store facility* may be available on a model implementing z/Architecture. The facility performs compare-and-swap operations

on 4- or 8-byte operands using interlocked update. If the operands are equal, a subsequent store operation of 1, 2, 4, or 8 bytes is performed. The facility provides the COMPARE AND SWAP AND STORE instruction. (April, 2007)

Compare-and-Swap-and-Store Facility 2

The *compare-and-swap-and-store facility* 2 may be available on a model implementing z/Architecture. The facility extends the compare-and-swap-and-store facility by providing a 16-byte compare-and-swap function and a 16-byte store operation. (February, 2008)

Conditional-SSKE Facility

The *conditional-SSKE facility* may be available on a model implementing z/Architecture. The facility provides performance improvements for the SET STORAGE KEY EXTENDED instruction. (April, 2007)

Configuration-Topology Facility

The *configuration-topology facility* may be available on a model implementing z/Architecture. The facility provides additional topology awareness to the program such that certain optimizations can be performed to improve cache hit ratios and thereby improve overall performance. The facility provides:

- The PERFORM TOPOLOGY FUNCTION (PTF) control instruction.
- A new system-information block (SYSIB 15.1.2) stored by the STORE SYSTEM INFORMATION instruction.

(February, 2008)

Configuration-z/Architecture-Architectural-Mode Facility

The *configuration-z/Architecture-architectural-mode (CZAM) facility* may be present on a model implementing z/Architecture. When the facility is installed in a configuration, the configuration is reset into the z/Architecture architectural mode (rather than into the ESA/390 architectural mode), and the configuration cannot be switched into the ESA/390 architectural mode.

The facility may be installed in a configuration operating in a logical partition and in a configuration operating as a guest of a logical partition. (September, 2017)

Constrained-Transactional-Execution Facility

The *constrained-transactional-execution facility* may available on a model in which the transactional-execution facility is installed. The constrained-transactional-execution facility ensures that – in the absence of repeated interruptions or conflicts with other CPUs or the I/O subsystem – transactional execution will eventually complete; thus, an abort-handler routine is not required. The facility provides the TRANSACTION BEGIN (TBEGINC) general instruction. (September, 2012)

DAT-Enhancement Facility 1

The *DAT-enhancement facility 1* may be available on a model implementing z/Architecture. The facility provides the following instructions:

- COMPARE AND SWAP AND PURGE (CSPG)
- INVALIDATE DAT TABLE ENTRY

COMPARE AND SWAP AND PURGE (CSPG) provides function similar to that of COMPARE AND SWAP AND PURGE (CSP), but CSPG has 64-bit operands whereas CSP has 32-bit operands.

INVALIDATE DAT TABLE ENTRY provides the means by which one or more region-table and segment-table entries in storage may be invalidated, and the corresponding TLB entries may be purged. The instruction provides an invalidation-and-clearing operation which invalidates and clears entries based on a specified virtual address, or a clear-by-ASCE operation which clears TLB entries based on the specified ASCE. (June, 2003)

DAT-Enhancement Facility 2

The *DAT-enhancement facility 2* may be available on a model implementing z/Architecture. When the DAT-enhancement facility 2 is installed, the LOAD PAGE-

TABLE-ENTRY ADDRESS instruction is available. Given a virtual address, the LOAD PAGE-TABLE-ENTRY ADDRESS instruction returns the 64-bit real address of the corresponding page-table entry. The address-space-control mode used by the dynamic-address-translation process is specified in the M_4 field of the instruction. (September, 2005)

Decimal-Floating-Point Facility

The *decimal-floating-point facility* supports three decimal-floating-point (DFP) data formats and provides 54 new instructions to operate on data in these formats. The formats: 32-bit (short), 64-bit (long), and 128-bit (extended), were developed in collaboration with the IEEE floating-point working group. (April, 2007)

Decimal-Floating-Point Packed-Conversion Facility

The *decimal floating point packed-conversion facility* may be available on models implementing the DFP facility. The facility includes the following instructions:

- CONVERT FROM PACKED (CDPT)
- CONVERT FROM PACKED (CXPT)
- CONVERT TO PACKED (CPDT)
- CONVERT TO PACKED (CPXT)

(March, 2015)

Decimal-Floating-Point-Rounding Facility

The *decimal-floating-point-rounding facility* provides a 3-bit DFP rounding mode field in the floating-point control (FPC) register, and the instruction SET DFP ROUNDING MODE, which may be used to set this field. The DFP rounding mode can specify any of eight rounding methods, including the five required by the IEEE floating-point working group.

The DFP rounding mode is used by PFPO and the decimal-floating-point instructions. (April, 2007)

Decimal-Floating-Point Zoned-Conversion Facility

The *decimal floating point zoned-conversion facility* may be available on models implementing the DFP facility. The facility includes the following features and instructions:

1. In Chapter 8, “Decimal Instructions,” the zoned format is defined to include the ASCII zone value (0011 binary).
2. In Chapter 20, “Decimal-Floating-Point Instructions,” the following instructions are defined:
   - CONVERT FROM ZONED (CDZT)
   - CONVERT FROM ZONED (CXZT)
   - CONVERT TO ZONED (CZDT)
   - CONVERT TO ZONED (CZXT)

(September, 2012)

Distinct-Operands Facility

The *distinct-operands facility* may be available on a model implementing z/Architecture. The facility provides alternate forms of selected arithmetic and logical instructions in which the result register may be different from either of the source registers. The facility provides alternate forms for the following instructions.

- ADD
- ADD IMMEDIATE
- ADD LOGICAL
- ADD LOGICAL WITH SIGNED IMMEDIATE
- AND
- EXCLUSIVE OR
- OR
- SHIFT LEFT SINGLE
- SHIFT LEFT SINGLE LOGICAL
- SHIFT RIGHT SINGLE
- SHIFT RIGHT SINGLE LOGICAL
- SUBTRACT
- SUBTRACT LOGICAL

(August, 2010)

Enhanced-DAT Facility 1

The *enhanced-DAT facility* 1 may be available on models implementing z/Architecture. When the facil-

ity is installed and enabled, DAT translation may produce either a page-frame real address or a segment-frame absolute address, determined by the STE-format control in the segment-table entry.

When the facility is installed in a configuration, a new bit in control register 0 enables the facility.

Note: The term *EDAT-1 applies* is used pervasively in this document to describe the condition of when the enhanced-DAT facility 1 is installed in the configuration and enabled by control register 0. See "Enhanced-DAT Terminology:" on page 3-41 for details on this terminology.

When EDAT-1 applies, the following function is available in the DAT process:

- Region-table entries include a DAT-protection bit, providing function similar to the DAT-protection bits in the segment- and page-table entries.
- The segment-table entry includes a STE-format control. When the STE-format control is zero, DAT proceeds as if EDAT-1 does not apply.
- When the STE-format control is one, the segment-table entry also contains the following:
  - A segment-frame absolute address (rather than a page-table origin) specifying the absolute storage location of the 1 M-byte block.
  - Access-control bits and a fetch-protection bit which optionally may be used instead of the corresponding bits in the segment's individual storage keys
  - A bit which determines the validity of the access-control bits and a fetch-protection bit in the segment-table entry

The facility adds the PERFORM FRAME MANAGEMENT FUNCTION control instruction. The facility includes enhancements or changes to the following control instructions:

- LOAD PAGE-TABLE-ENTRY ADDRESS
- MOVE PAGE
- SET STORAGE KEY EXTENDED
- TEST PROTECTION

(February, 2008)

Enhanced-DAT Facility 2

The *enhanced-DAT facility* 2 may be available on models implementing z/Architecture. When the facility is installed and enabled, DAT translation may produce either a page-frame real address, a segment-frame absolute address, or a region-frame absolute address, determined by format controls in the region-third-table entry (if any) and the segment-table entry (if any).

Note: The term *EDAT-2 applies* is used pervasively in this document to describe the condition of when the enhanced-DAT facility 2 is installed in the configuration and enabled by control register 0. See "Enhanced-DAT Terminology:" on page 3-41 for details on this terminology.

When EDAT-2 applies, the following function is available in the DAT process:

- EDAT-1 applies.
- The region-third-table entry includes a RTTE-format control. When the RTTE-format control is zero, DAT proceeds as is the case for when EDAT-1 applies.
- When the RTTE-format control is one, the region-third-table entry also contains the following:
  - A region-frame absolute address (rather than a segment-table origin) specifying the absolute storage location of the 2 G-byte block.
  - Access-control bits and a fetch-protection bit which optionally may be used in lieu of the corresponding bits in the region's individual storage keys
  - A bit which determines the validity of the access-control bits and a fetch-protection bit in the region-third-table entry

The enhanced-DAT facility 2 adds the COMPARE AND REPLACE DAT TABLE entry instruction, providing for the dynamic replacement of valid, attached DAT-table entries, and the selective clearing of any TLB entries created from the replaced entry.

The enhanced-DAT facility 2 also includes enhancements or changes to the following control instructions:

- INVALIDATE DAT TABLE ENTRY
- LOAD PAGE-TABLE-ENTRY ADDRESS
- MOVE PAGE
- PERFORM FRAME MANAGEMENT FUNCTION
- TEST PROTECTION

When the enhanced-DAT facility 2 is installed, the enhanced-DAT facility 1 is also installed. (September, 2012)

Enhanced-Monitor Facility

The *enhanced-monitor facility* may be available on a model implementing z/Architecture. The facility provides the means by which executions of the MONITOR CALL instruction may be counted in CPU-specific counter arrays where each entry in the array represents a different monitor code.

The enhanced-monitor facility is controlled by the enhanced-monitor masks in control register 8. The enhanced-monitor masks work in conjunction with the monitor masks in CR8. Existing programs that use the MONITOR CALL instruction continue to operate compatibly when the enhanced-monitor masks are zero; the enhanced counting operation is only provided when a monitor mask bit *and* its corresponding enhanced-monitor mask bit are both ones. (August, 2010)

Entropy Encoding Compression Facility

The *entropy-encoding-compression facility* may be available on models implementing z/Architecture. When the facility is installed an additional entropy encoding/decoding step may be performed as part of COMPRESSION CALL. (September, 2017)

ESA/390-Compatibility-Mode Facility

The *ESA/390-compatibility-mode (390-CM) facility* may be available on a model implementing z/Architecture. A configuration operating in the ESA/390-compatibility mode provides an environment supporting a subset of DAT-off ESA/390 programs in a hybrid architectural mode. See “ESA/390-Compatibility-Mode Facility” on page 5-111 for details. (September, 2017)

ETF2-Enhancement Facility

The *ETF2-enhancement facility* may be available on a model implementing z/Architecture. The facility includes modifications to the following instructions:

- New function is added to the TRANSLATE ONE TO ONE, TRANSLATE ONE TO TWO, TRANSLATE TWO TO ONE, and TRANSLATE TWO TO TWO instructions, allowing the test-character comparison to be bypassed.
- For TRANSLATE TWO TO ONE and TRANSLATE TWO TO TWO, the alignment requirements for the translate table are relaxed.

(September, 2005)

ETF3-Enhancement Facility

The *ETF3-enhancement facility* may be available on a model implementing z/Architecture. The facility provides improved well-formedness checking for the following instructions:

- CONVERT UTF-16 TO UTF-32
- CONVERT UTF-16 TO UTF-8
- CONVERT UTF-8 TO UTF-16
- CONVERT UTF-8 TO UTF-32

(September, 2005)

Execute-Extensions Facility

The *execute-extensions facility* may be available on a model implementing z/Architecture. The facility provides the EXECUTE RELATIVE LONG instruction. (February, 2008)

Execution-Hint Facility

The *execution-hint facility* may be available on a model implementing z/Architecture. The facility provides the means by which the program can provide hints which the CPU may take into consideration when deciding what information to maintain and use for storage-operand accesses and branch prediction. The facility provides the following instructions.

- BRANCH PREDICTION PRELOAD (BPP)

- BRANCH PREDICTION RELATIVE PRELOAD (BPRP)
- NEXT INSTRUCTION ACCESS INTENT (NIAI)

Additionally, new prefetch-intent codes are defined for PREFETCH DATA (PFD) and PREFETCH DATA RELATIVE LONG (PFDRL). (September, 2012)

Extended-Immediate Facility

The *extended-immediate facility* may be available on models implementing z/Architecture. The facility provides 32-bit immediate-operand versions of the following instructions:

- ADD IMMEDIATE
- ADD LOGICAL IMMEDIATE
- AND IMMEDIATE
- COMPARE IMMEDIATE
- COMPARE LOGICAL IMMEDIATE
- EXCLUSIVE OR IMMEDIATE
- INSERT IMMEDIATE
- LOAD IMMEDIATE
- LOAD LOGICAL IMMEDIATE
- OR IMMEDIATE
- SUBTRACT LOGICAL IMMEDIATE

Other new instructions added as a part of the extended-immediate facility include the following:

- FIND LEFTMOST ONE
- LOAD AND TEST (32-bit and 64-bit RXY format)
- LOAD BYTE (RRE format)
- LOAD HALFWORD (RRE format)
- LOAD LOGICAL CHARACTER (64-bit RRE format, and 32-bit RXY and RRE formats)
- LOAD LOGICAL HALFWORD (64-bit RRE format, and 32-bit RXY and RRE formats)

(September, 2005)

Extended-I/O-Measurement-Block Facility

The *extended-I/O-measurement-block facility* may be available on models implementing z/Architecture. The facility includes the following features:

- A new format of the channel-measurement block. The new measurement block, termed a format-1 channel-measurement block, is expanded to 64 bytes and is addressed using a separate measurement-block address for each subchannel. The new measurement-block format provides additional measurement information and the flexibility to store the measurement blocks in non-contiguous, real storage.
- The previously existing channel-measurement block is termed a format-0 channel-measurement block. A device-busy-time field is added to the format-0 channel-measurement block.

(June, 2003)

Extended-I/O-Measurement-Word Facility

The *extended-I/O-measurement-word facility* may be available on models implementing z/Architecture. The extended-measurement-word (EMW) is an extension to the interruption-response block (IRB) and allows channel-measurement data to be provided on an I/O operation basis. This reduces program overhead by alleviating the previous requirement that the program fetch the measurement block before and after an operation and calculate the difference between the respective measurement data values. (June, 2003)

Extended-Translation Facility 2

The *extended-translation facility 2* may be available on a model implementing z/Architecture. The facility performs operations on double-byte, ASCII, and decimal data. The double-byte data may be Unicode® data – that is, data that uses the binary codes of the Unicode Worldwide Character Standard and enables the use of characters of most of the world's written languages. The facility provides the following instructions:

- COMPARE LOGICAL LONG UNICODE
- MOVE LONG UNICODE
- PACK ASCII
- PACK UNICODE
- TEST DECIMAL
- TRANSLATE ONE TO ONE
- TRANSLATE ONE TO TWO
- TRANSLATE TWO TO ONE
- TRANSLATE TWO TO TWO
- UNPACK ASCII
- UNPACK UNICODE

The extended-translation facility 2 is called facility 2 since an extended-translation facility, now called facility 1, was introduced in ESA/390. Facility 1 is standard in z/Architecture. Facility 1 provides the instructions:

- CONVERT UNICODE TO UTF-8
- CONVERT UTF-8 TO UNICODE
- TRANSLATE EXTENDED

For when either or both of facility 1 and facility 2 are not installed on the machine, both facilities are simulated by the MVS CSRUNIC macro instruction, which is provided in OS/390® Release 10 and z/OS®.

OS/390 MVS Assembler Services Reference, GC28-1910-10, contains programming requirements, register information, syntax, return codes, and examples for the CSRUNIC macro instruction.

When CSRUNIC is used, the program exceptions listed in this publication do not cause program interruptions; instead, the exception conditions are indicated by CSRUNIC by means of return codes, as described in GC28-1910-10. (October, 2000)

Extended-Translation Facility 3

The *extended-translation facility 3* may be available on a model implementing z/Architecture. The facility performs operations on Unicode and Unicode-transformation-format (UTF) characters; it also includes a right-to-left TRANSLATE AND TEST operation. The facility provides the following instructions:

- CONVERT UTF-16 TO UTF-32
- CONVERT UTF-32 TO UTF-16
- CONVERT UTF-32 TO UTF-8
- CONVERT UTF-8 TO UTF-32
- SEARCH STRING UNICODE
- TRANSLATE AND TEST REVERSE

(May, 2004)

Extract-CPU-Time Facility

The *extract-CPU-time facility* may be available on a model implementing z/Architecture. The facility adds the general instruction EXTRACT CPU TIME. This instruction provides an efficient means by which a problem-state program can determine the amount of CPU time consumed, without requiring a supervisor-state service routine. (April, 2007)

Fast-BCR-Serialization Facility

The *fast-BCR-serialization facility* may be available on a model implementing z/Architecture. When the facility is installed, the execution of BRANCH ON CONDITION with the M_1 field containing 1110 binary and the R_2 field containing 0000 binary causes serialization without checkpoint synchronization. (August, 2010)

Fibre-Channel Extensions (FCX)

The *fibre-channel-extensions (FCX) facility* may be available on models implementing z/Architecture. The facility includes enhancements and performance extensions that provide the program a means to transport lists of commands to an I/O device for processing without the channel overhead of fetching and decoding each command.

Use of the FCX facility may be restricted to devices accessible by certain channel path types.

The FCX facility includes the following:

- Changes to the operation-request block (ORB), interruption-response block (IRB), and subchannel-information block (SCHIB).
- A form of a channel program that consists of a transport-control word (TCW) that designates a transport-command-control block (TCCB) and transport-status block (TSB). The TCCB includes a transport-command area (TCA) which contains a list of up to 30 I/O commands that are in the form of device-command words (DCWs). The TSB contains completion status and other information related to the TCW channel program that is complementary to the completion information contained in the IRB.

  A TCW may specify the transfer of either input data or output data.
- The ability to directly or indirectly designate any or all of the TCCB, the input data storage area, and the output data storage area. When a storage area is designated directly, the TCW specifies the location of a single, contiguous block of storage. When a storage area is designated indi-

rectly, the TCW designates the location of a list of one or more transport-indirect-data-address words (TIDAWs). TIDAW lists and the storage area designated by each TIDAW in a list are restricted from crossing 4 K-byte boundaries.

- An interrogate operation which may be initiated by CANCEL SUBCHANNEL to determine the state of an I/O operation.

(February, 2009)

FCX-Bidirectional-Data-Transfer Facility

The *FCX-bidirectional-data-transfer facility* may be available on a model implementing fibre-channel extensions. When the facility is installed and an attached device supports bidirectional-data transfer, transport-mode channel programs may specify the transfer of both input and output data. (September, 2012)

Floating-Point Extension Facility

The *floating-point extension facility* may be available on models implementing the DFP facility. The extension includes the following features and instructions:

Additions to Chapter 9, "Floating-Point Overview and Support Instructions:"

- A new floating-point-support instruction, SET BFP ROUNDING MODE (SRNMB), is added.
- A new exception, the quantum exception, is defined for DFP computational operations.
- Bit 5 of the floating-point-control (FPC) register is assigned to the quantum-exception mask
- Bit 13 of the FPC register is assigned to the quantum-exception flag.
- Data-exception code (DXC) 04 (hex) is assigned to the quantum exception.
- DXC 07 (hex) is assigned to the simulated quantum exception.
- The BFP-rounding-mode field in the FPC register is changed to 3 bits to support one additional BFP rounding mode, round to prepare for shorter precision.
- One new value of the effective rounding method field is assigned to support the round to prepare for shorter precision rounding method for CONVERT HFP TO BFP.
- For PFPO with a DFP result, bit 58 of general register 0 is assigned to be the DFP quantum-permission control.

Additions to Chapter 19, "Binary-Floating-Point Instructions:"

- The following new BFP instructions are added:
  - CONVERT FROM LOGICAL (CXLFBR, CDLFBR, CELFBR, CXLGBR, CDLGBR, and CELGBR).
  - CONVERT TO LOGICAL (CLFXBR, CLFDBR, CLFEBR, CLGXBR, CLGDBR, and CLGEBR)
- One new value of the effective rounding method field is assigned to support the round to prepare for shorter precision rounding method for CONVERT TO FIXED, DIVIDE TO INTEGER, and LOAD FP INTEGER
- An IEEE-inexact-exception control (XxC) is added to CONVERT TO FIXED and LOAD FP INTEGER.
- An effective rounding method field and an IEEE-inexact-exception control (XxC) are added to CONVERT FROM FIXED and LOAD ROUNDED.

Additions to Chapter 20, "Decimal-Floating-Point Instructions:"

- The following new DFP instructions are added:
  - CONVERT FROM FIXED (CXFTR, CDFTR)
  - CONVERT FROM LOGICAL (CXLGTR, CDLGTR, CXLFTR, CDLFTR)
  - CONVERT TO FIXED (CFXTR, CFDTR)
  - CONVERT TO LOGICAL (CLGXTR, CLGDTR, CLFXTR, CLFDTR)
- All reserved values in the effective rounding method field and a quantum-exception control (XqC) are assigned for LOAD FP INTEGER, LOAD ROUNDED, QUANTIZE, and REROUND.
- All reserved values in the effective rounding method field are assigned for CONVERT TO FIXED.
- An IEEE-inexact-exception control (XxC) is added to CONVERT TO FIXED and LOAD ROUNDED.
- An effective rounding method field and a quantum-exception control (XqC) are added to ADD, DIVIDE, MULTIPLY, and SUBTRACT.
- An effective rounding method field, an IEEE-inexact-exception control (XxC), and a quantum-exception control (XqC) are added to CONVERT FROM FIXED.

(August, 2010).

Floating-Point-Support-Sign-Handling Facility

The *floating-point-support-sign-handling* facility includes four instructions: COPY SIGN, LOAD COMPLEMENT, LOAD NEGATIVE, and LOAD POSITIVE. These instructions operate on a 64-bit (long) floating-point datum independent of the radix and do not set the condition code. (April, 2007)

FPR-GR-Transfer Facility

The *FPR-GR-transfer facility* includes the following two instructions:

- LOAD FPR FROM GR
- LOAD GR FROM FPR

These instructions provide the means to move a 64-bit (long) floating-point datum between a floating-point register and a general register. (April, 2007)

General-Instructions-Extension Facility

The *general-instructions-extension facility* may be available on a model implementing z/Architecture. The facility provides the following new instructions:

- ADD LOGICAL WITH SIGNED IMMEDIATE
- COMPARE AND BRANCH
- COMPARE AND BRANCH RELATIVE
- COMPARE AND TRAP
- COMPARE HALFWORD RELATIVE LONG
- COMPARE IMMEDIATE AND BRANCH
- COMPARE IMMEDIATE AND BRANCH RELATIVE
- COMPARE IMMEDIATE AND TRAP
- COMPARE LOGICAL AND BRANCH
- COMPARE LOGICAL AND BRANCH RELATIVE
- COMPARE LOGICAL AND TRAP
- COMPARE LOGICAL IMMEDIATE AND BRANCH
- COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE
- COMPARE LOGICAL IMMEDIATE AND TRAP
- COMPARE LOGICAL RELATIVE LONG
- COMPARE RELATIVE LONG
- EXTRACT CPU ATTRIBUTE
- LOAD HALFWORD RELATIVE LONG
- LOAD LOGICAL HALFWORD RELATIVE LONG
- LOAD LOGICAL RELATIVE LONG
- LOAD RELATIVE LONG
- MULTIPLY SINGLE IMMEDIATE
- PREFETCH DATA
- PREFETCH DATA RELATIVE LONG
- ROTATE THEN AND SELECTED BITS
- ROTATE THEN EXCLUSIVE OR SELECTED BITS
- ROTATE THEN INSERT SELECTED BITS
- ROTATE THEN OR SELECTED BITS
- STORE HALFWORD RELATIVE LONG
- STORE RELATIVE LONG

Additionally, the following instructions have been enhanced to include additional formats:

- ADD IMMEDIATE
- COMPARE HALFWORD
- COMPARE HALFWORD IMMEDIATE
- COMPARE LOGICAL IMMEDIATE
- LOAD ADDRESS EXTENDED
- LOAD AND TEST
- MOVE
- MULTIPLY
- MULTIPLY HALFWORD

(February, 2008)

Guarded-Storage Facility

The *guarded-storage facility* may be available on a model implementing z/Architecture. The facility provides a means by which a problem-state program can designate an area of logical storage comprising guarded-storage sections. The facility includes the following instructions:

- LOAD GUARDED (LGG)
- LOAD LOGICAL AND SHIFT GUARDED (LLGFSG)
- LOAD GUARDED STORAGE CONTROLS (LGSC)
- STORE GUARDED STORAGE CONTROLS (STGSC)

When the facility is installed in the configuration, enabled by the control program, and the second operand of the LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED designates a guarded-storage section, a guarded-storage event is recognized,

and control is passed to a guarded-storage event handler. Otherwise, the respective instructions simply perform their defined load operations.

The guarded-storage facility is intended to be used by various programming languages that implement storage-reclamation techniques commonly referred to as *garbage collection*. All other instructions that access a range of guarded storage are unaffected by the facility, only the LGG and LLGFSG instructions are capable of generating a guarded-storage event. (September, 2017)

HFP Multiply-and-Add/Subtract Facility

The *HFP-multiply-and-add/subtract facility* provides instructions for improved processing of hexadecimal floating-point numbers. The MULTIPLY AND ADD (or SUBTRACT) instruction is intended to be used in place of MULTIPLY followed by ADD (or SUBTRACT) NORMALIZED. (June, 2003)

HFP-Unnormalized-Extensions Facility

The *HFP-unnormalized-extension facility* may be available on a model implementing z/Architecture. The facility provides instructions for improved processing of unnormalized hexadecimal floating-point numbers. It extends the capabilities of the HFP-multiply-and-add/subtract facility, by providing the following instructions that operate on unnormalized operands:

- MULTIPLY UNNORMALIZED
- MULTIPLY AND ADD UNNORMALIZED

(September, 2005)

High-Word Facility

The *high-word facility* may be available on a model implementing z/Architecture. For selected 32-bit instructions, the high-word facility effectively provides sixteen additional 32-bit registers by utilizing bits 0-31 of the sixteen 64-bit general registers. The facility provides the following instructions.

- ADD HIGH
- ADD IMMEDIATE HIGH
- ADD LOGICAL HIGH
- ADD LOGICAL WITH SIGNED IMMEDIATE HIGH
- BRANCH RELATIVE ON COUNT HIGH
- COMPARE HIGH
- COMPARE IMMEDIATE HIGH
- COMPARE LOGICAL HIGH
- COMPARE LOGICAL IMMEDIATE HIGH
- LOAD BYTE HIGH
- LOAD HALFWORD HIGH
- LOAD HIGH
- LOAD LOGICAL CHARACTER HIGH
- LOAD LOGICAL HALFWORD HIGH
- ROTATE THEN INSERT SELECTED BITS HIGH
- ROTATE THEN INSERT SELECTED BITS LOW
- STORE CHARACTER HIGH
- STORE HALFWORD HIGH
- SUBTRACT HIGH
- SUBTRACT LOGICAL HIGH

(August, 2010)

IEEE-Exception-Simulation Facility

The *IEEE-exception-simulation facility* includes the instructions SET FPC AND SIGNAL and LOAD FPC AND SIGNAL. These instructions provide a means to simulate a data exception program interruption. (April, 2007)

Insert-Reference-Bits-Multiple Facility

The *insert-reference-bits-multiple* facility may be available on a model implementing z/Architecture. The facility provides the means by which the reference bits for 64 contiguous blocks of storage can be inspected in a single instruction. (September, 2017)

Instruction-Execution-Protection Facility

The *instruction-execution-protection facility* may be available on a model implementing z/Architecture. When the facility is installed and enabled, and an instruction is fetched from the primary or home address space, an instruction-execution-protection control in the leaf DAT-table entry used in the translation determines whether instructions may or may not be executed from the frame mapped by the entry.

The facility may be used by a control program to better segregate instructions from data. Improved system reliability and integrity may be realized by preventing the execution of instructions from storage locations intended to contain only data. For example, erroneously or maliciously modified data in a program stack can be prevented from being executed. (September, 2017)

Interlocked-Access Facility 1

The *interlocked-access facility 1* may be available on a model implementing z/Architecture. The facility provides the means by which a load, update, and store operation can be performed with interlocked update in a single instruction (as opposed to using a compare-and-swap type of update). The facility also provides an instruction to attempt to load from two distinct storage locations in an interlocked-fetch manner. The facility provides the following instructions.

- LOAD AND ADD
- LOAD AND ADD LOGICAL
- LOAD AND AND
- LOAD AND EXCLUSIVE OR
- LOAD AND OR
- LOAD PAIR DISJOINT

Additionally, when the interlocked-access facility 1 is installed, the storage-operand update reference for the following instructions appears to be an interlocked-update reference as observed by other CPUs and channel programs when the first operand is aligned on an integral boundary corresponding to its size:

- ADD IMMEDIATE (ASI and AGSI)
- ADD LOGICAL WITH SIGNED IMMEDIATE

(August, 2010)

Interlocked-Access Facility 2

The *interlocked-access facility 2* may be available on a model. When installed, the storage-operand update reference for the following instructions appears to be an interlocked-update as observed by other CPUs and channel programs.

- AND (NI, NIY)
- OR (OI, OIY)
- EXCLUSIVE OR (XI, XIY)

(September, 2012)

IPTE-Range Facility

The *IPTE-range facility* may be available on a model implementing z/Architecture. The facility provides performance improvements for the INVALIDATE PAGE TABLE ENTRY instruction. (August, 2010)

List-Directed Initial Program Load

The *list-directed initial-program-load* (IPL) function may be available on a model. List-directed IPL, also known as SCSI IPL in other System Library publications, provides the means by which a program can be loaded from an I/O device other than a classical channel-attached device (ESCON or FICON). Facilities are also provided for loading a stand-alone dump program using list-directed IPL. (May, 2004)

Load-and-Trap Facility

The *load-and-trap facility* may be available on a model implementing z/Architecture. The facility adds instructions that can be used to assist the program in determining when a value of all zeros (an address for example) has been loaded into a designated register. The facility provides the following instructions:

- LOAD AND TRAP (LAT and LGAT)
- LOAD HIGH AND TRAP (LFHAT)
- LOAD LOGICAL AND TRAP (LLGFAT)
- LOAD LOGICAL THIRTY ONE BITS AND TRAP (LLGTAT)

(September, 2012)

Load-and-Zero-Rightmost-Byte Facility

The *load-and-zero-rightmost-byte facility* may be available on a model implementing z/Architecture. The facility instructions of the facility load a value into a register from storage and zero the rightmost eight bits of the register. The facility provides the following instructions:

- LOAD AND ZERO RIGHTMOST BYTE (LZRF, LZRG)

- LOAD LOGICAL AND ZERO RIGHTMOST BYTE (LLZRGF)

(March, 2015)

Load/Store-on-Condition Facility 1

The *load/store-on-condition facility 1* may be available on a model implementing z/Architecture. The facility provides the means by which selected operations may be executed only when a condition-code-mask field of the instruction matches the current condition code in the PSW. The facility provides the following instructions.

- LOAD ON CONDITION
- STORE ON CONDITION

(August, 2010)

Load/Store-on-Condition Facility 2

The *load/store-on-condition facility 2* may be available on a model implementing z/Architecture. The facility provides six load-immediate operations, two load-high operations, and one store-high operation; all of which are executed only when a condition-code-mask field of the instruction matches the current condition code in the PSW. The facility includes the following instructions.

- LOAD HALFWORD HIGH IMMEDIATE ON CONDITION (LOCHHI)
- LOAD HALFWORD IMMEDIATE ON CONDITION (LOCHI, LOCGHI)
- LOAD HIGH ON CONDITION (LOCFHR, LOCFH)
- STORE HIGH ON CONDITION (STOCFH)

(March, 2015)

Local-TLB-Clearing Facility

The *local-TLB-clearing facility* may be available on a model implementing z/Architecture. The facility may provide performance improvements for the INVALIDATE DAT TABLE ENTRY and INVALIDATE PAGE TABLE ENTRY instructions by allowing the program to specify whether TLB clearing should be done in all CPUs of the configuration or only in the CPU executing the instruction. (September, 2012)

Long-Displacement Facility

The *long-displacement facility* provides a 20-bit signed-displacement field in 69 previously existing instructions (by using a previously unused byte in the instructions) and 44 new instructions. A 20-bit signed displacement allows relative addressing of up to 524,287 bytes beyond the location designated by a base register or base-and-index-register pair and up to 524,288 bytes before that location. The enhanced previously existing instructions generally are ones that handle 64-bit binary integers. The new instructions generally are new versions of instructions for 32-bit binary integers. The new instructions also include (1) a LOAD BYTE instruction that sign-extends a byte from storage to form a 32-bit or 64-bit result in a general register and (2) new floating-point LOAD and STORE instructions. The long-displacement facility provides register-constraint relief by reducing the need for base registers, code size reduction by allowing fewer instructions to be used, and additional improved performance through removal of possible address-generation interlocks. (June, 2003)

Message-Security Assist

The *message-security assist* (MSA) may be available on a model implementing z/Architecture. The MSA basic facility includes the following instructions:

- CIPHER MESSAGE
- CIPHER MESSAGE WITH CHAINING
- COMPUTE INTERMEDIATE MESSAGE DIGEST
- COMPUTE LAST MESSAGE DIGEST
- COMPUTE MESSAGE AUTHENTICATION CODE

Also included are five query functions and two functions for generating a message digest based on the secure-hash algorithm (SHA-1). The five query functions, one for each instruction, are used to determine the additional installed MSA facilities, which may include the following.

MSA Data-Encryption-Algorithm (DEA) Facility: The *MSA DEA facility* consists of nine functions for ciphering messages, with or without chaining, and for generating a message-authentication code (MAC) using a 56-bit, 112-bit, or 168-bit cryptographic key.[1]

All of these functions are based on the DEA algorithm. (June, 2003)

Message-Security-Assist Extension 1

The *message-security-assist extension 1* may be available on models implementing the message-security assist. The extension provides the following functions:

MSA SHA-256 Facility: This facility consists of two functions, one for generating an intermediate message digest and another for generating a final message digest.

MSA Advanced-Encryption-Standard (AES-128) Facility: This facility consists of two functions for ciphering messages, with or without chaining, using the AES-128 algorithm.

MSA Pseudo-Random-Number Generation (PRNG) Facility: This facility consists of a function for generating a multiple of 64-bit pseudo-random numbers using the ANSI® X9.17 pseudo-random-number algorithm.

(September, 2005)

Message-Security-Assist Extension 2

The *message-security-assist extension 2* may be available on models implementing the message-security assist. The extension provides the following functions:

MSA AES-192 Facility: This facility consists of two functions for ciphering a message, with or without chaining, using the AES-192 algorithm.

MSA AES-256 Facility: This facility consists of two functions for ciphering a message, with or without chaining, using the AES-256 algorithm.

MSA SHA-512 Facility: This facility consists of two functions, one for generating an intermediate message digest and another for generating a final message digest, using the SHA-512 algorithm.

(February, 2008)

Message-Security-Assist Extension 3

The *message-security-assist extension 3* provides a means to protect user cryptographic keys by encrypting them under machine-generated wrapping keys. When this extension is installed, two wrapping keys are provided for each configuration: one for protecting user DEA keys and another for protecting user AES keys. The wrapping keys reside in the machine so that, with an appropriate setting of controls, no clear value of user cryptographic keys is observed any where in the system by any program.

The message-security-assist extension 3 may be available on models implementing the message-security assist. The extension provides the following features:

- A 256-Bit AES Wrapping-Key Register: The register contents are used to protect user AES keys.
- A 256-Bit AES Wrapping-Key Verification-Pattern Register: The register contents are used to identify the version of the AES wrapping key.
- A 192-Bit DEA Wrapping-Key Register: The register contents are used to protect user DEA keys.
- A 192-Bit DEA Wrapping-Key Verification-Pattern Register: The register contents are used to identify the version of the DEA wrapping key.
- The following functions under the CIPHER MESSAGE instruction:
  - Encrypted DEA
  - Encrypted TDEA 128
  - Encrypted TDEA 192
  - Encrypted AES 128
  - Encrypted AES 192
  - Encrypted AES 256

  These functions use an encrypted cryptographic key.
- The following functions under the CIPHER MESSAGE WITH CHAINING instruction:
  - Encrypted DEA
  - Encrypted TDEA 128

1. These key lengths reflect the cryptographic strength. In subsequent chapters, they are referred to as 64-bit, 128-bit, or 192-bit, respectively, to include the DEA-key-parity bits.

- Encrypted TDEA 192
- Encrypted AES 128
- Encrypted AES 192
- Encrypted AES 256

These functions use an encrypted cryptographic key.

- The following functions under the COMPUTE MESSAGE AUTHENTICATION CODE instruction:

  - Encrypted DEA
  - Encrypted TDEA 128
  - Encrypted TDEA 192

  These functions use an encrypted cryptographic key.

- The PERFORM CRYPTOGRAPHIC KEY MANAGEMENT OPERATION instruction with the following functions:

  - Query
  - Encrypt DEA Key
  - Encrypt TDEA 128 Key
  - Encrypt TDEA 192 Key
  - Encrypt AES 128 Key
  - Encrypt AES 192 Key
  - Encrypt AES 256 Key

  These functions provide a means for importing clear cryptographic keys.

(August, 2010)

Message-Security-Assist Extension 4

The *message-security-assist extension 4* provides the support for the CFB (cipher feedback) mode, the OFB (output feedback) mode, and the CTR (counter) mode. In addition, a number of primitives are provided to facilitate the support for the CMAC (cipher-based message authentication code) mode, the CCM (counter with cipher block chaining – message authentication code) mode, the GCM (Galois/counter mode), and the XTS mode.

The message-security-assist extension 4 may be available on models implementing the message-security assist. The extension provides the following instructions and functions:

- The CIPHER MESSAGE WITH CIPHER FEEDBACK, CIPHER MESSAGE WITH COUNTER, and CIPHER MESSAGE WITH OUTPUT FEEDBACK instructions. Each of these instructions includes the following functions:

  - Query
  - DEA
  - TDEA 128
  - TDEA 192
  - Encrypted DEA
  - Encrypted TDEA 128
  - Encrypted TDEA 192
  - AES 128
  - AES 192
  - AES 256
  - Encrypted AES 128
  - Encrypted AES 192
  - Encrypted AES 256

- The PERFORM CRYPTOGRAPHIC COMPUTATION instruction with the following functions:

  - Query
  - Compute Last Block CMAC Using DEA
  - Compute Last Block CMAC Using TDEA 128
  - Compute Last Block CMAC Using TDEA 192
  - Compute Last Block CMAC Using Encrypted DEA
  - Compute Last Block CMAC Using Encrypted TDEA 128
  - Compute Last Block CMAC Using Encrypted TDEA 192
  - Compute Last Block CMAC Using AES 128
  - Compute Last Block CMAC Using AES 192
  - Compute Last Block CMAC Using AES 256
  - Compute Last Block CMAC Using Encrypted AES 128
  - Compute Last Block CMAC Using Encrypted AES 192
  - Compute Last Block CMAC Using Encrypted AES 256
  - Compute XTS Parameter Using AES 128
  - Compute XTS Parameter Using AES 256
  - Compute XTS Parameter Using Encrypted AES 128
  - Compute XTS Parameter Using Encrypted AES 256

- The following functions under the CIPHER MESSAGE instruction:

  - XTS AES 128
  - XTS AES 256
  - XTS Encrypted AES 128

    - XTS Encrypted AES 256

- The following function under the COMPUTE INTERMEDIATE MESSAGE DIGEST instruction:
    - GHASH
- The following functions under the COMPUTE MESSAGE AUTHENTICATION CODE instruction:
    - AES 128
    - AES 192
    - AES 256
    - Encrypted AES 128
    - Encrypted AES 192
    - Encrypted AES 256

The message-security-assist extension 4 requires the message-security-assist extension 3 as a prerequisite. (August, 2010)

Message-Security-Assist Extension 5

The *message-security-assist extension 5* provides support for deterministic pseudorandom-number seeding and generation that conforms to the National Institute of Standards and Technology (NIST) special publication 800-90A. The message-security-assist extension 5 may be available on models implementing the z/Architecture architectural mode. The extension provides the following instruction:

- PERFORM RANDOM NUMBER OPERATION (PRNO)

The PERFORM RANDOM NUMBER OPERATION instruction provides the following functions:

- PRNO-Query
- PRNO-SHA-512-DRNG

The message-security-assist extension 5 requires the secure-hash-algorithm (SHA-512) capabilities of the message-security-assist extension 2 as a prerequisite. (March, 2015)

Note: The PERFORM RANDOM NUMBER OPERATION instruction was formerly named PERFORM PSEUDORANDOM NUMBER OPERATION. With the addition of the message-security-assist extension 7, the same instruction is also capable of generating true random numbers, hence the name change. The former mnemonic PPNO is deprecated, but is still retained.

Message-Security-Assist Extension 6

The *message-security-assist extension 6* may be available on models implementing the message-security assist. The extension provides the following additional functions for the COMPUTE INTERMEDIATE MESSAGE DIGEST and COMPUTE LAST MESSAGE DIGEST instructions.

MSA SHA-3 Hash Facility: This facility consists of the following four functions for generating message digests having a length of 224, 256, 384, and 512 bits, using the respective SHA-3 functions defined in Reference [21.] on page xxx.

- KIMD-SHA3-224
- KIMD-SHA3-256
- KIMD-SHA3-384
- KIMD-SHA3-512
- KLMD-SHA3-224
- KLMD-SHA3-256
- KLMD-SHA3-384
- KLMD-SHA3-512

MSA SHA-3 Extendable-Output Facility: This facility consists of the following functions for producing message digests having any desired length using the SHAKE-128 and SHAKE-256 functions described in Reference [21.] on page xxx.

- KIMD-SHAKE-128
- KIMD-SHAKE-256
- KLMD-SHAKE-128
- KLMD-SHAKE-256

(September,2017)

Message-Security-Assist Extension 7

The *message-security-assist extension 7* may be available on models implementing the message-security-assist extension 5. The extension provides the following additional functions to the PERFORM RANDOM NUMBER OPERATION (PRNO) instruction related to true random number generation:

- PRNO-TRNG-Query-Raw-to-Conditioned Ratio
- PRNO-TRNG

(September,2017)

Message-Security-Assist Extension 8

The *message-security-assist extension 8* may be available on models implementing the z/Architecture architectural mode. The message-security-assist extension 8 provides support for both ciphering and authentication of a message, and the authentication of additional data (other than the message). The message-security-assist extension 8 provides the following instruction:

- CIPHER MESSAGE WITH AUTHENTICATION (KMA)

The CIPHER MESSAGE WITH AUTHENTICATION instruction provides support for the Galois-counter-mode (GCM) of ciphering, as described in Reference [17.] on page xxx. Functions provided by the instruction include the following:

- KMA-Query
- KMA-GCM-AES-128
- KMA-GCM-AES-192
- KMA-GCM-AES-256
- KMA-GCM-encrypted-AES-128
- KMA-GCM-encrypted-AES-192
- KMA-GCM-encrypted-AES-256

(September, 2017)

Miscellaneous-Instruction-Extensions Facility 1

The *miscellaneous-instruction-extensions facility 1* may be available on a model implementing the z/Architecture. The facility provides additional instruction formats for the following instructions:

- COMPARE LOGICAL AND TRAP (CLT and CLGT)
- ROTATE THEN INSERT SELECTED BITS (RISBGN)

(September, 2012)

Miscellaneous-Instruction-Extensions Facility 2

The *miscellaneous-instruction-extensions facility 2* may be available on a model implementing the z/Architecture. The facility provides the following additional instructions:

- ADD HALFWORD (AGH)
- BRANCH INDIRECT ON CONDITION
- MULTIPLY (MG, MGRK)
- MULTIPLY HALFWORD (MGH)
- MULTIPLY SINGLE (MSC, MSGC, MSGRKC, MSRKC)
- SUBTRACT HALFWORD (SGH)

(September, 2017)

Modified CCW Indirect Data Addressing Facility

The *modified-CCW-indirect-data-addressing (MIDA) facility* may be available on models implementing z/Architecture and provides the program an alternate means to transfer large amounts of data that spans noncontiguous blocks in main storage without the overhead of data chaining and without the strict boundary and count restrictions imposed by CCW indirect data addressing (IDA). Modified CCW indirect data addressing permits a single channel-command word to control the transfer of up to 65,535 bytes of data that spans noncontiguous blocks in main storage. Each block of main storage to be transferred may be specified on any boundary and length up to 4K bytes, provided the specified block does not cross a 4 K-byte boundary.

Use of modified CCW indirect data addressing may be restricted to devices accessible by certain channel-path types.

When modified CCW indirect data addressing is used, the CCW data address is not used to directly address data. Instead, the address points to a contiguous list of up to 256 quadwords called the modified-indirect-data-address list (MIDAL). Each quadword in the MIDAL is called a modified-indirect-data-address word (MIDAW) that describes a block of storage to be transferred and contains flags, a byte count, and a 64-bit address designating a data area in absolute storage. When modified CCW indirect data addressing is used, transfer of control from one MIDAW to

the next is made when the count of bytes specified by the MIDAW count field has been transferred and the total count of bytes transferred does not yet equal the count specified by the CCW. This is dissimilar to indirect data addressing where the transfer of control from one IDAW to the next is made when a program-specified 2K or 4K boundary is reached and the total count of bytes transferred does not yet equal the count specified in the CCW.

In addition to the MIDAW, the ORB modified-CCW-indirect-data-addressing-control bit and the CCW modified-indirect-data-address flag are added. (September, 2005)

Move-With-Optional-Specifications Facility

The *move-with-optional-specifications facility* may be available on a model implementing z/Architecture. The facility adds the semiprivileged MOVE WITH OPTIONAL SPECIFICATIONS instruction. This instruction provides the means of moving from a source operand to a destination operand using different address-space-control modes and different keys for each operand. (February, 2008)

Multiple-Epoch Facility

The *multiple-epoch facility* may be available on a model implementing z/Architecture. The facility provides the means by which a nonzero epoch index may be stored in byte 0 of the operand of the STORE CLOCK EXTENDED instruction and in bits 9-15 of the trace entry formed by the TRACE (TRACG) instruction.

The multiple-epoch facility also provides four additional functions for the PERFORM TIMING FACILITY FUNCTION (PTFF) instruction:

- Query steering information extended
- Query TOD offset user extended
- Set TOD offset extended
- Set TOD offset user extended

Additionally, the multiple-epoch facility provides the clock-comparator sign control, allowing the comparisons of the clock comparator to be either signed or unsigned.

When the multiple-epoch facility is installed, the store-clock-fast and TOD-clock-steering facilities are also installed. (September,2017)

Multiple-Subchannel-Set Facility

The *multiple-subchannel-set (MSS) facility* may be available on a model implementing z/Architecture and increases the maximum number of subchannels that can be configured to a program. When the MSS facility is not installed, a single set of subchannels, in the range 0-65,535, may be provided. When the MSS facility is installed, a maximum of four sets of subchannels may be provided for a program. Each subchannel set provides from one to 64K subchannels in the range 0 to-65,535. (February, 2008)

Multithreading Facility

The *multithreading (MT) facility* may be available on models implementing z/Architecture. The facility provides the means by which more efficient utilization of a configuration's resources may be realized. The facility introduces the architectural concept of a core, which, when multithreading is enabled, comprises a group of CPUs (sometimes called threads). The CPU comprises all of the architected resources available to the program such as the program-status word, registers, timing facilities, and so forth. When the multithreading facility is not enabled, a core consists of a single CPU.

When the multithreading facility is enabled, the CPUs within a core may share certain hardware resources such as execution units or caches. When one CPU in a core is waiting for other hardware resources (typically, while waiting for a storage access), other CPUs in the core can utilize the shared resources in the core rather than have them remain idle. (March, 2015)

Nonquiescing Key-Setting Facility

The *nonquiescing key-setting facility* may be available on a model implementing z/Architecture. The facility provides performance improvements for the PERFORM FRAME MANAGEMENT FUNCTION and SET STORAGE KEY EXTENDED instruction. (August, 2010)

Order Preserving Compression Facility

The *order-preserving-compression facility* may be available on models implementing z/Architecture. When the facility is installed the ordering of compressed symbols is the same as the ordering of uncompressed data when compression is performed by COMPRESSION CALL. (September, 2017)

Parsing-Enhancement Facility

The *parsing-enhancement facility* may be available on a model implementing z/Architecture. The facility adds the following instructions:

- TRANSLATE AND TEST EXTENDED
- TRANSLATE AND TEST REVERSE EXTENDED

These instructions perform functions similar to those of the TRANSLATE AND TEST and TRANSLATE AND TEST REVERSE instructions, respectively, but provide the capability of processing either single-byte or double-byte argument characters, returning either 8-bit or 16-bit function codes. (February, 2008)

PER-3 Facility

The *PER-3 facility* may be available on a model implementing z/Architecture. When the facility is installed, the following two functions are available for use by the program:

Breaking-Event Address Register: The breaking-event-address register is a 64-bit CPU register that is updated with the address of any instruction that causes a break in sequential instruction execution (that is, the instruction address in the PSW is replaced, rather than incremented by the length of the instruction). When the PER-3 facility is installed and a program interruption occurs, whether or not PER is indicated, the contents of the breaking-event-address register are stored in real storage locations 272-279. This can be used as a debugging assist for wild-branch detection.

PER Instruction-Fetching Nullification: Bit 39 of control register 9, when one, specifies that PER instruction-fetching events force nullification. Bit 39 of control register 9 is meaningful only when bit 33 of control register 9, the instruction-fetching PER-event mask, is also one. When the PER-3 facility is not installed, or when bit 39 of control register 9 is zero, nullification is not forced for PER instruction-fetching events (called a PER instruction-fetching basic event).

(September, 2005)

PER Zero-Address-Detection Facility

The *PER zero-address-detection facility* may be available on a model implementing z/Architecture. When the facility is installed and enabled, a PER zero-address-detection event is caused by execution of an instruction that accesses storage using an operand address formed from a general register containing zero. (August, 2010)

PFPO Facility

The *PFPO facility* provides the instruction PERFORM FLOATING-POINT OPERATION. This instruction, designed for future expansion, currently provides 54 conversion functions (from any of nine floating-point data formats to any of the six formats in another floating-point radix) using any of eight rounding methods. All conversions are correctly rounded. (April, 2007)

Population-Count Facility

The *population-count facility* may be available on a model implementing z/Architecture. The facility provides the POPULATION COUNT instruction which provides a count of one bits in each byte of a general register. (August, 2010)

Processor-Assist Facility

The *processor-assist facility* may be available on a model implementing z/Architecture. The facility provides the means by which the program can request that the processor perform an assist function. The program specifies which assist function is to be performed in an immediate (M_3) field of the instruction.

The facility provides the following general instruction:

- PERFORM PROCESSOR ASSIST

The processor-assist functions provided include the following:

- Transaction-abort assist

(September, 2012)

Reset-Reference-Bits-Multiple Facility

The *reset-reference-bits-multiple* facility may be available on a model implementing z/Architecture. The facility provides performance improvements for the inspection and resetting of reference bits by means of the RESET REFERENCE BITS MULTIPLE instruction. (August, 2010)

Restore-Subchannel Facility

The *restore-subchannel facility* may be available on a model implementing z/Architecture. The facility provides the means for the channel subsystem to recover a damaged subchannel and report the recovery to the program by means of a CRW. (February, 2008)

Server-Time-Protocol Facility

The *server-time-protocol facility* may be available on a model implementing z/Architecture. The facility provides the means by which the time-of-day (TOD) clocks in various systems can be synchronized using message links. STP operates in conjunction with the TOD-clock-steering facility, providing a new timing mode, new timing states, a new STP-timing-alert external interruption, and a new STP-sync-check machine-check conditions. (September, 2005)

Side-Effect-Access Facility

The *side-effect-access facility* may be available on a model implementing z/Architecture. When the facility is installed, the translation-exception identification (TEID) stored at real locations 168-175 during certain access exceptions contains an indication that the access was a side-effect access. A side-effect access is an implied access not directly associated with a storage operand of an instruction, is not an instruction fetch, is not a fetch of table information during ART or DAT, and is not a store of a trace entry. Additionally, when the side-effect-access facility is installed, an enhanced three-bit protection code is stored in the TEID. (September, 2017)

Store-Clock-Fast Facility

The *store-clock-fast facility* may be available on a model implementing z/Architecture. The facility provides a means by which an eight-byte time-of-day clock value may be stored without any artificial delay to ensure uniqueness. When the facility is installed, the TOD-clock bits stored by TRACE (TRACE and TRACG) are subject to additional control by a bit in control register 0. The facility provides the STORE CLOCK FAST instruction. (September, 2005)

Store-Facility-List-Extended Facility

The *store-facility-list-extended facility* may be available on a model implementing z/Architecture. The facility extends the function provided by the STORE FACILITY LIST (STFL) instruction. Whereas STFL is a control instruction that can store an indication of 32 facilities at real location 200, the new STORE FACILITY LIST EXTENDED (STFLE) instruction is a general instruction that can store an indication of up to 16,384 facilities at a program-specified location. (September, 2005)

Test-Pending-External-Interruption Facility

The *test-pending-external-interruption* facility may be available on a model implementing z/Architecture. The facility provides the means by which a control program can determine whether one or more of a subset of external interruptions is pending in the

CPU. The facility provides the TEST PENDING EXTERNAL INTERRUPTION instruction. (September, 2017)

TOD-Clock-Steering Facility

The *TOD-clock-steering facility* may be available on a model implementing z/Architecture. The facility provides a means by which apparent stepping rate of the time-of-day clock may be altered without changing the physical hardware oscillator which steps the physical clock. The facility adds the semiprivileged PERFORM TIMING FACILITY FUNCTION (PTFF) instruction which provides the means by which the program can query various timing-related parameters, and, optionally, the means by which an authorized timing-control program can influence certain of these parameters. (September, 2005)

Transactional-Execution Facility

The *transactional-execution facility* may be available on a model implementing z/Architecture. The facility provides the means by which a program can issue multiple instructions, the storage accesses of which either (a) appear to occur as a single concurrent operation or (b) do not appear to occur, as observed by other CPUs and by the channel subsystem.

The facility provides the following general instructions:

- EXTRACT TRANSACTION NESTING DEPTH
- NONTRANSACTIONAL STORE
- TRANSACTION ABORT
- TRANSACTION BEGIN (TBEGIN)
- TRANSACTION END

(September, 2012)

Vector-Enhancements Facility 1

The *vector-enhancements facility 1* may be available on models implementing the vector facility for z/Architecture. The facility adds support for the following features and functions:

The facility also includes the following new instructions:

- VECTOR BIT PERMUTE
- VECTOR MULTIPLY SUM LOGICAL
- VECTOR NOT EXCLUSIVE OR
- VECTOR NAND
- VECTOR OR WITH COMPLEMENT
- VECTOR FP MAXIMUM
- VECTOR FP MINIMUM

The facility adds support for halfword, word, and doubleword elements to VECTOR POPULATION COUNT

The facility adds support for BFP short format and BFP extended format elements to most instructions in chapter 24. (September, 2017)

Vector Facility for z/Architecture

The *vector facility for z/Architecture* may be available on models implementing z/Architecture. When the facility is installed and enabled, vector instructions are available, having access to 32 128-bit registers. The instructions are described in four chapters:

- Chapter 21 describes the vector facility support instructions
- Chapter 22 describes the vector facility integer instructions
- Chapter 23 describes the vector facility string instructions
- Chapter 24 describes the vector facility floating-point instructions.

Note: ESA/390 provided an optional vector facility, however this facility was never available on any processor capable of the z/Architecture architectural mode. The vector facility for z/Architecture differs from the ESA/390 vector facility in instruction and register definitions. (March, 2015)

Vector Packed-Decimal Facility

The *vector-packed-decimal facility* may be available on models implementing z/Architecture. When the facility is installed and enabled, vector decimal instructions are available, allowing operations on packed data in vector registers. The facility provides the following instructions:

- VECTOR ADD DECIMAL
- VECTOR COMPARE DECIMAL
- VECTOR CONVERT TO BINARY
- VECTOR CONVERT TO DECIMAL
- VECTOR DIVIDE DECIMAL
- VECTOR LOAD IMMEDIATE DECIMAL
- VECTOR LOAD RIGHTMOST WITH LENGTH
- VECTOR MULTIPLY DECIMAL
- VECTOR MULTIPLY AND SHIFT DECIMAL
- VECTOR PACK ZONED
- VECTOR PERFORM SIGN OPERATION DECIMAL
- VECTOR REMAINDER DECIMAL
- VECTOR SHIFT AND DIVIDE DECIMAL
- VECTOR SHIFT AND ROUND DECIMAL
- VECTOR STORE RIGHTMOST WITH LENGTH
- VECTOR SUBTRACT DECIMAL
- VECTOR TEST DECIMAL
- VECTOR UNPACK ZONED

(September, 2017)

Warning-Track Interruption Facility

The *warning-track-interruption facility* may be available on a model implementing the z/Architecture. The facility provides the means by which a warning-track external interruption can be presented to a CPU in a configuration with shared-CPU resources, such as a logical partition. The control program can use the warning-track external interruption as the signal to make the currently-executing dispatchable unit dispatchable on a different CPU of the configuration. (September, 2012)

The ESA/390 Base

z/Architecture includes all of the facilities of ESA/390 except for the asynchronous-pageout, asynchronous-data-mover, program-call-fast, and ESA/390 vector facilities. This section briefly outlines most of the remaining facilities that were additions in ESA/390 as compared to ESA/370.

ESA/390 is described in Reference [1.] on page xxix.

The CPU-related facilities that were new in ESA/390 are summarized below. ESA/390 was announced in September, 1990. Any extension added subsequently has the date of its announcement in parentheses at the end of its summary.

The following extensions are described in detail in SA22-7201 and in this publication:

- *Access-list-controlled protection* allows store-type storage references to an address space to be prohibited by means of a bit in the access-list entry used to access the space. Thus, different users having different access lists can have different capabilities to store in the same address space.

- The *program-event-recording facility 2 (PER 2)* is an alternative to the original PER facility, which is now named PER 1. (Neither of the names PER 1 and PER 2 is used in z/Architecture; only “PER” is used.) PER 2 provides the option of having a successful-branching event occur only when the branch target is within the designated storage area, and it provides the option of having a storage-alteration event occur only when the storage area is within designated address spaces. The use of these options improves performance by allowing only PER events of interest to occur. PER 2 deletes the ability to monitor for general-register-alteration events.

  PER 2 includes extensions that provide additional information about PER events. The extensions were described in detail beginning in the fourth edition of SA22-7201.

- *Concurrent sense* improves performance by allowing sense information to be presented at the time of an interruption due to a unit-check condition, thus avoiding the need for a separate I/O operation to obtain the sense information.

- *Broadcasted purging* provides the COMPARE AND SWAP AND PURGE instruction for conditionally updating tables associated with dynamic address translation and access-register translation and clearing associated buffers in multiple CPUs. This is described in detail beginning in the eighth edition of SA22-7201.

- *Storage-protection override* provides a new form of subsystem storage protection that improves the reliability of a subsystem executed in an address space along with possibly erroneous application programs. When storage-protection override is made active by a control-register bit, fetches and stores by the CPU are permitted to storage locations having a storage key of 9 regardless of the access key used by the CPU. If the subsystem is in key-8 storage and is exe-

cuted with a PSW key of 8, for example, and the application programs are in key-9 storage and are executed with a PSW key of 9, accesses by the subsystem to the application-program areas are permitted while accesses by the application programs to the subsystem area are denied. (September, 1991)

- *Move-page facility 2* extends the MOVE PAGE instruction introduced in ESA/370 by allowing use of a specified access key for either the source or the destination operand, by allowing improved performance when the destination operand will soon be referenced, and by allowing improved performance when an operand is invalid in both main and expanded storage. The ESA/370 version of MOVE PAGE is now called move-page facility 1 and is in Chapter 7, "General Instructions." MOVE PAGE of move-page facility 2 is in Chapter 10, "Control Instructions." Some details about the means for control-program support of MOVE PAGE are not provided. (September, 1991) (The z/Architecture MOVE PAGE instruction is described only in Chapter 10 of this publication. MOVE PAGE no longer can move data to or from expanded storage, and all details about MOVE PAGE are provided.)

- The *square-root facility* consists of the SQUARE ROOT instruction and the square-root exception. The instruction extracts the square root of a floating-point operand in either the long or short format. The instruction is the same as that provided on some models of the IBM 4341, 4361, and 4381 Processors. (September, 1991)

- The *cancel-I/O facility* allows the program to withdraw a pending start function from a designated subchannel without signaling the device, which is useful in certain error-recovery situations. (September, 1991)

  The cancel-I/O facility provides the CANCEL SUBCHANNEL instruction and is described in detail beginning in the eighth edition of SA22-7201.

- The *string-instruction facility* (or *logical string assist*) provides instructions for (1) moving a string of bytes until a specified ending byte is found, (2) logically comparing two strings until an inequality or a specified ending byte is found, and (3) searching a string of a specified length for a specified byte. The first two instructions are particularly useful in a C program in which strings are normally delimited by an ending byte of all zeros. (June, 1992)

- The *suppression-on-protection facility* causes a protection exception due to page protection to result in suppression of instruction execution instead of termination of instruction execution, and it causes the address and an address-space identifier of the protected page to be stored in low storage. This is useful in performing the AIX/ESA® copy-on-write function, in which AIX/ESA causes the same page of different address spaces to map to a single page frame of real storage so long as a store in the page is not attempted and then, when a store is attempted in a particular address space, assigns a unique page frame to the page in that address space and copies the contents of the page to the new page frame. (February, 1993)

- The *set-address-space-control-fast facility* consists of the SET ADDRESS SPACE CONTROL FAST (SACF) instruction, which possibly can be used instead of the previously existing SET ADDRESS SPACE CONTROL (SAC) instruction, depending on whether all of the SAC functions are required. SACF, unlike SAC, does not perform the serialization and checkpoint-synchronization functions, nor does it cause copies of prefetched instructions to be discarded. SACF provides improved performance on some models. (February, 1993)

- The *subspace-group facility* includes the BRANCH IN SUBSPACE GROUP instruction, which can be used to give or return control from one address space to another in a group of address spaces called a subspace group, with this giving and returning of control being done with better performance than can be obtained by means of the PROGRAM CALL and PROGRAM RETURN or PROGRAM TRANSFER instructions. One address space in the subspace group is called the base space, and the other address spaces in the group are called subspaces. It is intended that each subspace contain a different subset of the storage in the base space, that the base space and each subspace contain a subsystem control program, such as CICS®, and application programs, and that each subspace contain the data for a single transaction being processed under the subsystem control program. The placement of the data for each transaction in a different subspace prevents the processing of a transaction from erroneously damaging the data

of other transactions. The data of the control program can be protected from the transaction processing by means of the storage-protection-override facility. (April, 1994)

- The *virtual-address enhancement of suppression on protection* provides that if dynamic address translation (DAT) was on when a protection exception was recognized, the suppression-on-protection result is indicated, and the address of the protected location is stored, only if the address is one that was to be translated by DAT; the suppression-on-protection result is not indicated if the address that would be stored is a real address. This enhancement allows the stored address to be translated reliably by the control program to determine if the exception was due to page protection as opposed to key-controlled protection. The enhancement extends the usefulness of suppression on protection to operating systems like MVS/ESA™ that use key-controlled protection. (September, 1994)

- The *immediate-and-relative-instruction facility* includes 13 new instructions, most of which use a halfword-immediate value for either signed-binary arithmetic operations or relative branching. The facility reduces the need for general registers, and, in particular, it eliminates the need to use general registers to address branch targets. As a result, the general registers and access registers can be allocated more efficiently in programs that require many registers. (September, 1996)

- The *compare-and-move-extended facility* provides new versions of the COMPARE LOGICAL LONG and MOVE LONG instructions. The new versions increase the size of the operand-length specifications from 24 bits to 32 bits, which can be useful when objects larger than 16M bytes are processed through the use of 31-bit addressing. The new versions also periodically complete to allow software polling in a multiprocessing system. (September, 1996)

- The *checksum facility* consists of the CHECKSUM instruction, which can be used to compute a 16-bit or 32-bit checksum in order to improve Transmission-Control Protocol/Internet Protocol (TCP/IP) performance. (September, 1996)

- The *called-space-identification facility* improves serviceability by further identifying the called address space in a linkage-stack state entry formed by the PROGRAM CALL instruction. (September, 1996)

- The *branch-and-set-authority facility* consists of the BRANCH AND SET AUTHORITY instruction, which can be used to improve the performance of linkages within an address space by replacing PROGRAM CALL, PROGRAM TRANSFER, and SET PSW KEY FROM ADDRESS instructions. (June, 1997)

- The *perform-locked-operation facility* consists of the unprivileged PERFORM LOCKED OPERATION instruction, which appears to provide concurrent interlocked-update references to multiple storage operands. A function code of the instruction can specify any of six operations: compare and load, compare and swap, double compare and swap, compare and swap and store, compare and swap and double store, and compare and swap and triple store. The function code further specifies either word or doubleword operands. The instruction can be used to avoid the use of programmed locks in a multiprocessing system. (June, 1997)

- Four additional floating-point facilities improve the hexadecimal-floating-point (HFP) capability of the machine and add a binary-floating-point (BFP) capability. The facilities are:

  – *Basic floating-point extensions*, which provides 12 additional floating-point registers to make a total of 16 floating-point registers. This facility also includes a floating-point-control register and means for saving the contents of the new registers during a store-status operation or a machine-check interruption.

  – *Floating-point-support (FPS) extensions*, which provides eight new instructions, including four to convert data between the HFP and BFP formats.

  – *Hexadecimal-floating-point (HFP) extensions*, which provides 26 new instructions to operate on data in the HFP format. All of these are counterparts to new instructions provided by the BFP facility, including conversion between floating-point and fixed-point formats, and a more complete set of operations on the extended format.

  – *Binary floating-point (BFP)*, which defines short, long, and extended binary-floating-

point (BFP) data formats and provides 87 new instructions to operate on data in these formats. The BFP formats and operations provide everything necessary to conform to the IEEE standard (ANSI/IEEE Standard 754-2008, as defined in Reference [20.] on page xxx) except for conversion between binary-floating-point numbers and decimal strings, which must be provided in software.

(May, 1998)

- The *resume-program facility* consists of the RESUME PROGRAM instruction, which restores, from a specified save area, the instruction address and certain other fields in the current PSW and also the contents of an access-and-general-register pair. RESUME PROGRAM allows a problem-state interruption-handling program to restore the state of an interrupted program and return to that program despite that a register is required for addressing the save area from which the state is restored. (May, 1998)

- The *trap facility* provides the TRAP instructions (a two-byte TRAP2 instruction and a four-byte TRAP4 instruction) that can overlay instructions in an application program to give control to a program that can perform fix-up operations on data being processed, such as dates that may be a "Year-2000" problem. RESUME PROGRAM can be used to return from the fix-up program. TRAP and RESUME PROGRAM can improve performance by avoiding program interruptions that would otherwise be needed to give control to and from the fix-up program. (May, 1998)

- The *extended-TOD-clock facility* includes (1) an extension of the TOD clock from 64 bits to 104 bits, allowing greater resolution; (2) a TOD programmable register, which contains a TOD programmable field that can be used to identify the configuration providing a TOD-clock value in a sysplex; (3) the SET CLOCK PROGRAMMABLE FIELD instruction, for setting the TOD programmable field in the TOD programmable register; and (4) the STORE CLOCK EXTENDED instruction, which stores both the longer TOD-clock value and the TOD programmable field. STORE CLOCK EXTENDED can be used in the future when the TOD clock is further extended to contain time values that exceed the current year-2042 limit (when there is a carry out of the current bit 0 of the TOD clock). (August, 1998)

- The *TOD-clock-control-override facility* provides a control-register bit that allows setting the TOD clock under program control, without use of the manual TOD-clock control of any CPU. (August, 1998)

- The *store-system-information facility* provides the privileged STORE SYSTEM INFORMATION instruction, which can be used to obtain information about a component or components of a virtual machine, a logical partition, or the basic machine. (January, 1999)

- The *extended-translation facility*, now called the extended-translation facility 1, includes the CONVERT UNICODE TO UTF-8, CONVERT UTF-8 TO UNICODE, and TRANSLATE EXTENDED instructions, all of which can improve performance. TRANSLATE EXTENDED can be used in place of a TRANSLATE AND TEST instruction that locates an escape character, followed by a TRANSLATE instruction that translates the bytes preceding the escape character. (April, 1999)

The following extensions are described in detail in other publications:

- The *Enterprise Systems Connection Architecture® (ESCON®)* introduces a new type of channel that uses an optical-fiber communication link between channels and control units. Information is transferred serially by bit, at 200 million bits per second, up to a maximum distance of 60 kilometers. The optical-fiber technology and serial transmission simplify cabling and improve reliability. See the publication *IBM Enterprise Systems Architecture/390 ESCON I/O Interface*, SA22-7202.

- The *ESCON channel-to-channel adapter (ESCON CTCA)* provides the same type of function for serial channel paths as is available for the parallel-I/O-interface channel paths. See the publication *IBM Enterprise Systems Architecture/390 ESCON Channel-to-Channel Adapter*, SA22-7203.

- *I/O-device self-description* allows a device to describe itself and its position in the I/O configuration. See the publication *IBM Enterprise Systems Architecture/390 Common I/O-Device Commands and Self Description*, SA22-7204.

- The *compression facility* performs a Ziv-Lempel type of compression and expansion by means of

static (nonadaptive) dictionaries that are to be prepared by a program before the compression and expansion operations. Because the dictionaries are static, the compression facility can provide good compression not only for long sequential data streams (for example, archival or network data) but also for randomly accessed short records (for example, 80 bytes). See the publication *IBM Enterprise Systems Architecture/390 Data Compression*, SA22-7208. (February, 1993) (The z/Architecture COMPRESSION CALL instruction is described in this publication. However, introductory information and information about dictionary formats still is provided only in SA22-7208.)

The remaining extensions of ESA/390, for which detailed descriptions are not provided, are as follows:

- The integrated *cryptographic facility* provides a number of instructions to protect data privacy, to support message authentication and personal identification, and to facilitate key management. The high-performance cipher capability of the facility is designed for financial-transaction and bulk-encryption environments, and it complies with the Data Encryption Standard (DES).
  - Usability of the cryptographic facility is extended to virtual-machine environments, which allows the facility to be used by MVS/ESA being executed under VM/ESA®, which in turn may be executed either under another VM/ESA or in a logical partition. (September, 1991)
- The *external-time-reference facility* provides a means to initiate and maintain the synchronization of TOD clocks to an external time reference (ETR). Synchronization tolerance of a few microseconds can be achieved, and the effect of leap seconds is taken into account. The facility consists of an ETR sending unit (Sysplex Timer®), which may be duplexed, two or more ETR receiving units, and optical-fiber cables. The cables are used to connect the ETR sending unit, which is an external device, to ETR receiving units of the configuration. CPU instructions are provided for setting the TOD clock to the value supplied by the ETR sending unit.
  - The ETR *automatic-propagation-delay-adjustment* function adjusts the time signals from the ETR to the attached processors to compensate for the propagation delay on the cables to the processors, thus allowing the cables to be of different lengths. (September, 1991)
  - The ETR *external-time-source* function synchronizes the ETR to a time signal received from a remote location by means of a telephone or radio. (September, 1991)
- *Extended sorting* provides instructions that improve the performance of the DB2® sorting function.
- Other *PER extensions*, besides those described beginning in the fourth edition of this publication, are an augmentation of PER 2 that provide additional PER function in the interpretive-execution mode.
- *Channel-subsystem call* provides various functions for use in the management of the I/O configuration. Some of the functions acquire information about the configuration from the accessible elements of the configuration, while others dynamically change the configuration.
- The *operational extensions* are a number of other improvements that result in increased availability and ease of use of the system, as follows:
  - *Automatic-reconfiguration* permits an operating system in an LPAR partition to declare itself willing to be terminated suddenly, usually to permit its storage and CPU resources to be acquired by an adjacent partition that is dynamically absorbing the work load of another system that has failed. Other functions deactivate and reset designated participating partitions.
  - A new *storage-reconfiguration* command decreases the time needed to reconfigure storage by allowing multiple requests for reconfiguration to be made by means of a single communication with the service processor.
  - *SCP-initiated reset* allows a system control program (SCP) to reset its I/O configuration prior to entering the disabled wait state following certain check conditions.
  - *Console integration* simplifies configuration requirements by reducing by one the number of consoles required by MVS.

  - The *processor-availability facility* enables a CPU experiencing an unrecoverable error that will cause a check stop to save its state and alert the other CPUs in the configuration. This allows, in many cases, another CPU to continue execution of the program that was in execution on the failing CPU. The facility is applicable in both the ESA/390 mode and the LPAR mode. (April, 1991)

- *Extensions for virtual machines* are a number of improvements to the interpretive-execution facility, as follows:

  - The *VM-data-space facility* provides for making the ESA/390 access-register architecture more useful in virtual-machine applications. The facility improves the ability to address a larger amount of data and to share data. For information on how VM/ESA uses the VM-data-space facility, see the publication *VM/ESA CP Programming Services*, SC24-5520.

  - A new *storage-key function* improves performance by removing the need for the previously used RCP area.

  - Other improvements include an optional special-purpose lookaside for some of the guest-state information and greater freedom in certain implementation choices.

- The *ESCON-multiple-image facility (EMIF)* allows multiple logical partitions to share ESCON channels (and FICON channels) and optionally to share any of the control units and associated I/O devices configured to these shared channels. This can reduce channel requirements, improve channel utilization, and improve I/O connectivity. (June, 1992)

- *PR/SM LPAR* mode is enhanced to allow up to 10 logical partitions in a single-image configuration and 20 in a physically-partitioned configuration. The previous limits were seven and 14, respectively. (June, 1992)

  Coincident with z/Architecture, PR/SM LPAR mode allows 15 logical partitions, and physical partitioning is not supported.

- The *coupling facility* enables high-performance data sharing among MVS/ESA systems that are connected by means of the facility. The coupling facility provides storage that can be dynamically partitioned for caching data in shared buffers, maintaining work queues and status information in shared lists, and locking data by means of shared lock controls. MVS/ESA services provide access to and manipulation of the coupling-facility contents. (April, 1994)

The ESA/370 and 370-XA Base

ESA/390 includes the complete set of facilities of ESA/370 as its base. This section briefly outlines most of the facilities that were additions in 370-XA as compared to System/370 and that were additions in ESA/370 as compared to 370-XA.

The CPU-related facilities that were new in 370-XA are as follows:

- *Bimodal addressing* provides two modes of operation: a 24-bit addressing mode for the execution of old programs and a 31-bit addressing mode.

- *31-bit logical addressing* extends the virtual address space from the 16M bytes addressable with 24-bit addresses to 2G bytes (2,147,483,648 bytes).

- *31-bit real and absolute addressing* provides addressability for up to 2G bytes of main storage.

- The 370-XA *protection* facilities include key-controlled protection on only 4 K-byte blocks, page protection, and, as in System/370, low-address protection for addresses below 512. Fetch-protection override eliminates fetch protection for locations 0-2047.

- The *tracing* facility assists in the determination of system problems by providing an ongoing record in storage of significant events.

- The COMPARE AND FORM CODEWORD and UPDATE TREE instructions facilitate sorting applications.

- The *interpretive-execution facility* allows creation of virtual machines that may operate according to several architectures and whose performance is enhanced because many virtual-machine functions are directly interpreted by the machine rather than simulated by the program. This facility is described in the publication *IBM 370-XA Interpretive Execution*, SA22-7095.

- The *service-call-logical-processor (SCLP) facility* provides a means of communicating between the control program and the service processor for the purpose of describing and changing the configuration. This facility is not described.

The I/O-related differences between 370-XA and System/370 result from the 370-XA *channel subsystem*, which includes:

- *Path-independent addressing* of I/O devices, which permits the initiation of I/O operations without regard to which CPU is executing the I/O instruction or how the I/O device is attached to the channel subsystem. Any I/O interruption can be handled by any CPU enabled for it.
- *Path management*, whereby the channel subsystem determines which paths are available for selection, chooses a path, and manages any busy conditions encountered while attempting to initiate I/O processing with the associated devices.
- *Dynamic reconnection*, which permits any I/O device using this capability to reconnect to any available channel path to which it has access in order to continue execution of a chain of commands.
- *Programmable interruption subclasses*, which permit the programmed assignment of I/O-interruption requests from individual I/O devices to any one of eight maskable interruption queues.
- An *additional CCW format* for the direct use of 31-bit addresses in channel programs. The new CCW format, called format 1, is provided in addition to the System/370 CCW format, now called format 0.
- *Address-limit checking*, which provides an additional storage-protection facility to prevent data access to storage locations above or below a specified absolute address.
- *Monitoring facilities*, which can be invoked by the program to cause the channel subsystem to measure and accumulate key I/O-resource usage parameters.
- *Status-verification facility*, which reports inappropriate combinations of device-status bits presented by a device.
- A set of 13 *I/O instructions*, with associated control blocks, which are provided for the control of the channel subsystem.

The facilities that were new in ESA/370 are as follows:

- Sixteen *access registers* permit the program to have immediate access to storage operands in up to 16 2 G-byte address spaces, including the address space in which the program resides. In a dynamic-address-translation mode named *access-register mode*, the instruction B field, or for certain instructions the R field, designates both a general register and an access register, and the contents of the access register, along with the contents of protected tables, specify the operand address space to be accessed. By changing the contents of the access registers, the program, under the control of an authorization mechanism, can have fast access to hundreds of different operand address spaces.
- A *linkage stack* is used in a functionally expanded mechanism for passing control between programs in either the same or different address spaces. This mechanism makes use also of the previously existing PROGRAM CALL instruction, an extended entry-table entry, and a new PROGRAM RETURN instruction. The mechanism saves various elements of status, including access-register and general-register contents, during a calling linkage, provides for changing the current status during the calling linkage, and restores the original status during the returning linkage. The linkage stack can also be used to save and restore access-register and general-register contents during a branch-type linkage performed by the new instruction BRANCH AND STACK.
- A translation mode named *home-space mode* provides an efficient means for the control program to obtain control in the address space, called the home address space, in which the principal control blocks for a dispatchable unit (a task or process) are kept.
- The semiprivileged MOVE WITH SOURCE KEY and MOVE WITH DESTINATION KEY instructions allow bidirectional movement of data between storage areas having different storage keys, without the need to change the PSW key.

- The privileged LOAD USING REAL ADDRESS and STORE USING REAL ADDRESS instructions allow the control program to access data in real storage more efficiently.
- The *private-space facility* allows an address space not to contain any common segments and causes low-address protection and fetch-protection override not to apply to the address space.
- The unprivileged MOVE PAGE instruction allows the program to move a page of data between main and expanded storage, provided that the source and destination pages are both valid. Some details about the means for control-program support of MOVE PAGE are not provided. The ESA/370 version of MOVE PAGE is now called move-page facility 1.
- The *Processor Resource/Systems Manager™ (PR/SM™)* feature provides support for multiple preferred guests under VM/XA and provides the logically partitioned (LPAR) mode, with the latter providing flexible partitioning of processor resources among multiple logical partitions. Certain aspects of the LPAR use of PR/SM are described in the publication *IBM ES/3090 Processor Complex Processor Resource/Systems Manager Planning Guide*, GA22-7123.
- The COMPARE UNTIL SUBSTRING EQUAL instruction provides improved performance of the compression of IMS log data sets and can be useful in other programs also.

System Program

z/Architecture is designed to be used with a control program that coordinates the use of system resources and executes all I/O instructions, handles exceptional conditions, and supervises scheduling and execution of multiple programs.

Compatibility

Compatibility among z/Architecture Systems

Although systems operating as defined by z/Architecture may differ in implementation and physical capabilities, logically they are upward and downward compatible. Compatibility provides for simplicity in education, availability of system backup, and ease in system growth. Specifically, any program written for z/Architecture gives identical results on any z/Architecture implementation, provided that the program:

1. Is not time-dependent.
2. Does not depend on system facilities (such as storage capacity, I/O equipment, or optional facilities) being present when the facilities are not included in the configuration.
3. Does not depend on system facilities being absent when the facilities are included in the configuration. For example, the program must not depend on interruptions caused by the use of operation codes or command codes that are not installed in some models. Also, it must not use or depend on fields associated with uninstalled facilities. For example, data should not be placed in an area used by another model for fixed-logout information. Similarly, the program must not use or depend on unassigned fields in machine formats (control registers, instruction formats, etc.) that are not explicitly made available for program use.
4. Does not depend on results or functions that are defined to be unpredictable or model-dependent or are identified as undefined. This includes the requirement that the program should not depend on the assignment of device numbers and CPU addresses.
5. Does not depend on results or functions that are defined in the functional-characteristics publication for a particular model to be deviations from the architecture.
6. Takes into account any changes made to the architecture that are identified as affecting compatibility.

Compatibility between z/Architecture and ESA/390

Control-Program Compatibility

Control programs written for ESA/390 cannot be directly transferred to systems operating as defined by z/Architecture. This is because the general-register and control-register sizes, PSW size, assigned

storage locations, and dynamic address translation are changed.

Problem-State Compatibility

A high degree of compatibility exists at the problem-state level in going forward from ESA/390 to z/Architecture. Because the majority of a user's applications are written for the problem state, this problem-state compatibility is useful in many installations.

A problem-state program written for ESA/390 operates with z/Architecture, provided that the program:

1. Complies with the limitations described in "Compatibility among z/Architecture Systems".
2. Is not dependent on privileged facilities which are unavailable on the system.

Programming Note: This publication assigns meanings to various operation codes, to bit positions in instructions, channel-command words, registers, and table entries, and to fixed locations in the low 512 bytes and bytes 4096-8191 of storage. Unless specifically noted, the remaining operation codes, bit positions, and low-storage locations are reserved for future assignment to new facilities and other extensions of the architecture.

To ensure that existing programs operate if and when such new facilities are installed, programs should not depend on an indication of an exception as a result of invalid values that are currently defined as being checked. If a value must be placed in unassigned positions that are not checked, the program should enter zeros. When the machine provides a code or field, the program should take into account that new codes and bits may be assigned in the future. The program should not use unassigned low-storage locations for keeping information since these locations may be assigned in the future in such a way that the machine causes the contents of the locations to be changed.

Availability

Availability is the capability of a system to accept and successfully process an individual job. Systems operating in accordance with z/Architecture permit substantial availability by (1) allowing a large number and broad range of jobs to be processed concurrently, thus making the system readily accessible to any particular job, and (2) limiting the effect of an error and identifying more precisely its cause, with the result that the number of jobs affected by errors is minimized and the correction of the errors facilitated.

Several design aspects make this possible.

- A program is checked for the correctness of instructions and data as the program is executed, and program errors are indicated separate from equipment errors. Such checking and reporting assists in locating failures and isolating effects.
- The protection facilities, in conjunction with dynamic address translation and the separation of programs and data in different address spaces, permit the protection of the contents of storage from destruction or misuse caused by erroneous or unauthorized storing or fetching by a program. This provides increased security for the user, thus permitting applications with different security requirements to be processed concurrently with other applications.
- Dynamic address translation allows isolation of one application from another, still permitting them to share common resources. Also, it permits the implementation of virtual machines, which may be used in the design and testing of new versions of operating systems along with the concurrent processing of application programs. Additionally, it provides for the concurrent operation of incompatible operating systems.
- The use of access registers allows programs, data, and different collections of data to reside in different address spaces, and this further reduces the likelihood that a store using an incorrect address will produce either erroneous results or a system-wide failure.
- Multiprocessing and the channel subsystem permit better use of storage and processing capabilities, more direct communication between CPUs, and duplication of resources, thus aiding in the continuation of system operation in the event of machine failures.
- MONITOR CALL, program-event recording, and the timing facilities permit the testing and debugging of programs without manual intervention and with little effect on the concurrent processing of other programs.
- On most models, error checking and correction (ECC) in main storage, CPU retry, and command

retry provide for circumventing intermittent equipment malfunctions, thus reducing the number of equipment failures.

- An enhanced machine-check-handling mechanism provides model-independent fault isolation, which reduces the number of programs impacted by uncorrected errors. Additionally, it provides model-independent recording of machine-status information. This leads to greater machine-check-handling compatibility between models and improves the capability for loading and operating a program on a different model when a system failure occurs.
- A small number of manual controls are required for basic system operation, permitting most operator-system interaction to take place *via* a unit operating as an I/O device and thus reducing the possibility of operator errors.
- The logical partitions made available by the PR/SM feature allow continued reliable production operations in one or more partitions while new programming systems are tested in other partitions. This is an advancement in particular for non-VM installations.
- The operational extensions and channel-subsystem-call facility of ESA/390 and z/Architecture improve the ability to continue execution of application programs in the presence of system incidents and the ability to make configuration changes with less disruption to operations.

Chapter 2. Organization

Main Storage 2-2
Expanded Storage 2-2
CPU 2-2
CPU Types 2-3
Multithreading 2-3
PSW 2-3
General Registers 2-4
Floating-Point Registers 2-4
Floating-Point-Control Register 2-5
Vector Registers 2-5
Control Registers 2-6
Access Registers 2-6
Cryptographic Facility 2-7
External Time Reference 2-7
I/O 2-7
Channel Subsystem 2-7
Channel Paths 2-7
I/O Devices and Control Units 2-8
Operator Facilities 2-8

Logically, a system consists of main storage, one or more central processing units (CPUs), operator facilities, a channel subsystem, and I/O devices. I/O devices are attached to the channel subsystem through control units. The connection between the channel subsystem and a control unit is called a channel path.

A channel path employs either a parallel-transmission protocol or a serial-transmission protocol and, accordingly, is called either a parallel or a serial channel path. A serial channel path may connect to a control unit through a dynamic switch that is capable of providing different internal connections between the ports of the switch.

Expanded storage may also be available in the system, a cryptographic unit may be included in a CPU, and an external time reference (ETR) may be connected to the system.

The physical identity of the above functions may vary among implementations, called “models”. Figure 2-1 depicts the logical structure of a two-CPU multiprocessing system that includes expanded storage and a cryptographic unit and that is connected to an ETR.

Specific processors may differ in their internal characteristics, the installed facilities, the number of subchannels, channel paths, and control units which can be attached to the channel subsystem, the size of main and expanded storage, and the representation of the operator facilities.

A system viewed without regard to its I/O devices is referred to as a configuration. All of the physical equipment, whether in the configuration or not, is referred to as the installation.

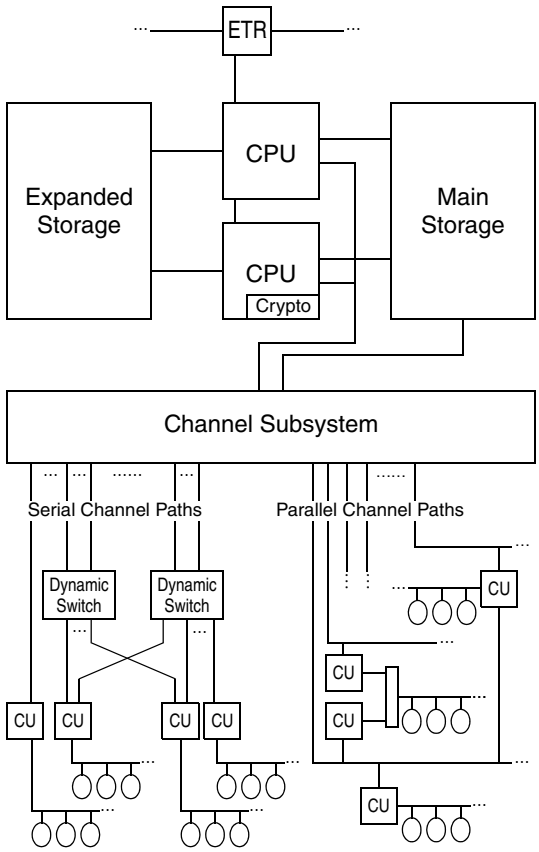


Figure 2-1. Logical Structure of a z/Architecture System with Two CPUs

Model-dependent reconfiguration controls may be provided to change the amount of main and expanded storage and the number of CPUs and channel paths in the configuration. In some instances, the reconfiguration controls may be used to partition a single configuration into multiple config-

© Copyright IBM Corp. 2000, 2017

urations. Each of the configurations so reconfigured has the same structure, that is, main and expanded storage, one or more CPUs, and one or more subchannels and channel paths in the channel subsystem.

Each configuration is isolated in that the main and expanded storage in one configuration is not directly addressable by the CPUs and the channel subsystem of another configuration. It is, however, possible for one configuration to communicate with another by means of shared I/O devices or a channel-to-channel adapter. At any one time, the storage, CPUs, subchannels, and channel paths connected together in a system are referred to as being in the configuration. Each CPU, subchannel, channel path, main-storage location, and expanded-storage location can be in only one configuration at a time.

Main Storage

Main storage, which is directly addressable, provides for high-speed processing of data by the CPUs and the channel subsystem. Both data and programs must be loaded into main storage from input devices before they can be processed. The amount of main storage available in the system depends on the model, and, depending on the model, the amount in the configuration may be under control of model-dependent configuration controls. The storage is available in multiples of 4 K-byte blocks. At any instant, the channel subsystem and all CPUs in the configuration have access to the same blocks of storage and refer to a particular block of main-storage locations by using the same absolute address.

Main storage may include a faster-access buffer storage, sometimes called a cache. Each CPU may have an associated cache. The effects, except on performance, of the physical construction and the use of distinct storage media are not observable by the program.

Expanded Storage

Expanded storage may be available on some models. Expanded storage, when available, can be accessed by all CPUs in the configuration by means of instructions that transfer 4 K-byte blocks of data from expanded storage to main storage or from main storage to expanded storage. These instructions are the PAGE IN and PAGE OUT instructions, described in “PAGE IN” on page 10-72 and “PAGE OUT” on page 10-73.

Each 4 K-byte block of expanded storage is addressed by means of a 32-bit unsigned binary integer called an expanded-storage block number.

CPU

The central processing unit (CPU) is the controlling center of the system. It contains the sequencing and processing facilities for instruction execution, interruption action, timing functions, initial program loading, and other machine-related functions.

The physical implementation of the CPU may differ among models, but the logical function remains the same. The result of executing an instruction is the same for each model, providing that the program complies with the compatibility rules.

The CPU, in executing instructions, can process binary integers and floating-point numbers (binary, decimal, and hexadecimal) of fixed length, decimal integers of variable length, and logical information of either fixed or variable length. Processing may be in parallel or in series; the width of the processing elements, the multiplicity of the shifting paths, and the degree of simultaneity in performing the different types of arithmetic differ from one model of CPU to another without affecting the logical results.

Instructions which the CPU executes fall into eight classes: general, decimal, floating-point-support (FPS), binary-floating-point (BFP), decimal-floating-point (DFP), hexadecimal-floating-point (HFP), control, and I/O instructions. The general instructions are used in performing binary-integer-arithmetic operations and logical, branching, and other nonarithmetic operations. The decimal instructions operate on data in the decimal format. The BFP, DFP, and HFP instructions operate on data in the BFP, DFP, and HFP formats, respectively, while the FPS instructions operate on floating-point data independent of the format or convert from one format to another. The privileged control instructions and the I/O instructions can be executed only when the CPU is in the supervisor state; the semiprivileged control instructions can be executed in the problem state, subject to the appropriate authorization mechanisms.

The CPU provides registers which are available to programs but do not have addressable representations in main storage, as follows:

- Access registers
- Breaking-event-address register
- Clock comparator
- Control registers
- CPU timer
- Current program-status word (PSW)
- Floating-point-control register
- Floating-point registers
- General registers
- Guarded-storage-control registers
- Prefix register
- TOD-programmable register
- Vector registers

This set of registers is sometimes referred to as the CPU's *architected register context*.

Each CPU in an installation provides access to a time-of-day (TOD) clock and epoch index, which are shared by all CPUs in the installation. The instruction operation code determines which type of register is to be used in an operation. See Figure 2-2 on page 2-5 for the format of the control, access, general, and floating-point registers.

CPU Types

Each CPU has a type attribute that indicates whether it provides the full complement of functions and facilities (called a general CPU), or whether it is intended to process specific types of workloads. A primary CPU is either a general CPU or a CPU having the same type as the CPU started following the last IPL operation (the IPL CPU). A secondary CPU is any CPU other than a general CPU, and whose CPU type differs from that of the IPL CPU.

Programming Note: CPUs intended to process specific types of workloads include those used by the integrated facility for Linux (IFL), the internal coupling facility (ICF), the IBM zEnterprise Application-Assist Processor (zAAP), and the IBM z Integrated Information Processor (zIIP).

Multithreading

When the multithreading (MT) facility is installed, it may be enabled by execution of the set-multithreading SIGP order (see "Set Multithreading" on page 4-91 for details). The following terms are applicable when the MT facility is installed:

Core: When the multithreading facility is enabled, the following applies:

- The number of CPUs in the configuration is increased by a multiple, the value of which is determined by a program-specified maximum thread identification (thread ID, or simply TID). The number of CPUs in a core is one more than the program-specified maximum thread identification.
- A number of CPUs corresponding to this multiple are grouped into a *core*.
- Each core has the same number of CPUs. Each CPU within a core is of the same CPU type; however, based on the model and CPU type, some CPUs within a core may not be operational.

When the multithreading facility is not installed, or the facility is installed but not enabled, a core comprises a single CPU, and the term CPU is generally used in favor of the term core.

Thread: When the multithreading facility is installed, a *thread* is synonymous with a CPU that is a member of a core. When the multithreading facility is not installed, or when the facility is installed but not enabled, the term thread is not applicable.

A control program must explicitly enable multithreading in order for it to be usable, as described in "Set Multithreading" on page 4-91. However, an application program is generally unaware of whether multithreading has been enabled.

When multithreading is enabled, the CPU addresses of all CPUs in the configuration are adjusted to include a core identification (or core ID) in the leftmost bits of the address and a thread identification (thread ID, or TID) in the rightmost bits of the address. See "CPU-Address Identification" on page 4-83 for details on the core-ID and thread-ID fields of the CPU address.

CPUs within a core may share certain hardware facilities such as execution units or lower-level caches, thus execution within one CPU of a core may affect the performance of other CPUs in the core.

PSW

The program-status word (PSW) includes the instruction address, condition code, and other information

used to control instruction sequencing and to determine the state of the CPU. The active or controlling PSW is called the current PSW. It governs the program currently being executed.

The CPU has an interruption capability, which permits the CPU to switch rapidly to another program in response to exceptional conditions and external stimuli. When an interruption occurs, the CPU places the current PSW in an assigned storage location, called the old-PSW location, for the particular class of interruption. The CPU fetches a new PSW from a second assigned storage location. This new PSW determines the next program to be executed. When it has finished processing the interruption, the program handling the interruption may reload the old PSW, making it again the current PSW, so that the interrupted program can continue.

There are six classes of interruption: external, I/O, machine check, program, restart, and supervisor call. Each class has a distinct pair of old-PSW and new-PSW locations permanently assigned in real storage.

General Registers

Instructions may designate information in one or more of 16 general registers. The general registers may be used as base-address registers and index registers in address arithmetic and as accumulators in general arithmetic and logical operations. Each register contains 64 bit positions. The general registers are identified by the numbers 0-15 and are designated by a four-bit R field in an instruction. Some instructions provide for addressing multiple general registers by having several R fields. For some instructions, the use of a specific general register is implied rather than explicitly designated by an R field of the instruction.

For some operations, either bits 32-63 or bits 0-63 of two adjacent general registers are coupled, providing a 64-bit or 128-bit format, respectively. In these operations, the program must designate an even-numbered register, which contains the leftmost (high-order) 32 or 64 bits. The next higher-numbered register contains the rightmost (low-order) 32 or 64 bits.

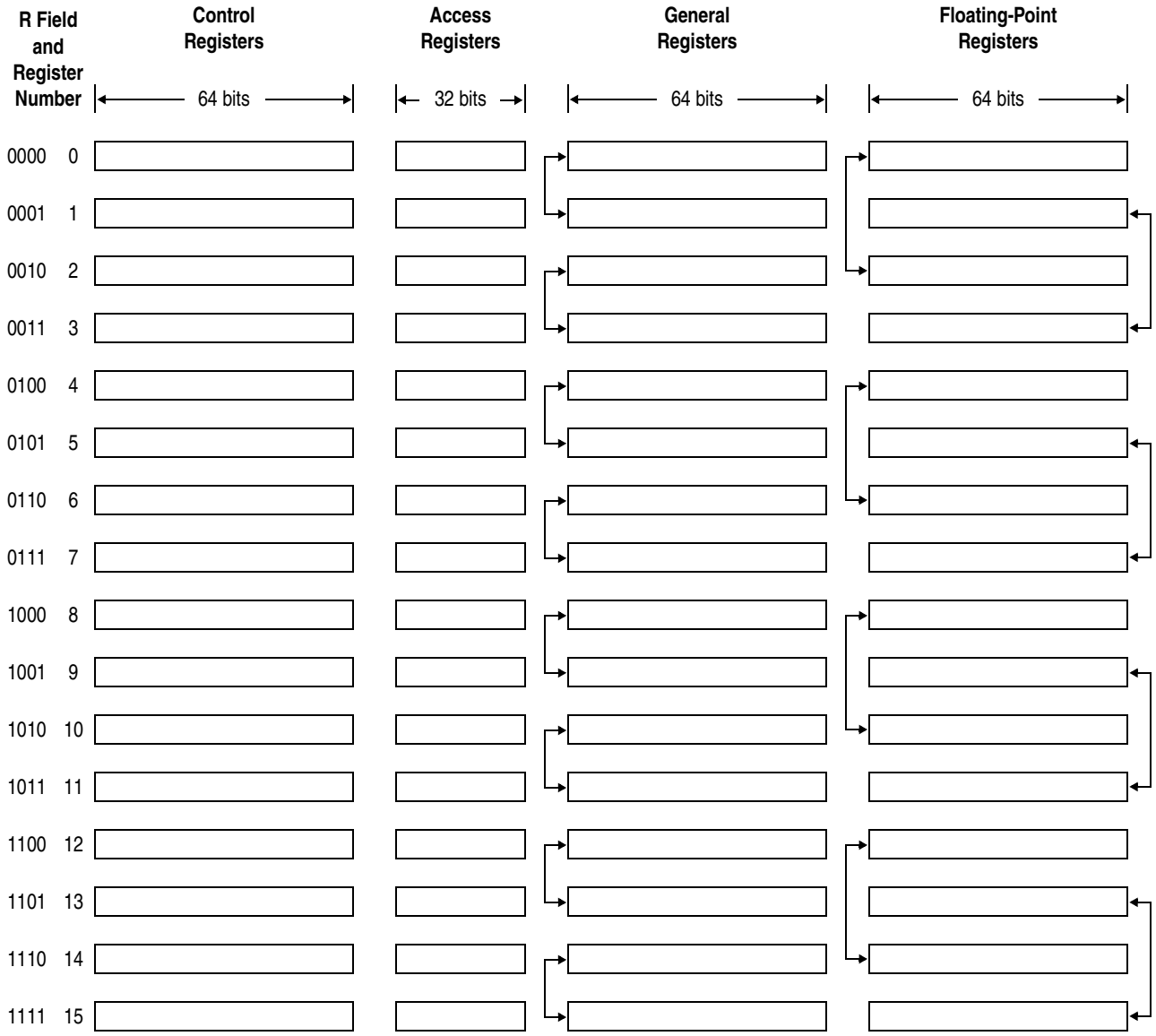
In addition to their use as accumulators in general arithmetic and logical operations, 15 of the 16 general registers are also used as base-address and index registers in address generation. In these cases, the registers are designated by a four-bit B field or X field in an instruction. A value of zero in the B or X field specifies that no base or index is to be applied, and, thus, general register 0 cannot be designated as containing a base address or index.

While in the ESA/390 architectural mode, the contents of bit positions 0-31 of the 64-bit general registers of all CPUs in the configuration cannot be altered by any instruction.

In the ESA/390-compatibility mode, unless stated otherwise, it is unpredictable whether instructions that are unique to the z/Architecture architectural mode result in an operation exception or in execution as defined in z/Architecture. Therefore, it is possible that bits 0-31 of a 64-bit general register may be altered by a z/Architecture instruction that is executed in the ESA/390-compatibility mode.

Floating-Point Registers

All floating-point instructions (FPS, BFP, DFP, and HFP) use the same set of floating-point registers. The CPU has 16 floating-point registers. The floating-point registers are identified by the numbers 0-15 and are designated by a four-bit R field in floating-point instructions. Each floating-point register is 64 bits long and can contain either a short (32-bit) or a long (64-bit) floating-point operand. As shown in Figure 2-2 on page 2-5, pairs of floating-point registers can be used for extended (128-bit) operands. Each of the eight pairs is referred to by the number of the lower-numbered register of the pair.



Note: The arrows indicate that the two registers may be coupled as a double-register pair, designated by specifying the lower-numbered register in the R field. For example, the floating-point register pair 13 and 15 is designated by 1101 binary in the R field.

Figure 2-2. Control, Access, General, and Floating-Point Registers

Floating-Point-Control Register

The floating-point-control (FPC) register is a 32-bit register that contains mask bits, flag bits, a data-exception code, and rounding-mode bits. The FPC register is described in the section "Floating-Point-Control (FPC) Register" on page 9-9.

Vector Registers

All vector instructions use the same set of 32 vector registers identified by the numbers 0-31 and are designated by the concatenation of a single register number extension bit with a four-bit field. Each vector register is 128 bits long and can contain from one to sixteen equal sized elements. When there are multi-

ple elements in the register the elements are indexed from left to right starting at element zero.

The floating-point registers overlay the vector registers as shown in Figure 2-3 on page 2-6. Bits 0-63 of vector registers 0-15 correspond to floating-point registers 0-15. Whenever a floating-point instruction or floating point support instruction writes to a floating point register, or a floating point instruction that reads a register pair reads from floating-point registers, bits 64-127 of the corresponding vector register are unpredictable. Any use of vector registers 0-15 by a vector instruction, except for VECTOR LOAD ELEMENT specifying an element not in bits 0-63 of the vector register will cause the data in the floating point register to be overwritten as well.

0	FPR 0 or VR 0 (bits 0-63)	VR 0 (bits 64-127)
1	FPR 1 or VR 1 (bits 0-63)	VR 1 (bits 64-127)
2	FPR 2 or VR 2 (bits 0-63)	VR 2 (bits 64-127)
3	FPR 3 or VR 3 (bits 0-63)	VR 3 (bits 64-127)
4	FPR 4 or VR 4 (bits 0-63)	VR 4 (bits 64-127)
5	FPR 5 or VR 5 (bits 0-63)	VR 5 (bits 64-127)
6	FPR 6 or VR 6 (bits 0-63)	VR 6 (bits 64-127)
7	FPR 7 or VR 7 (bits 0-63)	VR 7 (bits 64-127)
8	FPR 8 or VR 8 (bits 0-63)	VR 8 (bits 64-127)
9	FPR 9 or VR 9 (bits 0-63)	VR 9 (bits 64-127)
10	FPR 10 or VR 10 (bits 0-63)	VR 10 (bits 64-127)
11	FPR 11 or VR 11 (bits 0-63)	VR 11 (bits 64-127)
12	FPR 12 or VR 12 (bits 0-63)	VR 12 (bits 64-127)
13	FPR 13 or VR 13 (bits 0-63)	VR 13 (bits 64-127)
14	FPR 14 or VR 14 (bits 0-63)	VR 14 (bits 64-127)
15	FPR 15 or VR 15 (bits 0-63)	VR 15 (bits 64-127)
16	VR 16	
17	VR 17	
18	VR 18	
19	VR 19	
20	VR 20	
21	VR 21	
22	VR 22	
23	VR 23	
24	VR 24	
25	VR 25	
26	VR 26	
27	VR 27	
28	VR 28	
29	VR 29	
30	VR 30	
31	VR 31	

Figure 2-3. Vector and Floating Point Registers

Control Registers

The CPU has 16 control registers, each having 64 bit positions. The bit positions in the registers are assigned to particular facilities in the system, such as program-event recording, and are used either to specify that an operation can take place or to furnish special information required by the facility.

The control registers are identified by the numbers 0-15 and are designated by four-bit R fields in the instructions LOAD CONTROL and STORE CONTROL. Multiple control registers can be addressed by these instructions.

While in the ESA/390 architectural mode or the ESA/390-compatibility mode, the contents of bit positions 0-31 of the 64-bit control registers of all CPUs in the configuration cannot be altered by any instruction. If a configuration returns to the ESA/390-compatibility mode from the z/Architecture architectural mode, the contents of bit positions 0-31 of the 64-bit control registers are all set to zeros.

Access Registers

The CPU has 16 access registers numbered 0-15. An access register consists of 32 bit positions containing an indirect specification (not described here in detail) of an address-space-control element. An address-space-control element is a parameter used by the dynamic-address-translation (DAT) mechanism to translate references to a corresponding address space. When the CPU is in a mode called the access-register mode (controlled by bits in the PSW), an instruction B field, used to specify a logical address for a storage-operand reference, designates an access register, and the address-space-control element specified by the access register is used by DAT for the reference being made. For some instructions, an R field is used instead of a B field. Instructions are provided for loading and storing the contents of the access registers and for moving the contents of one access register to another.

Each of access registers 1-15 can designate any address space, including the current instruction space (the primary address space). Access register 0 always designates the current instruction space. When one of access registers 1-15 is used to designate an address space, the CPU determines which address space is designated by translating the con-

tents of the access register. When access register 0 is used to designate an address space, the CPU treats the access register as designating the current instruction space, and it does not examine the actual contents of the access register. Therefore, the 16 access registers can designate, at any one time, the current instruction space and a maximum of 15 other spaces.

Cryptographic Facility

Depending on the model, an integrated cryptographic facility may be provided as an extension of the CPU. When the cryptographic facility is provided on a CPU, it functions as an integral part of that CPU. A summary of the benefits of the cryptographic facility is given on page 1-31; the facility is otherwise not described.

External Time Reference

Depending on the model, an external time reference (ETR) may be connected to the configuration. A summary of the benefits of the ETR is given on page 1-31; the facility is otherwise not described.

I/O

Input/output (I/O) operations involve the transfer of information between main storage and an I/O device. I/O devices and their control units attach to the channel subsystem, which controls this data transfer.

Channel Subsystem

The channel subsystem directs the flow of information between I/O devices and main storage. It relieves CPUs of the task of communicating directly with I/O devices and permits data processing to proceed concurrently with I/O processing. The channel subsystem uses one or more channel paths as the communication link in managing the flow of information to or from I/O devices. As part of I/O processing, the channel subsystem also performs the path-management function of testing for channel-path availability, selecting an available channel path, and initiating execution of the operation with the I/O device. Within the channel subsystem are subchannels.

One subchannel is provided for and dedicated to each I/O device accessible to the channel subsystem. Each subchannel contains storage for information concerning the associated I/O device and its attachment to the channel subsystem. The subchannel also provides storage for information concerning I/O operations and other functions involving the associated I/O device. Information contained in the subchannel can be accessed by CPUs using I/O instructions as well as by the channel subsystem and serves as the means of communication between any CPU and the channel subsystem concerning the associated I/O device. The actual number of subchannels provided depends on the model and the configuration; the maximum number of subchannels is 65,536.

Channel Paths

I/O devices are attached through control units to the channel subsystem via channel paths. Control units may be attached to the channel subsystem via more than one channel path, and an I/O device may be attached to more than one control unit. In all, an individual I/O device may be accessible to a channel subsystem by as many as eight different channel paths, depending on the model and the configuration. The total number of channel paths provided by a channel subsystem depends on the model and the configuration; the maximum number of channel paths is 256.

A channel path can use one of three types of communication links:

- System/360 and System/370 I/O interface, called the parallel-I/O interface; the channel path is called a parallel channel path
- ESCON I/O interface, called a serial-I/O interface; the channel path is called a serial channel path
- FICON I/O interface, also called a serial-I/O interface; the channel path again is called a serial channel path

Each parallel-I/O interface consists of a number of electrical signal lines between the channel subsystem and one or more control units. Eight control units can share a single parallel-I/O interface. Up to 256

I/O devices can be addressed on a single parallel-I/O interface. The parallel-I/O interface is described in the publication *IBM System/360 and System/370 I/O Interface Channel to Control Unit Original Equipment Manufacturers' Information*, GA22-6974.

Each serial-I/O interface consists of two optical-fiber conductors between any two of a channel subsystem, a dynamic switch, and a control unit. A dynamic switch can be connected by means of multiple serial-I/O interfaces to either the same or different channel subsystems and to multiple control units. The number of control units which can be connected on one channel path depends on the channel-subsystem and dynamic-switch capabilities. Up to 256 devices can be attached to each control unit that uses the serial-I/O interface, depending on the control unit. The ESCON I/O interface is described in the publication *ESA/390 ESCON I/O Interface*, SA22-7202. The FICON I/O interface is described in the ANSI standards document *Fibre Channel - Single-Byte Command Code Sets-2 (FC-SB-2)*.

I/O Devices and Control Units

I/O devices include such equipment as printers, magnetic-tape units, direct-access-storage devices, displays, keyboards, communications controllers, teleprocessing devices, and sensor-based equipment. Many I/O devices function with an external medium, such as paper or magnetic tape. Other I/O devices handle only electrical signals, such as those found in displays and communications networks. In all cases, I/O-device operation is regulated by a control unit that provides the logical and buffering capabilities necessary to operate the associated I/O device. From the programming point of view, most control-unit functions merge with I/O-device functions. The control-unit function may be housed with the I/O device or in the CPU, or a separate control unit may be used.

Operator Facilities

The operator facilities provide the functions necessary for operator control of the machine. Associated with the operator facilities may be an operator-console device, which may also be used as an I/O device for communicating with the program.

The main functions provided by the operator facilities include resetting, clearing, initial program loading, start, stop, alter, and display.

Chapter 3. Storage

Storage Addressing 3-2
Information Formats. 3-3
Integral Boundaries 3-3
Address Types and Formats 3-4
Address Types. 3-4
Absolute Address. 3-4
Real Address 3-4
Virtual Address 3-5
Primary Virtual Address 3-5
Secondary Virtual Address 3-5
AR-Specified Virtual Address 3-5
Home Virtual Address 3-5
Logical Address 3-5
Instruction Address 3-5
Effective Address. 3-5
Address Size and Wraparound 3-6
Address Wraparound. 3-7
Storage Key. 3-9
Protection 3-10
Key-Controlled Protection 3-11
Storage-Protection-Override Control 3-12
Fetch-Protection-Override Control 3-13
Access-List-Controlled Protection 3-13
DAT Protection 3-13
Low-Address Protection. 3-14
Instruction-Execution Protection 3-14
Suppression on Protection. 3-15
Reference Recording. 3-19
Change Recording. 3-20
Prefixing. 3-21
Prefixing in the z/Architecture Architectural Mode 3-22
Prefixing in the ESA/390-Compatibility Mode 3-22
Common Prefixing Attributes. 3-22
Address Spaces 3-23
Changing to Different Address Spaces . . . 3-23
Address-Space Number 3-23
ASN-Second-Table-Entry Sequence Number 3-24
ASN-Second-Table-Entry Instance Number and ASN Reuse. 3-25
ASN Translation 3-30
ASN-Translation Controls 3-30
Control Register 14 3-30
ASN-Translation Tables. 3-31
ASN-First-Table Entries. 3-31
ASN-Second-Table Entries 3-31
ASN-Translation Process 3-33
ASN-First-Table Lookup 3-33
ASN-Second-Table Lookup 3-34
Recognition of Exceptions during ASN Translation 3-35
ASN Authorization. 3-35
ASN-Authorization Controls 3-35
Control Register 4 3-35
ASN-Second-Table Entry 3-35
Authority-Table Entries 3-36
ASN-Authorization Process 3-36
Authority-Table Lookup. 3-36
Recognition of Exceptions during ASN Authorization. 3-38
Dynamic Address Translation. 3-38
Translation Control 3-40
Translation Modes. 3-40
Control Register 0 3-41
Control Register 1 3-42
Control Register 7 3-44
Control Register 13 3-45
Translation Tables 3-45
Region-Table Entries 3-46
Segment-Table Entries 3-49
Page-Table Entries 3-51
Translation Process 3-52
Inspection of Real-Space Control 3-54
Inspection of Designation-Type Control . . . 3-54
Lookup in a Table Designated by an Address-Space-Control Element 3-57
Lookup in a Table Designated by a Region-Table Entry 3-59
Page-Table Lookup. 3-61
Formation of the Real and Absolute Addresses. 3-62
Recognition of Exceptions during Translation. . 3-62
Translation-Lookaside Buffer 3-62
TLB Structure 3-63
Formation of TLB Entries 3-63
Use of TLB Entries 3-65
Modification of Translation Tables 3-67
Address Summary 3-70
Addresses Translated. 3-70
Handling of Addresses 3-71
Assigned Storage Locations. 3-73
Assigned Storage Locations in the z/Architecture Architectural Mode. 3-73
Assigned Storage Locations in the ESA/390-Compatibility Mode 3-83

© Copyright IBM Corp. 2000, 2017

This chapter discusses the representation of information in main storage, as well as addressing, protection, and reference and change recording. The aspects of addressing which are covered include the format of addresses, the concept of address spaces, the various types of addresses, and the manner in which one type of address is translated to another type of address.

A list of permanently assigned storage locations appears at the end of the chapter.

Main storage provides the system with directly addressable fast-access storage of data. Both data and programs must be loaded into main storage (from input devices) before they can be processed.

Main storage may include one or more smaller faster-access buffer storages, sometimes called caches. A cache is usually physically associated with a CPU or an I/O processor. The effects, except on performance, of the physical construction and use of distinct storage media are generally not observable by the program.

Separate caches may be maintained for instructions and for data operands. Information within a cache is maintained in contiguous bytes on an integral boundary called a cache block or cache line (or line, for short). A model may provide the EXTRACT CPU ATTRIBUTE instruction which returns various cache attributes such as line size, number of cache levels, and set associativity. A model may also provide any or all of the PREFETCH DATA, PREFETCH DATA RELATIVE LONG, and NEXT INSTRUCTION ACCESS INTENT instructions which affect the prefetching of storage into the data cache, the releasing of data from the cache, or indicating access intent of storage operands.

Fetching and storing of data by a CPU are not normally affected by any concurrent channel-subsystem activity or by a concurrent reference to the same storage location by another CPU. When concurrent requests to a main-storage location occur, access normally is granted in a sequence determined by the system. If a reference changes the contents of the location, any subsequent storage fetches obtain the new contents. However, when the transactional-execution facility is installed, a transaction executing on one CPU may be aborted due to conflicting fetches and stores by another CPU or by the channel subsystem.

Main storage may be volatile or nonvolatile. If it is volatile, the contents of main storage are not preserved when power is turned off. If it is nonvolatile, turning power off and then back on does not affect the contents of main storage, provided all CPUs are in the stopped state and no references are made to main storage when power is being turned off. In both types of main storage, the contents of storage keys are not necessarily preserved when the power for main storage is turned off.

Note: Because most references in this publication apply to virtual storage, the abbreviated term “storage” is often used in place of “virtual storage.” The term “storage” may also be used in place of “main storage,” “absolute storage,” or “real storage” when the meaning is clear. The terms “main storage” and “absolute storage” are used to describe storage which is addressable by means of an absolute address. The terms describe fast-access storage, as opposed to auxiliary storage, such as that provided by direct-access storage devices. “Real storage” is synonymous with “absolute storage” except for the effects of prefixing.

Programming Note: On models that implement separate caches for instructions and data operands, a significant delay may be experienced if the program stores into a cache line from which instructions are subsequently fetched, regardless of whether the store alters the instructions that are subsequently fetched.

Storage Addressing

Storage is viewed as a long horizontal string of bits. For most operations, accesses to storage proceed in a left-to-right sequence. The string of bits is subdivided into units of eight bits. An eight-bit unit is called a byte, which is the basic building block of all information formats.

Each byte location in storage is identified by a unique nonnegative integer, which is the address of that byte location or, simply, the byte address. Adjacent byte locations have consecutive addresses, starting with 0 on the left and proceeding in a left-to-right sequence. Addresses are unsigned binary integers and are 24, 31, or 64 bits. Addresses are described in “Address Size and Wraparound” on page 3-6.

Information Formats

Information is transmitted between storage and a CPU or the channel subsystem one byte, or a group of bytes, at a time. Unless otherwise specified, a group of bytes in storage is addressed by the leftmost byte of the group. The number of bytes in the group is either implied or explicitly specified by the operation to be performed. When used in a CPU operation, a group of bytes is called a field.

Within each group of bytes, bits are numbered in a left-to-right sequence. The leftmost bits are sometimes referred to as the "high-order" bits and the rightmost bits as the "low-order" bits. Bit numbers are not storage addresses, however. Only bytes can be addressed. To operate on individual bits of a byte in storage, it is necessary to access the entire byte.

The bits in a byte are numbered 0 through 7, from left to right.

The bits in an address may be numbered 8-31 or 40-63 for 24-bit addresses or 1-31 or 33-63 for 31-bit addresses; they are numbered 0-63 for 64-bit addresses. Within any other fixed-length format of multiple bytes, the bits making up the format are consecutively numbered starting from 0.

For purposes of error detection, and in some models for correction, one or more check bits may be transmitted with each byte or with a group of bytes. Such check bits are generated automatically by the machine and cannot be directly controlled by the program. References in this publication to the length of data fields and registers exclude mention of the associated check bits. All storage capacities are expressed in number of bytes.

When the length of a storage-operand field is implied by the operation code of an instruction, the field is said to have a fixed length, which can be one, two, four, eight, or sixteen bytes. Larger fields may be implied for some instructions.

When the length of a storage-operand field is not implied but is stated explicitly, the field is said to have a variable length. Variable-length operands can vary in length by increments of one byte.

When information is placed in storage, the contents of only those byte locations are replaced that are included in the designated field, even though the width of the physical path to storage may be greater than the length of the field being stored.

Integral Boundaries

Certain units of information must be on an integral boundary in storage. A boundary is called integral for a unit of information when its storage address is a multiple of the length of the unit in bytes. Special names are given to fields of 2, 4, 8, 16, and 32 bytes on an integral boundary. A halfword is a group of two consecutive bytes on a two-byte boundary and is the basic building block of instructions. A word is a group of four consecutive bytes on a four-byte boundary. A doubleword is a group of eight consecutive bytes on an eight-byte boundary. A quadword is a group of 16 consecutive bytes on a 16-byte boundary. An octoword is a group of 32 consecutive bytes on a 32-byte boundary. (See Figure 3-1 on page 3-4.)

When storage addresses designate halfwords, words, doublewords, quadwords, and octowords, the binary representation of the address contains one, two, three, four, or five rightmost zero bits, respectively.

Instructions must be on two-byte integral boundaries, and CCWs, IDAWs, MIDAWs, TIDAWs, and the storage operands of certain instructions must be on other integral boundaries. The storage operands of most instructions do not have boundary-alignment requirements.

Programming Notes:

1. For fixed-field-length operations with field lengths that are a power of 2, significant performance degradation is possible when storage operands are not positioned at addresses that are integral multiples of the operand length. To improve performance, frequently used storage operands should be aligned on integral boundaries.
2. Some vector instructions have an alignment-hint-control, whereby the program can notify the CPU of the alignment of an operand without the CPU

having to perform the address arithmetic to determine the alignment.

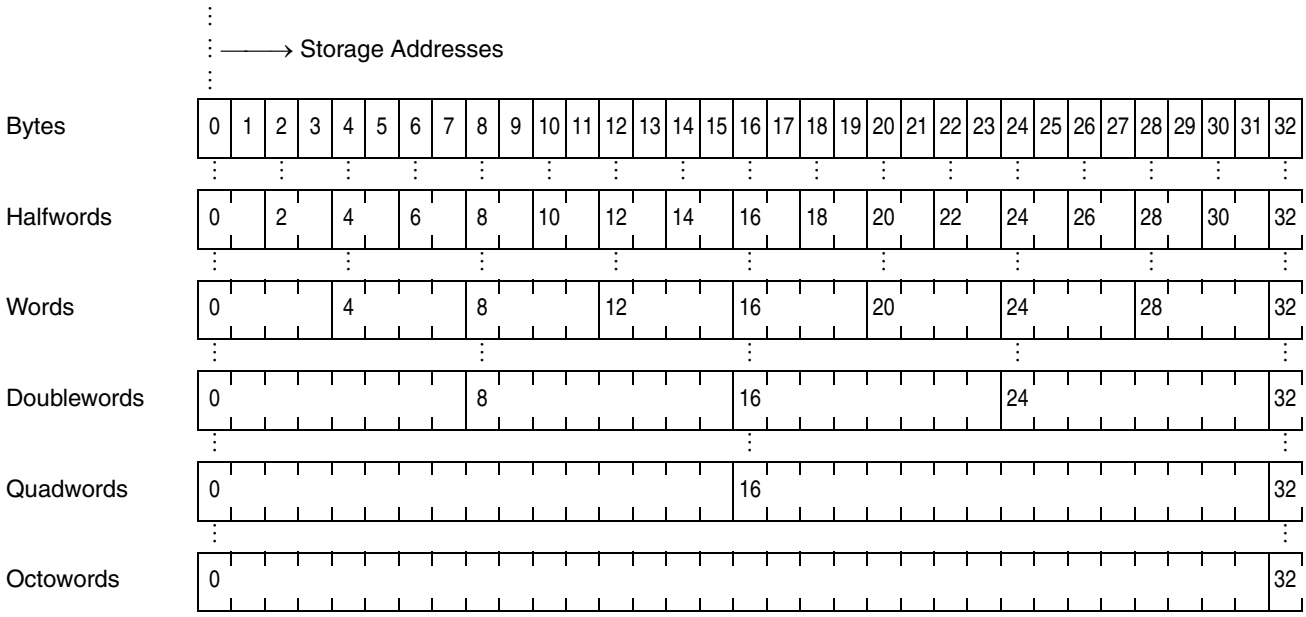


Figure 3-1. Integral Boundaries with Storage Addresses

Address Types and Formats

Address Types

For purposes of addressing main storage, three basic types of addresses are recognized: absolute, real, and virtual. The addresses are distinguished on the basis of the transformations that are applied to the address during a storage access. Address translation converts virtual to real, and prefixing converts real to absolute. In addition to the three basic address types, additional types are defined which are treated as one or another of the three basic types, depending on the instruction and the current mode.

In the ESA/390-compatibility mode, dynamic-address translation (DAT) is not supported, thus virtual addresses – including primary, secondary, AR-specified, and home virtual addresses – are not applicable. The ESA/390-compatibility mode may support absolute-storage address spaces as defined in Reference [12.] on page xxx by means of host access-register translation.

Absolute Address

An absolute address is the address assigned to a main-storage location. An absolute address is used for a storage access without any transformations performed on it.

The channel subsystem and all CPUs in the configuration refer to a shared main-storage location by using the same absolute address. Available main storage is usually assigned contiguous absolute addresses starting at 0, and the addresses are always assigned in complete 4 K-byte blocks on integral boundaries. An exception is recognized when an attempt is made to use an absolute address in a block which has not been assigned to physical locations. On some models, storage-reconfiguration controls may be provided which permit the operator to change the correspondence between absolute addresses and physical locations. However, at any one time, a physical location is not associated with more than one absolute address.

Storage consisting of byte locations sequenced according to their absolute addresses is referred to as absolute storage.

Real Address

A real address identifies a location in real storage. When a real address is used for an access to main storage, it is converted, by means of prefixing, to an absolute address.

At any instant there is one real-address to absolute-address mapping for each CPU in the configuration. When a real address is used by a CPU to access main storage, it is converted to an absolute address by prefixing. The particular transformation is defined by the value in the prefix register for the CPU.

Storage consisting of byte locations sequenced according to their real addresses is referred to as real storage.

Virtual Address

A virtual address identifies a location in virtual storage. When a virtual address is used for an access to main storage, it is translated by means of dynamic address translation, either (a) to a real address which is then further converted by prefixing to an absolute address, or (b) directly to an absolute address.

Primary Virtual Address

A primary virtual address is a virtual address which is to be translated by means of the primary address-space-control element. Logical addresses are treated as primary virtual addresses when in the primary-space mode. Instruction addresses are treated as primary virtual addresses when in the primary-space mode, secondary-space mode, or access-register mode. The first-operand address of MOVE TO PRIMARY and the second-operand address of MOVE TO SECONDARY are always treated as primary virtual addresses.

Secondary Virtual Address

A secondary virtual address is a virtual address which is to be translated by means of the secondary address-space-control element. Logical addresses are treated as secondary virtual addresses when in the secondary-space mode. The second-operand address of MOVE TO PRIMARY and the first-operand address of MOVE TO SECONDARY are always treated as secondary virtual addresses.

AR-Specified Virtual Address

An AR-specified virtual address is a virtual address which is to be translated by means of an access-register-specified address-space-control element. Logical addresses are treated as AR-specified addresses when in the access-register mode.

Home Virtual Address

A home virtual address is a virtual address which is to be translated by means of the home address-space-control element. Logical addresses and instruction addresses are treated as home virtual addresses when in the home-space mode.

Logical Address

Except where otherwise specified, the storage-operand addresses for most instructions are logical addresses. Logical addresses are treated as real addresses in the real mode, as primary virtual addresses in the primary-space mode, as secondary virtual addresses in the secondary-space mode, as AR-specified virtual addresses in the access-register mode, and as home virtual addresses in the home-space mode. Some instructions have storage-operand addresses or storage accesses associated with the instruction which do not follow the rules for logical addresses. In all such cases, the instruction definition contains a definition of the type of address.

Instruction Address

Addresses used to fetch instructions from storage are called instruction addresses. Instruction addresses are treated as real addresses in the real mode, as primary virtual addresses in the primary-space mode, secondary-space mode, or access-register mode, and as home virtual addresses in the home-space mode. The following are instruction addresses:

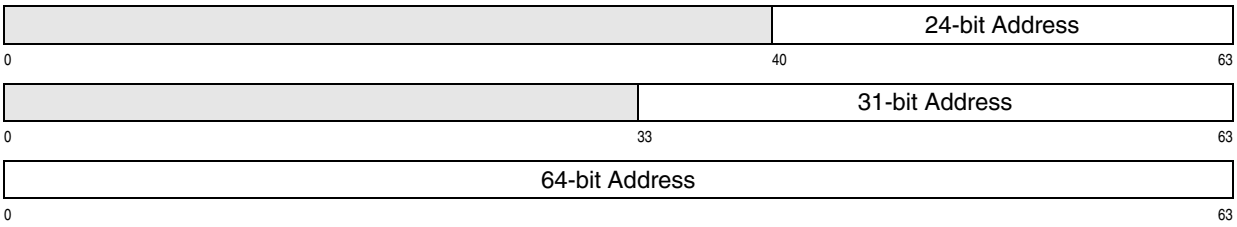
- The instruction address in the current PSW
- The target address of execute-type instructions (EXECUTE and EXECUTE RELATIVE LONG)
- The instruction address in the transaction-abort PSW
- The aborted-transaction instruction address (in the transaction diagnostic block)

Effective Address

In some situations, it is convenient to use the term “effective address.” An effective address is the address which exists before any transformation by dynamic address translation or prefixing is performed. An effective address may be specified directly in a register or may result from address arithmetic. Address arithmetic is the addition of the base and displacement or of the base, index, and displacement.

Address Size and Wraparound

An address size refers to the maximum number of significant bits that can represent an address. Three sizes of addresses are provided: 24-bit, 31-bit, and 64-bit. A 24-bit address can accommodate a maximum of 16,777,216 (16M) bytes; with a 31-bit address, 2,147,483,648 (2G) bytes can be addressed; and, with a 64-bit address, 18,446,744,073,709,551,616 (16E) bytes can be addressed. The bits of a 24-bit, 31-bit, or 64-bit address produced by address arithmetic under the control of the current addressing mode are numbered 40-63, 33-63, and 0-63, respectively, corresponding to the numbering of base-address and index bits in a general register, as shown below:



The byte and bit positions of a 64-bit address and corresponding power-of-two and numerical values are shown in Figure 3-2.

| Byte | 0 | | | | | | | | 1 | | | | | | | | 2 | | | | | | | | 3 | | | | | | | | 4 | | | | | | | | 5 | | | | | | | | 6 | | | | | | | | 7 | | | | | | | |
|---|---|---

bits 31 and 32 are both one, the CPU is in the 64-bit addressing mode, and 64-bit operand and instruction effective addresses are specified (see “Address Generation” on page 5-10).

The sizes of the real or absolute addresses used or yielded by the ASN-translation, ASN-authorization, PC-number-translation, and access-register-translation processes are always 31 bits regardless of the current addressing mode. Similarly, the sizes of the real or absolute addresses used or yielded by the DAT, stacking, unstacking, and tracing processes are always 64 bits.

The size of the data address in a CCW is under control of the CCW-format-control bit in the operation-request block (ORB) designated by a START SUBCHANNEL instruction. The CCWs with 24-bit and 31-bit addresses are called format-0 and format-1 CCWs, respectively. Format-0 and format-1 CCWs are described in “Basic I/O Functions” on page 15-1. Similarly, the size of the data address in an IDAW is under control of the IDAW-format-control bit in the ORB. The IDAWs with 31-bit and 64-bit addresses are called format-1 and format-2 IDAWs, respectively. MIDAWs contain 64-bit data addresses. IDAWs and MIDAWs are described in Chapter 15, “Basic I/O Functions.”

Address Wraparound

The CPU performs address generation when it forms an operand or instruction address or when it generates the address of a table entry from the appropriate table origin and index. It also performs address generation when it increments an address to access successive bytes of a field. Similarly, the channel subsystem performs address generation when it increments an address (1) to fetch a CCW, (2) to fetch an IDAW, (3) to fetch a MIDAW, (4) to transfer data, or (5) to compute the address of an I/O measurement block.

When, during the generation of the address, an address is obtained that exceeds the value allowed for the address size (2^{24} - 1, 2^{31} - 1, or 2^{64} - 1), one of the following two actions is taken:

1. The carry out of the high-order bit position of the address is ignored. This handling of an address of excessive size is called *wraparound*.
2. An interruption condition is recognized.

The effect of wraparound is to make an address space appear circular; that is, address 0 appears to follow the maximum allowable address. Address arithmetic and wraparound occur before transformation, if any, of the address by DAT or prefixing.

Addresses generated by the CPU that may be virtual addresses always wrap. Wraparound also occurs when the linkage-stack-entry address in control register 15 is decremented below 0 by PROGRAM RETURN. For CPU table entries that are addressed by real or absolute addresses, it is unpredictable whether the address wraps or an addressing exception is recognized.

For channel-program execution, when the generated address exceeds the value for the address size (or, for the read-backward command is decremented below 0), an I/O program-check condition is recognized.

Figure 3-3 on page 3-7 identifies what limit values apply to the generation of different addresses and how addresses are handled when they exceed the allowed value.

Address Generation for	Address Type	Handling when Address Would Wrap
Instructions and operands when EAM and BAM are zero	L,I,R,V	W24
Successive bytes of instructions and operands when EAM and BAM are zero	I,L,V[1]	W24
Instructions and operands when EAM is zero and BAM is one	L,I,R,V	W31
Successive bytes of instructions and operands when EAM is zero and BAM is one	I,L,V[1]	W31
Instructions and operands when EAM and BAM are one	L,I,R,V	W64

Figure 3-3. Address Wraparound (Part 1 of 3).

Address Generation for	Address Type	Handling when Address Would Wrap
Successive bytes of instructions and operands when EAM and BAM are one	I,L,V[1]	W64
DAT-table entries when used for implicit translation or for LPTEA, LRA, LRAG, or STRAG	A or R[2]	X64
ASN-second-table, authority-table (during ASN authorization), linkage-table, linkage-first-table, linkage-second-table, and entry-table entries	R	X31
Authority-table (during access-register translation) and access-list entries	A or R[2]	X31
Linkage-stack entry	V	W64
I/O measurement block	A	P31
For a channel program with format-0 CCWs:		
Successive CCWs	A	P24
Successive IDAWs	A	P24
Successive MIDAWs	A	P24
Successive bytes of I/O data (without IDAWs and MIDAWs)	A	P24
Successive bytes of I/O data (with format-1 IDAWs)	A	P31
Successive bytes of I/O data (with format-2 IDAWs)	A	P64
Successive bytes of I/O data (with MIDAWs)	A	P64
For a channel program with format-1 CCWs:		
Successive CCWs	A	P31
Successive IDAWs	A	P31
Successive MIDAWs	A	P31
Successive bytes of I/O data (without IDAWs and MIDAWs)	A	P31
Successive bytes of I/O data (with format-1 IDAWs)	A	P31
Successive bytes of I/O data (with format-2 IDAWs)	A	P64
Successive bytes of I/O data (with MIDAWs)	A	P64

Figure 3-3. Address Wraparound (Part 2 of 3).

Address Generation for	Address Type	Handling when Address Would Wrap

Explanation:

1 Real addresses do not apply in this case since the instructions which designate operands by means of real addresses cannot designate operands that cross boundary 2^{24}, 2^{31}, 2^{64}.

2 It is unpredictable whether the address is absolute or real.

A Absolute address.

BAM Basic-addressing-mode bit in the PSW.

EAM Extended-addressing-mode bit in the PSW.

I Instruction address.

L Logical address.

P24 An I/O program-check condition is recognized when the address exceeds 2^{24} - 1 or is decremented below zero.

P31 An I/O program-check condition is recognized when the address exceeds 2^{31} - 1 or is decremented below zero.

P64 An I/O program-check condition is recognized when the address exceeds 2^{64} - 1 or is decremented below zero.

R Real address.

V Virtual address.

W24 Wrap to location 0 after location 2^{24} - 1 and vice versa.

W31 Wrap to location 0 after location 2^{31} - 1 and vice versa.

W64 Wrap to location 0 after location 2^{64} - 1 and vice versa.

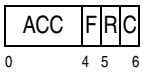
X31 When the address exceeds 2^{31} - 1, it is unpredictable whether the address wraps to location 0 after location 2^{31} - 1 or whether an addressing exception is recognized.

X64 When the address exceeds 2^{64} - 1, it is unpredictable whether the address wraps to location 0 after location 2^{64} - 1 or whether an addressing exception is recognized.

Figure 3-3. Address Wraparound (Part 3 of 3).

Storage Key

A storage key is associated with each 4 K-byte block of storage that is available in the configuration. The storage key has the following format:



The bit positions in the storage key are allocated as follows:

Access-Control Bits (ACC): If a reference is subject to key-controlled protection, the four access-control bits, bits 0-3, are matched with the four-bit access key when information is stored and when information is fetched from a location that is protected against fetching.

Fetch-Protection Bit (F): If a reference is subject to key-controlled protection, the fetch-protection bit, bit 4, controls whether key-controlled protection applies to fetch-type references: a zero indicates that only store-type references are monitored and that fetching with any access key is permitted; a one indicates that key-controlled protection applies to both fetching and storing. No distinction is made between the fetching of instructions and of operands.

Reference Bit (R): The reference bit, bit 5, normally is set to one each time a location in the corresponding storage block is referred to either for storing or for fetching of information.

Change Bit (C): The change bit, bit 6, is set to one each time information is stored at a location in the corresponding storage block.

Storage keys are not part of addressable storage. The entire storage key may be set by SET STORAGE KEY EXTENDED and inspected by INSERT STORAGE KEY EXTENDED. INSERT REFERENCE BITS MULTIPLE provides a means of inspecting the reference bits of 64 contiguous 4 K-byte blocks. RESET REFERENCE BIT EXTENDED provides a means of inspecting the reference and change bits and of setting the reference bit to zero. RESET REFERENCE BITS MULTIPLE provides a means of inspecting the reference bits of 64 contiguous 4 K-byte blocks and setting the reference bits to zero.

Bits 0-4 of the storage key are inspected by the INSERT VIRTUAL STORAGE KEY instruction. The contents of the storage key are unpredictable during and after the execution of the usability test of the TEST BLOCK instruction. When the conditional SSKE facility is installed, SET STORAGE KEY EXTENDED may be used to set all or portions of a storage key based on program-specified criteria. When the enhanced-DAT facility is installed, the SET STORAGE KEY EXTENDED or PERFORM FRAME MANAGEMENT FUNCTION instructions may be used to set all or portions of one or more storage keys based on program-specified criteria.

When EDAT-1 applies,[1] the invalid bit of the segment-table entry used in the translation is zero, and the STE-format-control is one, the following additional conditions are in effect:

- Bit position 47 of the segment-table entry contains the ACCF-validity control. The ACCF-validity control determines the validity of the access-control and fetch-protection bits in the STE. When the ACCF-validity control is zero, key-controlled protection uses the access-control and fetch-protection bits in the storage key for the 4 K-byte block corresponding to the address.
- When the ACCF-validity control is one, bit positions 48-52 of the segment-table entry contain the access-control bits and the fetch-protection bit for the segment. When determining accessibility to a storage operand, it is unpredictable whether bits 48-52 of the STE or bits 0-4 of the individual storage keys for the 4 K-byte blocks composing the segment are examined. See "Translation Process" on page 3-52 for further details.

When EDAT-2 applies[1], the invalid bit of the region-third-table entry (RTTE) used in the translation is zero, and the RTTE-format-control is one, the following conditions are in effect:

- Bit position 47 of the RTTE contains the ACCF-validity control. The ACCF-validity control determines the validity of the access-control and fetch-protection bits in the RTTE. When the ACCF-validity control is zero, key-controlled protection uses the access-control and fetch-protection bits in the storage key for the 4 K-byte block corresponding to the address.
- When the ACCF-validity control is one, bit positions 48-52 of the RTTE contain the access-control bits and the fetch-protection bit for the region. When determining accessibility to a storage operand, it is unpredictable whether bits 48-52 of the RTTE or bits 0-4 of the individual storage keys for the 4 K-byte blocks composing the segment are examined. See "Translation Process" on page 3-52 for further details.

Programming Note: When EDAT-1 applies, and both the ACCF-validity control and the STE-format control (bits 47 and 53 of the segment-table entry, respectively) are one, bits 48-52 of the STE contain access-control and fetch-protection bits which may be used in lieu of the corresponding bits in the storage keys. In this case, the program is responsible for ensuring that the access-control and fetch-protection bits in the STE are identical to the corresponding bits in each of the 256 storage keys for the segment.

Similarly, when EDAT-2 applies, and both the ACCF-validity control and the RTTE-format control (bits 47 and 53 of the region-third-table entry, respectively) are one, bits 48-52 of the RTTE contain access-control and fetch-protection bits which may be used in lieu of the corresponding bits in the storage keys. In this case, the program is responsible for ensuring that the access-control and fetch-protection bits in the RTTE are identical to the corresponding bits in each of the 524,288 storage keys for the region.

See "Modification of Translation Tables" on page 3-67 for restrictions on (a) modifying the STE or the storage keys when the STE-format control is one or (b) modifying the RTTE or the storage keys when the RTTE-format control is one.

Protection

Five protection facilities are provided to protect the contents of main storage from destruction or misuse by programs that contain errors or are unauthorized:

- Key-controlled protection
- Access-list-controlled protection
- DAT protection
- Low-address protection
- Instruction-execution protection (when installed)

1. See "Enhanced-DAT Terminology:" on page 3-41 for an explanation of EDAT applicability.

Key-Controlled Protection

When key-controlled protection applies to a storage access, a store is permitted only when the storage key matches the access key associated with the request for storage access; a fetch is permitted when the keys match or when the fetch-protection bit of the storage key is zero.

Key-controlled protection affords protection against improper storing or against both improper storing and fetching, but not against improper fetching alone.

The keys are said to match when the four access-control bits of the storage key are equal to the access key, or when the access key is zero.

The protection action is summarized in Figure 3-4.

Conditions		Is Access to Storage Permitted	
Fetch-Protection Bit of Storage Key	Key Relation	Fetch	Store
0	Match	Yes	Yes
0	Mismatch	Yes	No
1	Match	Yes	Yes
1	Mismatch	No	No

Explanation:

Match — The four access-control bits of the storage key are equal to the access key, or the access key is zero.
Yes — Access is permitted.
No — Access is not permitted. On fetching, the information is not made available to the program; on storing, the contents of the storage location are not changed.

Figure 3-4. Summary of Protection Action

When the access to storage is initiated by the CPU and key-controlled protection applies, the PSW key is the access key, except that the access key is specified in a general register for the first operand of MOVE TO SECONDARY and MOVE WITH DESTINATION KEY, for the second operand of MOVE TO PRIMARY, MOVE WITH KEY, and MOVE WITH SOURCE KEY, and for either the first or the second operand of MOVE PAGE. The access key may be the PSW key or the key specified in an operand-access control for either operand of MOVE WITH OPTIONAL SPECIFICATIONS. The PSW key occupies bit positions 8-11 of the current PSW.

When the access to storage is for the purpose of channel-program execution, the subchannel key associated with that channel program is the access key. The subchannel key for a channel program is specified in the operation-request block (ORB). When, for purposes of channel-subsystem monitoring, an access to the measurement block is made, the measurement-block key is the access key. The measurement-block key is specified by the SET CHANNEL MONITOR instruction. Even when the enhanced-DAT facility 1 is installed, channel subsystem accesses continue to reference the storage keys for each 4 K-byte block; because channel subsystem accesses are not subject to dynamic address translation, the access-control bits and fetch-protection bit in the segment-table entry are not used by the channel subsystem.

When a CPU access is prohibited because of key-controlled protection, the execution of the instruction is terminated, and a program interruption for a protection exception takes place. However, the unit of operation or the execution of the instruction may be suppressed, as described in the section “Suppression on Protection” on page 3-15. When a channel-program access is prohibited, the start function is ended, and the protection-check condition is indicated in the associated interruption-response block (IRB). When a measurement-block access is prohibited, the I/O measurement-block protection-check condition is indicated.

When a store access is prohibited because of key-controlled protection, the contents of the protected location remain unchanged. When a fetch access is prohibited, the protected information is not loaded into a register, moved to another storage location, or provided to an I/O device. For a prohibited instruction fetch, the instruction is suppressed, and an arbitrary instruction-length code is indicated.

Key-controlled protection is independent of whether the CPU is in the problem or the supervisor state and, except as described below, does not depend on the type of CPU instruction or channel-command word being executed.

Except where otherwise specified, all accesses to storage locations that are explicitly designated by the program and that are used by the CPU to store or fetch information are subject to key-controlled protection.

Key-controlled protection does not apply when the storage-protection-override control is one and the value of the four access-control bits of the storage key is 9. Key-controlled protection for fetches may or may not apply when the fetch-protection-override control is one, depending on the effective address and the private-space control.

The storage-protection-override control and fetch-protection-override control do not affect storage references made by the channel subsystem.

Accesses to the second operand of TEST BLOCK are not subject to key-controlled protection.

All storage accesses by the channel subsystem to access the I/O measurement block, or by a channel program to fetch a CCW, IDAW, or MIDAW or to access a data area designated during the execution of a CCW, are subject to key-controlled protection. However, if a CCW, an IDAW, a MIDAW, or output data is prefetched, a protection check is not indicated until the CCW, IDAW, or MIDAW is due to take control or until the data is due to be written.

Key-controlled protection is not applied to accesses that are implicitly made for any of such sequences as:

- An interruption
- CPU logout
- Fetching of table entries for access-register translation, dynamic-address translation, PC-number translation, ASN translation, or ASN authorization
- Tracing
- A store-status function
- Storing in real locations 184-191 when TEST PENDING INTERRUPTION has an operand address of zero
- Initial program loading
- Storing in real locations 6,144-6,399 when transactional execution is aborted due to an unfiltered program-exception condition

Similarly, protection does not apply to accesses initiated via the operator facilities for altering or displaying information. However, when the program explicitly designates these locations, they are subject to protection.

Storage-Protection-Override Control

Bit 39 of control register 0 is the storage-protection-override control. When this bit is one, storage-protection override is active. When this bit is zero, storage-protection override is inactive. When storage-protection override is active, key-controlled storage protection is ignored for storage locations having an associated storage-key value of 9. When storage-protection override is inactive, no special action is taken for a storage-key value of 9.

Storage-protection override applies to instruction fetch and to the fetch and store accesses of instructions whose operand addresses are logical, virtual, or real. It does not apply to accesses made for the purpose of channel-program execution or for the purpose of channel-subsystem monitoring.

Storage-protection override has no effect on accesses which are not subject to key-controlled protection.

Programming Notes:

1. Storage-protection override can be used to improve reliability in the case when a possibly erroneous application program is executed in conjunction with a reliable subsystem, provided that the application program needs to access only a portion of the storage accessed by the subsystem. The technique for doing this is as follows. The storage accessed by the application program is given storage key 9. The storage accessed by only the subsystem is given some other nonzero storage key, for example, key 8. The application is executed with PSW key 9. The subsystem is executed with PSW key 8 (in this example). As a result, the subsystem can access both the key-8 and the key-9 storage, while the application program can access only the key-9 storage.

2. Storage-protection override affects the accesses to storage made by the CPU and also affects the result set by TEST PROTECTION. However, those instructions which, in the problem state, test the PSW-key mask to determine if a particular key value may be used are not affected by whether storage-protection override is active. These instructions include, among others, MOVE WITH KEY and SET PSW KEY FROM ADDRESS. To permit these instructions to use an access key of 9 in the problem state, bit 9 of the PSW-key mask must be one.

Fetch-Protection-Override Control

Bit 38 of control register 0 is the fetch-protection-override control. When the bit is one, key-controlled fetch protection is ignored for locations at effective addresses 0-2047. An effective address is the address which exists before any transformation by dynamic address translation or prefixing. However, key-controlled fetch protection is not ignored if the effective address is subject to dynamic address translation and the private-space control, bit 55, is one in the address-space-control element used in the translation.

Fetch-protection override applies to instruction fetch and to the fetch accesses of instructions whose operand addresses are logical, virtual, or real. It does not apply to fetch accesses made for the purpose of channel-program execution or for the purpose of channel-subsystem monitoring. When this bit is set to zero, fetch protection of locations at effective addresses 0-2047 is determined by the state of the fetch-protection bit of the storage key associated with those locations.

Fetch-protection override has no effect on accesses which are not subject to key-controlled protection.

Programming Note: The fetch-protection-override control allows fetch protection of locations at addresses 2048-4095 along with no fetch protection of locations at addresses 0-2047.

Access-List-Controlled Protection

In the access-register mode, bit 6 of the access-list entry, the fetch-only bit, controls which types of operand references are permitted to the address space specified by the access-list entry. When the entry is used in the access-register-translation part of a reference and bit 6 is zero, both fetch-type and store-type references are permitted; when bit 6 is one, only fetch-type references are permitted, and an attempt to store causes a protection exception to be recognized and the execution of the instruction to be suppressed.

The fetch-only bit is included in the ALB access-list entry. A change to the fetch-only bit in an access-list entry in main storage does not necessarily have an immediate, if any, effect on whether a protection exception is recognized. However, this change to the bit will have an effect immediately after PURGE ALB or a COMPARE AND SWAP AND PURGE instruction that purges the ALB is executed.

TEST PROTECTION takes into consideration access-list-controlled protection when the CPU is in the access-register mode. A violation of access-list-controlled protection causes condition code 1 to be set, except that it does not prevent condition code 2 or 3 from being set when the conditions for those codes are satisfied.

Programming Note: A violation of access-list-controlled protection always causes suppression. A violation of any of the other protection types may cause termination.

DAT Protection

The DAT-protection function[2] controls access to virtual storage by using the DAT-protection bit in each page-table entry and segment-table entry, and, when the enhanced-DAT facility is installed, in each region-table entry. It provides protection against improper storing.

The DAT-protection bit, bit 54 of the page-table entry, controls whether storing is allowed into the corresponding 4 K-byte page. When the bit is zero, both fetching and storing are permitted; when the bit is one, only fetching is permitted. When an attempt is made to store into a protected page, the contents of the page remain unchanged, the unit of operation or the execution of the instruction is suppressed, and a program interruption for protection takes place.

The DAT-protection bit, bit 54 of the segment-table entry, controls whether storing is allowed into the corresponding 1 M-byte segment, as follows:

- When EDAT-1 does not apply, or when EDAT-1 applies and the STE-format control is zero, the DAT-protection bit of the segment-table entry is treated as being ORed into the DAT-protection-bit position of each entry in the page table designated by the segment-table entry. Thus, when the segment-table-entry DAT-protection bit is one, the effect is as if the DAT-protection bit were one in each entry in the designated page table.
- When EDAT-1 applies and the STE-format control is one, the DAT-protection bit of the segment-

2. Prior to the enhanced-DAT facility, the DAT-protection function was known as the page-protection facility.

table entry controls whether storing is allowed into the corresponding 1 M-byte segment. When the bit is zero, both fetching and storing are permitted; when the bit is one, only fetching is permitted. When an attempt is made to store into a protected segment, the contents of the segment remain unchanged, the unit of operation or the execution of the instruction is suppressed, and a program interruption for protection takes place

When EDAT-1 applies, the DAT-protection bit, bit 54 of the region-table entry, controls whether storing is allowed into the corresponding region(s). The DAT-protection bit in a region-table entry is treated as being ORed into the DAT-protection bit position of any subsequent region-table entry and segment-table entry that is used in the translation. When the STE-format control is zero, the DAT-protection bit is further propagated to the page-table entry, as described above.

DAT protection applies to all store-type references that use a virtual address.

Low-Address Protection

The low-address-protection facility provides protection against the destruction of main-storage information used by the CPU during interruption processing. In the z/Architecture architectural mode, this is accomplished by prohibiting instructions from storing with effective addresses in the ranges 0 through 511 and 4096 through 4607 (the first 512 bytes of each of the first and second 4 K-byte effective-address blocks). In the ESA/390-compatibility mode, low-address protection is limited to effective addresses in the range of 0-511 (the first 512 bytes of the first 4 K-byte effective-address block). The range criterion is applied before address transformation, if any, of the address by dynamic address translation or prefixing. However, the range criterion is not applied, with the result that low-address protection does not apply, if the effective address is subject to dynamic address translation and the private-space control, bit 55, is one in the address-space-control element used in the translation. Low-address protection does not apply if the address-space-control element to be used is not available due to another type of exception.

Low-address protection is under control of bit 35 of control register 0, the low-address-protection-control bit. When the bit is zero, low-address protection is off; when the bit is one, low-address protection is on.

If an access is prohibited because of low-address protection, the contents of the protected location remain unchanged, the execution of the instruction is terminated, and a program interruption for a protection exception takes place. However, the unit of operation or the execution of the instruction may be suppressed, as described in the section “Suppression on Protection” on page 3-15.

Any attempt by the program to store by using effective addresses in the range 0 through 511 is subject to low-address protection. In the z/Architecture architectural mode, any attempt to store by using effective addresses in the range of 4096 through 4607 is also subject to low-address protection. Low-address protection is applied to the store accesses of instructions whose operand addresses are logical, virtual, real, or absolute. Low-address protection is also applied to the trace table.

Low-address protection is not applied to accesses made by the CPU or the channel subsystem for such sequences as interruptions, CPU logout, the storing of the I/O-interruption code in real locations 184-195 by TEST PENDING INTERRUPTION, the storing of the results of STORE FACILITY LIST in real locations 200-203, and the initial-program-loading and store-status functions, nor is it applied to data stores during I/O data transfer. When the enhanced-monitor facility is installed, low-address protection is not applied when a monitor-event counting operation results in the updating the enhanced-monitor exception count in real locations 268-271. However, explicit stores by a program at any of these locations are subject to low-address protection.

Instruction-Execution Protection

The instruction-execution-protection (IEP) facility may be available on a model implementing z/Architecture. The IEP facility is installed in a configuration when facility indication 130, as stored by the STORE FACILITY LIST EXTENDED instruction, is one. When the IEP facility is installed, the access-exception-fetch/store-indication and side-effect-access facilities and the enhanced suppression-on-protection facility 2 are also installed. When installed, the IEP facility is enabled in a CPU when the instruction-execution-protection-enablement (IEPE) control, bit 43 of control register 0, is one.

When the facility is enabled, and an instruction is fetched from the primary or home address space, an

instruction-execution-protection control in the DAT leaf-table entry used in the translation determines whether instructions in the frame mapped by the entry may or may not be executed. Instruction-execution protection applies to both an instruction designated directly by the PSW instruction address and to an instruction that is the target of an execute-type instruction.

The facility may be used by a control program to better segregate instructions from data. Improved system reliability and integrity may be realized by preventing instructions from being executed in storage locations intended to contain only data. For example, erroneously or maliciously modified data in a program stack can be prevented from being executed.

When the instruction-execution-protection facility is enabled, the instruction-execution-protection control, bit 55 of a format-1 region-third-table entry, a format-1 segment-table entry, or a page-table entry, controls whether instructions in the respective region, segment, or page are allowed to be executed. When the bit is zero, the execution of instructions in the respective region, segment, or page is permitted; when the bit is one, the execution of instructions in the respective region, segment, or page is suppressed, and a protection exception is recognized with an arbitrary instruction-length code indicated.

When two DAT leaf-table entries are required to fetch an instruction, an instruction-execution-protection exception is recognized if bit 55 is one in either or both of the leaf-table entries used to translate the instruction's address.

When a protection exception is recognized due to instruction-execution protection, the protection code in bits 56, 60, and 61 of the translation-exception identification (TEID) at real locations 168-175 contain 101 binary, and it is unpredictable whether the bit positions 0-51 of the TEID contain the virtual address of the instruction or zeros.

Further details on the instruction-execution-protection facility may be found in the section "Dynamic Address Translation" on page 3-38.

Programming Notes:

1. Low-address protection and key-controlled protection apply to the same store accesses, except that:

   a. Low-address protection does not apply to storing performed by the channel subsystem, whereas key-controlled protection does.

   b. Key-controlled protection does not apply to tracing, the second operand of TEST BLOCK, or instructions that operate specifically on the linkage stack, whereas low-address protection does.

2. Because fetch-protection override and low-address protection do not apply to an address space for which the private-space control is one in the address-space-control element, locations 0-2047 and, in the z/Architecture architectural mode, locations 4096-4607 in the address space are usable the same as the other locations in the space.

Suppression on Protection

The suppression-on-protection facilities provide the means by which a control program can effectively determine whether a protection exception resulted in suppression, and if so, whether the exception was due to DAT protection – without having to unnecessarily scan DAT-table entries to determine if DAT protection was not the cause. Determining whether a protection exception was caused by DAT protection is necessary for the implementation of the POSIX fork function, discussed in programming notes below.

Depending on the model, one of three types of suppression-on-protection facility is installed:

- Basic suppression-on-protection (SOP) facility (without the side-effect access facility)
- Enhanced suppression-on-protection facility 1 (ESOP-1; without the side-effect access facility)
- Enhanced suppression on protection facility 2 (ESOP-2; with the side-effect access facility)

Whether or not a protection exception is suppressing depends on both the type of suppression-on-protec-

tion facility that is installed and the protection exception that is recognized, as shown below in Figure 3-5.

Protection Exception Recognized	Type of Suppression-on-Protection Facility Installed		
	SOP	ESOP-1	ESOP-2
Access-List-Controlled	✓	✓	✓
DAT	✓	✓	✓
Instruction-Execution	—	—	✓
Key-Controlled	?	?	☑
Low-Address	?	?	☑

Explanation:

✓	Exception is suppressing.
☑	Exception is suppressing when the protection code in bits 56, 60, and 61 of the translation-exception identification is nonzero.
—	Not applicable
?	Unpredictable whether the exception results in suppression or termination
ESOP	Enhanced suppression on protection (facility -1 or -2)
SOP	Basic suppression on protection

Figure 3-5. Suppression Attributes for Protection Exceptions

Regardless of which suppression-on-protection facility is installed, the following applies:

- When multiple protection-exception conditions apply to the same reference, the exceptions are recognized in the order described in Figure 6-9, “Priority of Access Exceptions” on page 6-54. When (a) low-address protection applies to a reference, and (b) any of access-list-controlled protection, DAT protection, or key-controlled protection also apply to the same reference, it is unpredictable whether the former or latter is indicated.
- Information identifying the cause of the protection exception is stored in the translation-exception identification (TEID) at real locations 168-175. The extent of information stored in the TEID depends on both type of suppression-on-protection facility that is installed and the type of protection exception. The TEID stored during a protection exception is further described in the section “Translation-Exception Identification for Protection Exceptions” on page 3-77.

Depending on the type of suppression-on-protection facility installed and the type of protection exception, additional information may be stored in the exception-access identification (EAID) at real location 160. The EAID is further described in the section “Exception Access Identification (EAID): During a program interruption due to an ASCE-type, region-first-translation, region-second-translation, region-third-translation, segment-translation, or page-translation exception, an indication of the address space to which the exception applies may be stored at location 160, as follows:” on page 3-74.

The following describes the fundamental differences between the three types of suppression-on-protection facilities.

Basic Suppression-on-Protection Facility

When TEID bit 61 is zero, the operation is either suppressed or terminated, and the remainder of the TEID and the EAID are unpredictable. TEID bit 61 is zero if DAT was on, but the effective address causing the exception was either real or absolute.

When TEID bit 61 is one, the following applies:

- The operation is suppressed.
- TEID bit positions 0-51 contain the effective address that caused the protection exception.
- TEID bits 52-59 are unpredictable.
- If DAT was on, the following applies:
  - The effective address that caused the protection exception is a virtual address.
  - When TEID bit 60 is zero, access-list-controlled protection is not the cause of the exception. When TEID bit 60 is one, access-list-controlled protection is the cause of the exception.
  - TEID bit positions 62 and 63 and the EAID contain additional information identifying the address space.

  If DAT was off, TEID bits 60 and 62-63 and the EAID are unpredictable.

The results of basic suppression on protection are summarized in Figure 3-6.

Exception Conditions				Presented Fields		
LAP or KCP	DAT	ALCP or DATP	Effective Address	Bit 61	If Bit 61 is One	
					Bits 62, 63 and Loc. 160	Bit 60
No	On	Yes	Log.	1	P	1A
Yes	On	Yes	Log.	U1	P	1A
Yes	Off	No	Log.	U2	U3	U3
Yes	Off	No	R/A	U2	U3	U3
Yes	On	No	Log.	U2	P	0
Yes	On	No	R/A	0R	–	–

Explanation:

–	Immaterial or not applicable.
0R	Zero because effective address is real.
1A	One if bit 61 is set to one because of access-list-controlled protection; zero otherwise.
ALCP	Access-list-controlled protection.
DATP	DAT protection.
KCP	Key-controlled protection.
LAP	Low-address protection.
Log.	Logical.
P	Predictable.
R/A	Real or absolute
U1	Unpredictable because low-address or key-controlled protection may be recognized instead of access-list-controlled or DAT protection.
U2	Unpredictable because bit 61 is only required to be set to one for access-list-controlled or DAT protection.
U3	Unpredictable because DAT is off.

Figure 3-6. Basic Suppression-on-Protection Results

Programming Note: When the basic suppression-on-protection facility is installed, the information provided in the TEID can be used to determine that DAT protection *was not* the cause of the exception, but it cannot be used directly to determine that DAT protection *was* the cause.

Enhanced Suppression-on-Protection Facility 1

The enhanced suppression-on-protection facility 1 (ESOP-1) provides additional function over that of the basic suppression-on-protection facility. When ESOP-1 is installed, the side-effect-access facility is not installed. Characteristics of ESOP-1 are described in this section.

The operation is always suppressed, regardless of the contents of TEID bit 61.

When TEID bit 61 is zero, the exception is due to either key-controlled protection or low-address protection, and the remainder of the TEID and the EAID are unpredictable.

When TEID bit 61 is one, the following applies:

- The exception is due to either access-list-controlled protection or DAT protection, as determined by TEID bit 60. When TEID bit 60 is zero, DAT protection is the cause of the exception. When TEID bit 60 is one, access-list-controlled protection is the cause of the exception.
- TEID bit positions 0-51 contain the virtual address that caused the protection exception.
- When the access-exception-fetch/store-indication facility is installed, TEID bit positions 52-53 contain the access-exception fetch/store indication.
- TEID bit positions 54-59 are unpredictable.
- TEID bit positions 62 and 63 and the EAID contain additional information identifying the address space.

Figure 3-7 summarizes the results of a protection exception when the enhanced suppression-on-protection facility 1 is installed.

Exception Type	DAT	Bits 0-51	Bits 52-53	Bit 60	Bit 61	Bits 62, 63 and Loc. 160
LAP	–	–	–	–	0	–
KCP	–	–	–	–	0	–
ALCP	Yes	A	FS	1	1	AS
DATP	Yes	A	FS	0	1	AS

Explanation:

–	Undefined.
A	Bits 0-51 of the effective address that caused the exception (This must be a virtual address).
ALCP	Access-list-controlled protection.

Figure 3-7. Enhanced Suppression-on-Protection Facility 1 Results

AS	Identifies the address space containing the effective address that caused the exception.
DAT	Dynamic address translation is performed.
DATP	DAT protection.
FS	Access-exception fetch/store indicator if the facility is installed, otherwise unpredictable.
KCP	Key-controlled protection.
LAP	Low-address protection.

Figure 3-7. Enhanced Suppression-on-Protection Facility 1 Results (Continued)

Enhanced Suppression-on-Protection Facility 2

The enhanced suppression-on-protection facility 2 (ESOP-2) provides additional function over that of ESOP-1. When ESOP-2 is installed, the side-effect-access facility is also installed. Characteristics of ESOP-2 are described in this section.

The contents of TEID bit positions 56, 60, and 61 form a three-bit binary code that identifies the cause of the protection exception. When the protection code is zero, the remainder of the TEID and the EAID are unpredictable. When the protection code is non-zero, the following applies:

- The three-bit code identifies the cause of the protection exception, as follows:

Code	Meaning
000	Key-controlled or low-address protection
001	DAT protection
010	Key-controlled protection
011	Access-list-controlled protection
100	Low-address protection
101	Instruction-execution-protection (when the instruction-execution-protection facility is installed.

- When the protection code is not 000 binary, the operation is always suppressed (regardless of the contents of TEID bit 61).
- Except as noted below, TEID bits position 0-51 contain the effective address that caused the protection exception. For access-list-controlled-protection, DAT-protection, and instruction-execution-protection exceptions, the effective address is virtual. The effective address may be virtual for key-controlled-protection and low-address-protection exceptions when DAT is on; otherwise, it is real or absolute.

  For key-controlled-protection or instruction-execution-protection exceptions recognized when fetching an instruction, it is unpredictable whether bit positions 0-51 of the TEID contain the address of the instruction or zeros.
- TEID bit positions 52-53 contain the access-exception fetch/store indication.
- TEID bit position 54 contains the side-effect-access indication.
- Bit 55 is unpredictable.
- TEID bits 57-59 are unpredictable.
- TEID bit positions 62 and 63 and the EAID contain additional information identifying the address space.

Figure 3-8 summarizes the TEID for a protection exception when the enhanced suppression-on-protection facility 2 is installed.

Exception Type	DAT	Addr Type	TEID Bit Positions (168-175)						
						Protection Code			62, 63, and Loc. 160
			0-51	52-53	54	56	60	61	
KCP or LAP[1]	–	–	–	–	–	0	0	0	–
DATP	Yes	V	@	FS	S	0	0	1	AS
KCP	No	R/A	@[2]	FS	S	0	1	0	–[3]
	Yes	V	@[2]	FS	S	0	1	0	AS
ALCP	Yes	V	@	FS	S	0	1	1	AS
LAP	No	R/A	@	FS	S	1	0	0	–[3]
	Yes	V	@	FS	S	1	0	0	AS
IEP	Yes	V	@[2]	FS	0	1	0	1	AS

Explanation:

1	Presented if TEID details are not available.
2	For an instruction fetch, zeros may be stored instead of the translation-exception address.
3	Address type cannot be determined from TEID.
@	Bits 0-51 of the address that caused the exception.
A	Absolute address.
ALCP	Access-list-controlled protection.
AS	Identifies the address space containing the effective address that caused the exception.

Figure 3-8. Enhanced Suppression-on-Protection Facility 2 Results

DAT	Dynamic address translation is performed.
DATP	DAT protection.
E	Effective address.
FS	Access-exception fetch/store indicator.
IEP	Instruction-execution protection.
KCP	Key-controlled protection.
LAP	Low-address protection.
R	Real address.
S	Side-effect-access indicator.
V	Virtual address.
–	Undefined, unpredictable.

Protection codes 110-111 binary are reserved.

Figure 3-8. Enhanced Suppression-on-Protection Facility 2 Results (Continued)

Facility indication 131, when one, indicates that the enhanced-suppression-on-protection facility 2 and the side-effect-access facility are both installed.

Programming Notes: Except as noted, the following program notes are applicable regardless of which suppression-on-protection facility is installed.

1. The suppression-on-protection function is useful in performing the POSIX fork function, which causes a duplicate address space to be created. The following discussion pertains to (a) when EDAT-1 does not apply, (b) when EDAT-1 applies but the STE-format-control in the segment-table entry is zero, or (c) when EDAT-2 applies but the RTTE-format control in the region-third-table entry is zero.

   The duplicate address space is initially created by making a copy of the DAT tables mapping the original address space, but marking each page-table entry in the duplicate address space as DAT protected. Fetch accesses in the duplicate address space use the same common page frame as that of the original address space. However, when a store is attempted in the duplicate address space, the control program recognizes the DAT-protection exception, assigns a unique page frame for this page in the duplicate address space, and copies the contents of the original block into the duplicate address space's block (a process known as the copy-on-write function). The control program may initially set the DAT-protection bit to one in a higher-level DAT-table entry to detect an attempt to store anywhere in the blocks mapped by lower-level DAT-table entries.

   When EDAT-1 applies, and the STE-format control in the segment-table entry is one, a similar technique may be used to map a single segment frame of absolute storage. Similarly, when EDAT-2 applies, and the RTTE-format control in the region-third-table entry is one, this technique may be used to map a single region frame of absolute storage.

2. For the basic suppression-on-protection facility, when DAT is on, TEID bit 61 being one indicates that the address that caused a protection exception is virtual, and TEID bit 60 being zero indicates that the exception was not caused by access-list-controlled protection. These indications allow programmed forms of access-register translation and dynamic address translation to be performed to locate the DAT-table entries used in the translation, and determine whether the exception was due to DAT protection (as opposed to low-address or key-controlled protection).

   For the enhanced-suppression-on-protection facility 1, TEID bits 60 and 61 being 01 binary indicate that DAT protection was the cause of the exception. For the enhanced-suppression-on-protection facility 2, TEID bits 56, 60, and 61 being 001 binary indicate that DAT protection was the cause of the exception.

   However, regardless of which suppression-on-protection-exception facility applies, the control program must still locate the appropriate DAT-table entries to effect a copy-on-write function.

Reference Recording

Reference recording provides information for use in selecting pages for replacement. Reference recording uses the reference bit, bit 5 of the storage key. The reference bit is set to one each time a location in the corresponding storage block is referred to either for fetching or for storing information, regardless of whether DAT is on or off.

Reference recording is always active and takes place for all storage accesses, including those made by any CPU, any operator facility, or the channel subsystem. It takes place for implicit accesses made by the

machine, such as those which are part of interruptions and I/O-instruction execution.

Reference recording does not occur for operand accesses of the following instructions since they directly refer to a storage key without accessing a storage location:

- INSERT REFERENCE BITS MULTIPLE
- INSERT STORAGE KEY EXTENDED
- RESET REFERENCE BIT EXTENDED (reference bit is set to zero)
- RESET REFERENCE BITS MULTIPLE (reference bits of 64 consecutive 4 K-byte blocks are set to zero)
- PERFORM FRAME MANAGEMENT FUNCTION and SET STORAGE KEY EXTENDED (reference bit may be set to a specified value)

The record provided by the reference bit is substantially accurate. The reference bit may be set to one by fetching data or instructions that are neither designated nor used by the program, and, under certain conditions, a reference may be made without the reference bit being set to one. Under certain unusual circumstances, a reference bit may be set to zero by other than explicit program action.

When the CPU is in the transactional-execution mode, it is unpredictable whether the reference bit is set coincident with the transaction's storage accesses or at the end of the transaction. Reference bits that are set during the execution of a transaction may remain set if the transaction is aborted.

Change Recording

Change recording provides information as to which pages have to be saved in auxiliary storage when they are replaced in main storage. Change recording uses the change bit, bit 6 of the storage key.

The change bit is set to one each time a store access causes the contents of the corresponding storage block to be changed. A store access that does not change the contents of storage may or may not set the change bit to one.

The change bit is not set to one for an attempt to store if the access is prohibited. In particular:

1. For the CPU, a store access is prohibited whenever an access exception exists for that access, or whenever an exception exists which is of higher priority than the priority of an access exception for that access.
2. For the channel subsystem, a store access is prohibited whenever a key-controlled-protection violation exists for that access.

Change recording is always active and takes place for all store accesses to storage, including those made by any CPU, any operator facility, or the channel subsystem. It takes place for implicit references made by the machine, such as those which are part of interruptions.

Change recording does not take place for the operands of the following instructions since they directly modify a storage key without modifying a storage location:

- RESET REFERENCE BIT EXTENDED
- RESET REFERENCE BITS MULTIPLE
- SET STORAGE KEY EXTENDED (change bit may be set to a specified value)

Change bits which have been changed from zeros to ones are not necessarily restored to zeros on CPU retry (see "CPU Retry" on page 11-2). See "Exceptions to Nullification and Suppression" on page 5-26 for a description of the handling of the change bit in certain unusual situations. It is possible that the change bit may be over-indicated for any storage which is referenced before execution of the first instruction that sets or extracts a storage key, sets the PSW key to a non-zero value or uses a non-zero alternate key for storage access.

When the condition specified by the M_3 field of a STORE ON CONDITION instruction is not met, it is model dependent whether the change bit is set for the storage location designated by the second operand.

When the CPU is in the transactional-execution mode, it is unpredictable whether the change bit is set coincident with the transaction's store accesses or at the end of the transaction. Change bits that are set during the execution of a transaction may remain set if the transaction is aborted.

Prefixing

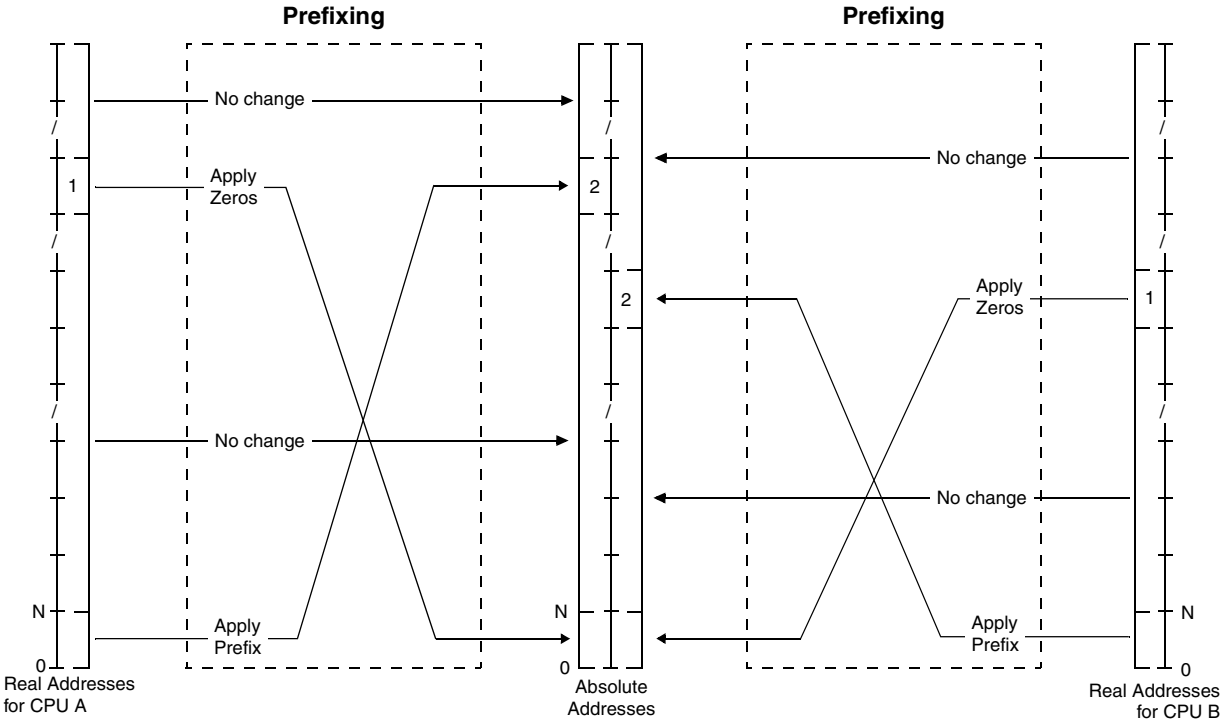
Prefixing provides the ability to assign the block of real addresses containing assigned storage locations to a different block in absolute storage for each CPU, thus permitting more than one CPU sharing main storage to operate concurrently with a minimum of interference, especially in the processing of interruptions.

In the z/Architecture architectural mode, prefixing causes real addresses in the range 0-8191 to correspond one-for-one to the block of 8 K-byte absolute addresses (the prefix area) identified by the value in bit positions 0-50 of the prefix register for the CPU, and the block of real addresses identified by that value in the prefix register to correspond one-for-one to absolute addresses 0-8191.

In the ESA/390-compatibility mode, prefixing causes real addresses in the range 0-4,095 to correspond one-for-one to the block of 4 K-byte absolute addresses (the prefix area) identified by the value in bit positions 0-51 of the prefix register for the CPU, and the block of real addresses identified by that value in the prefix register to correspond one-for-one to absolute addresses 0-4,095.

The remaining real addresses are the same as the corresponding absolute addresses. This transformation allows each CPU to access all of main storage, including the first 8K bytes and the locations designated by the prefix registers of other CPUs.

The relationship between real and absolute addresses is graphically depicted in Figure 3-9 on page 3-21.



Explanation:

N In the z/Architecture architectural mode, N is 8,192. In the ESA/390-compatibility mode, N is 4,096.

1 In the z/Architecture architectural mode, real addresses in which bits 0-50 are equal to bits 0-50 of the prefix for this CPU (A or B). In the ESA/390-compatibility mode, real addresses in which bits 0-51 are equal to bits 0-51 of the prefix for this CPU (A or B).

2 Absolute addresses of the block that contains for this CPU (A or B) the real locations 0 through (N–1)

Figure 3-9. Relationship between Real and Absolute Addresses

Prefixing in the z/Architecture Architectural Mode

In the z/Architecture architectural mode, the prefix is a 51-bit quantity contained in bit positions 0-50 of the prefix register. Figure 3-10 illustrates the prefix register in the z/Architecture architectural mode.

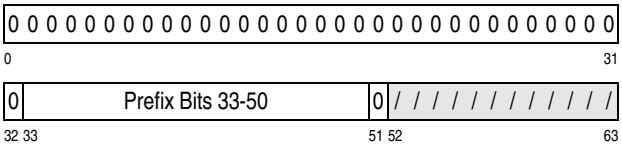


Figure 3-10. z/Architecture Prefix Register

Bits 33-50 of the register can be set and inspected by the privileged instructions SET PREFIX and STORE PREFIX, respectively.

SET PREFIX sets bits 33-50 of the prefix register with the value in bit positions 1-18 of a word in storage, and it ignores the contents of bit positions 0 and 19-31 of the word. STORE PREFIX stores the value in bit positions 33-50 of the prefix register in bit positions 1-18 of a word in storage, and it stores zeros in bit positions 0 and 19-31 of the word.

When prefixing is applied, the real address is transformed into an absolute address by using one of the following rules, depending on bits 0-50 of the real address:

1. Bits 0-50 of the address, if all zeros, are replaced with bits 0-50 of the prefix.
2. Bits 0-50 of the address, if equal to bits 0-50 of the prefix, are replaced with zeros.
3. Bits 0-50 of the address, if not all zeros and not equal to bits 0-50 of the prefix, remain unchanged.

Bit 51 of the prefix register is always set to zero and does not participate in the prefixing process.

Prefixing in the ESA/390-Compatibility Mode

In the ESA/390-compatibility mode, the prefix is a 52-bit quantity contained in bit positions 0-51 of the prefix register. Figure 3-11 illustrates the prefix register in the ESA/390-compatibility mode.

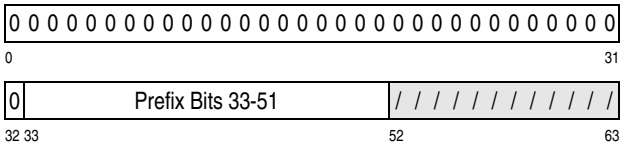


Figure 3-11. ESA/390-Compatibility Mode Prefix Register

Bits 33-51 of the register can be set and inspected by the privileged instructions SET PREFIX and STORE PREFIX, respectively.

SET PREFIX sets bits 33-51 of the prefix register with the value in bit positions 1-19 of a word in storage, and it ignores the contents of bit positions 0 and 20-31 of the word. STORE PREFIX stores the value in bit positions 33-51 of the prefix register in bit positions 1-19 of a word in storage, and it stores zeros in bit positions 0 and 20-31 of the word.

When prefixing is applied, the real address is transformed into an absolute address by using one of the following rules, depending on bits 0-51 of the real address:

1. Bits 0-51 of the address, if all zeros, are replaced with bits 0-51 of the prefix.
2. Bits 0-51 of the address, if equal to bits 0-51 of the prefix, are replaced with zeros.
3. Bits 0-51 of the address, if not all zeros and not equal to bits 0-51 of the prefix, remain unchanged.

Common Prefixing Attributes

Bits 0-32 of the prefix register are always all zeros. When the contents of the prefix register are changed, the change is effective for the next sequential instruction.

Only the address presented to storage is translated by prefixing. The contents of the source of the address remain unchanged.

The distinction between real and absolute addresses is made even when the prefix register contains all zeros, in which case a real address and its corresponding absolute address are identical.

Address Spaces

Note: In the ESA/390-compatibility mode, ASN translation and dynamic address translation are not supported, however absolute-storage address spaces may be supported, as described in Reference [12.] on page xxx.

An address space is a consecutive sequence of integer numbers (virtual addresses), together with the specific transformation parameters which allow each number to be associated with a byte location in storage. The sequence starts at zero and proceeds left to right.

When a virtual address is used by a CPU to access main storage, it is first converted, by means of dynamic address translation (DAT), to a real address, and then, by means of prefixing, to an absolute address. DAT may use from five to two levels of tables (region first table, region second table, region third table, segment table, and page table) as transformation parameters. The designation (origin and length) of the highest-level table for a specific address space is called an address-space-control element, and it is found for use by DAT in a control register or as specified by an access register. Alternatively, the address-space-control element for an address space may be a real-space designation, which indicates that DAT is to translate the virtual address simply by treating it as a real address and without using any tables.

DAT uses, at different times, the address-space-control elements in different control registers or specified by the access registers. The choice is determined by the translation mode specified in the current PSW. Four translation modes are available: primary-space mode, secondary-space mode, access-register mode, and home-space mode. Different address spaces are addressable depending on the translation mode.

At any instant when the CPU is in the primary-space mode or secondary-space mode, the CPU can translate virtual addresses belonging to two address spaces — the primary address space and the secondary address space. At any instant when the CPU is in the access-register mode, it can translate virtual addresses of up to 16 address spaces — the primary address space and up to 15 AR-specified address spaces. At any instant when the CPU is in the home-space mode, it can translate virtual addresses of the home address space.

The primary address space is identified as such because it consists of primary virtual addresses, which are translated by means of the primary address-space-control element (ASCE). Similarly, the secondary address space consists of secondary virtual addresses translated by means of the secondary ASCE, the AR-specified address spaces consist of AR-specified virtual addresses translated by means of AR-specified ASCEs, and the home address space consists of home virtual addresses translated by means of the home ASCE. The primary and secondary ASCEs are in control registers 1 and 7, respectively. The AR-specified ASCEs are in control registers 1 and 7 and in table entries called ASN-second-table entries. The home ASCE is in control register 13.

Changing to Different Address Spaces

A program can cause different address spaces to be addressable by using the semiprivileged SET ADDRESS SPACE CONTROL or SET ADDRESS SPACE CONTROL FAST instruction to change the translation mode to the primary-space mode, secondary-space mode, access-register mode, or home-space mode. However, SET ADDRESS SPACE CONTROL and SET ADDRESS SPACE CONTROL FAST can set the home-space mode only in the supervisor state. The program can cause still other address spaces to be addressable by using unprivileged instructions to change the contents of the access registers and by using semiprivileged or privileged instructions to change the address-space-control elements in control registers 1 and 7. The semiprivileged instructions are ones that cause linkage from one address space to another and are the BRANCH IN SUBSPACE GROUP, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, and SET SECONDARY ASN WITH INSTANCE instructions. The privileged instructions are the LOAD ADDRESS SPACE PARAMETERS and LOAD CONTROL instructions. Only LOAD CONTROL is available for changing the home address-space-control element in control register 13.

Address-Space Number

An address space may be assigned an address-space number (ASN) by the control program. The ASN designates, within a two-level table structure in

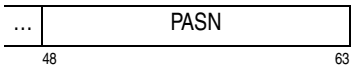
main storage, an ASN-second-table entry containing information about the address space. If the ASN-second-table entry is marked as valid, it contains the address-space-control element that defines the address space.

Under certain circumstances, the semiprivileged instructions which place a new address-space-control element in control register 1 or 7 fetch this element from an ASN-second-table entry. Some of these instructions use an ASN-translation mechanism which, given an ASN, can locate the designated ASN-second-table entry.

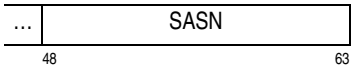
The 16-bit unsigned binary format of the ASN permits 64K unique ASNs.

The ASNs for the primary and secondary address spaces are assigned positions in control registers. The ASN for the primary address space, called the primary ASN, is assigned bits 48-63 in control register 4, and that for the secondary address space, called the secondary ASN, is assigned bits 48-63 in control register 3. Bits 48-63 of these registers have the following formats:

Control Register 4



Control Register 3



A semiprivileged instruction that loads the primary or secondary address-space-control element into the appropriate control register also loads the corresponding ASN into the appropriate control register.

The ASN for the home address space is not assigned a position in a control register.

An access register containing the value 0 or 1 specifies the primary or secondary address space, respectively; and the address-space-control element specified by the access register is in control register 1 or 7, respectively. An access register containing any other value designates an entry in a table called an access list. The designated access-list entry contains the real address of an ASN-second-table entry for the address space specified by the access register. The address-space-control element specified by the access register is in the ASN-second-table entry.

Translating the contents of an access register to obtain an address-space-control element for use by DAT does not involve the use of an ASN.

Note: Virtual storage consisting of byte locations ordered according to their virtual addresses in an address space is usually referred to as “storage.”

Programming Note: Because an ASN-second-table entry is located from an access-list entry by means of its address instead of by means of its ASN, the ASN-second-table entries designated by access-list entries can be “pseudo” ASN-second-table entries, that is, entries which are not in the two-level structure able to be indexed by means of the ASN-translation process. The number of unique pseudo ASN-second-table entries can be greater than the number of unique ASNs and is limited only by the amount of storage available to be occupied by the ASN-second-table entries. Thus, in a sense, there is no limit on the number of possible address spaces.

ASN-Second-Table-Entry Sequence Number

The ASN-second-table entry contains an ASN-second-table-entry sequence number (ASTESN) that may be used to control storage references to the related address space by means of an access register or through use of the SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, or LOAD ADDRESS SPACE PARAMETERS instruction and that may be used to control linkages to or back to the address space by means of the BRANCH IN SUBSPACE GROUP, PROGRAM CALL, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, and PROGRAM RETURN instructions. These uses of the ASTESN are described below.

When any instruction uses an access register to designate an access-list entry, which in turn designates an ASN-second-table entry, to perform a storage reference to the address space specified by the ASN-second-table entry, an ASTESN in the access-list entry is compared to the ASTESN in the ASN-second-table entry, and these ASTESNs must be equal; otherwise, an ASTE-sequence exception is recognized. This use of the ASTESN allows an access-list entry to be made unusable if some authorization policy is changed or the designated ASN-second-table entry is reassigned to specify a conceptually different address space. The entry can be made unusable by

changing the ASTESN in the ASN-second-table entry. The use is further described in “Revoking Accessing Capability:” on page 5-51.

The ASTESN is used in connection with subspace groups as follows:

- When BRANCH IN SUBSPACE GROUP uses an access register to designate an access-list entry, which in turn designates an ASN-second-table entry, to transfer control to the subspace specified by the ASN-second-table entry, an ASTESN in the access-list entry is compared to the ASTESN in the ASN-second-table entry, and these ASTESNs must be equal.
- When BRANCH IN SUBSPACE GROUP uses, in an access register, an access-list-entry token with the value 00000001 hex to transfer control to the subspace specified by the subspace-ASN-second-table-entry origin in the current dispatchable-unit control table (designated by control register 2), a subspace-ASN-second-table-entry sequence number (SSASTESN) in the dispatchable-unit control table is compared to the ASTESN in the ASN-second-table entry for the subspace, and the SSASTESN and ASTESN must be equal.
- When LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, or SET SECONDARY ASN WITH INSTANCE sets the primary address-space-control element in control register 1 or the secondary address-space-control element in control register 7 with bits of the address-space-control element in the ASN-second-table entry for a subspace, the SSASTESN is compared to the ASTESN in the ASN-second-table entry for the subspace, and the SSASTESN and ASTESN must be equal.

Otherwise, in all of the above three cases, an ASTE-sequence exception is recognized, except that LOAD ADDRESS SPACE PARAMETERS sets condition code 1 or 2 depending on whether the subspace is the primary address space or the secondary address space, respectively. These uses of the ASTESN allow an access-list entry and the subspace-ASN-second-table-entry origin in the current dispatchable-unit control table to be made unusable when the ASN-second-table entry either of them designates is reassigned to specify a conceptually different address space. The access-list entry and subspace-ASN-second-table-entry origin can be made unusable by changing the ASTESN in the ASN-second-table entry. The uses are further described in “Subspace Groups” on page 5-66.

Programming Note: The above operations use an ASN-second-table-entry origin in either an access-list entry or the dispatchable-unit control table; they do not use an ASN. The designated ASN-second-table entry is normally a pseudo ASN-second-table entry (one not in the structure able to be indexed by means of an ASN).

ASN-Second-Table-Entry Instance Number and ASN Reuse

The ASN-and-LX-reuse facility may be installed on the model. If this facility is installed, the ASN-second-table entry contains an ASN-second-table-entry instance number (ASTEIN), and certain new definitions related to the ASTEIN and to a new linkage-second-table-entry sequence number (LSTESN) apply. The definitions related to the ASTEIN are summarized below and also given in the appropriate sections of this publication.

- The ASN-and-LX-reuse facility includes the following new instructions:
  - EXTRACT PRIMARY ASN AND INSTANCE
  - EXTRACT SECONDARY ASN AND INSTANCE
  - PROGRAM TRANSFER WITH INSTANCE
  - SET SECONDARY ASN WITH INSTANCE
- The facility also includes the following new control bits:
  - The ASN-and-LX-reuse control (R), bit 44 of control register 0
  - The controlled-ASN bit (CA), bit 30 of word 1 of the ASN-second-table entry
  - The reusable-ASN bit (RA), bit 31 of word 1 of the ASN-second-table entry

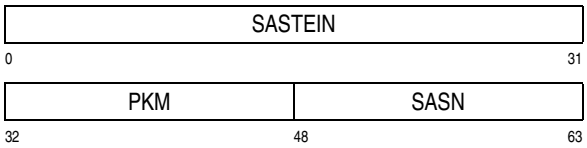
  The three control bits are shown as follows:

  Control Register 0

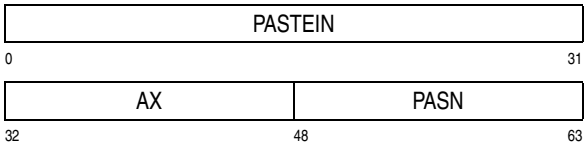
Word 1 of ASN-Second-Table Entry

- The facility also includes the primary ASTEIN in bit positions 0-31 of control register 4 and the secondary ASTEIN in bit positions 0-31 of control register 3. The primary ASTEIN is a copy of the ASTEIN in the ASN-second-table entry for the current primary address space specified by the PASN in control register 4, and the secondary ASTEIN is a copy of the ASTEIN in the ASN-second-table entry for the current secondary address space specified by the SASN in control register 3. The complete formats of control registers 3 and 4 are as follows:

  Control Register 3



  Control Register 4



- The following operations are performed if the ASN-and-LX-reuse control, bit 44 of control register 0, is one. When LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, or SET SECONDARY ASN (1) sets the primary ASN in control register 4 or the secondary ASN in control register 3 equal to a specified ASN and (2) sets the primary address-space-control element in control register 1 or the secondary address-space-control element in control register 7, respectively, with the value of an address-space-control element obtained from an ASN-second-table entry located by means of ASN translation of the specified ASN (or located, by PROGRAM CALL only, by means of an ASTE address), the instruction also sets the primary ASTEIN in control register 4 or secondary ASTEIN in control register 3, respectively, with the value of the ASTEIN in the ASN-second-table entry. However, PROGRAM TRANSFER and SET SECONDARY ASN, which are performing their space-switching operations, recognize a special-operation exception if the reusable-ASN bit, bit 31 of word 1, in the ASN-second-table entry is one.

  When the ASN-and-LX-reuse control is one and any of the instructions named above or BRANCH IN SUBSPACE GROUP sets the secondary ASN and secondary address-space-control element equal to the primary ASN (in the space-switching stacking PROGRAM CALL operation, this may be the old or the new primary ASN, as determined by a bit in the entry-table entry used) and primary address-space-control element, respectively, it also sets the secondary ASTEIN in control register 3 equal to the primary ASTEIN in control register 4 (again, in the space-switching stacking PROGRAM CALL case, this may be the old or the new primary ASTEIN). Since this function does not include the accessing of an ASN-second-table entry, it is not affected by a reusable-ASN bit.

  PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE operate as described above except (1) independent of the value of the ASN-and-LX-reuse control (they always update the primary ASTEIN and secondary ASTEIN as described above), (2) without recognizing an exception due to the reusable-ASN bit (they ignore the bit), and (3) in their space-switching forms and the problem state at the beginning of the operation, by recognizing a special-operation exception if the controlled-ASN bit, bit 30 of word 1, is one in the ASN-second-table entry located by ASN translation. PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE ignore the controlled-ASN bit if the CPU is in the supervisor state at the beginning of the operation. (PROGRAM TRANSFER WITH INSTANCE may switch the CPU from the supervisor state to the problem state.)

- PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE operate the same as PROGRAM TRANSFER and SET SECONDARY ASN, respectively, except as described in the preceding item and except as follows.

  Independent of the ASN-and-LX-reuse control and the reusable-ASN bit, but not of the controlled-ASN bit, each of PROGRAM TRANSFER

WITH INSTANCE with space switching and SET SECONDARY ASN WITH INSTANCE with space switching, after it has used the ASN in the R_1 general register to locate an ASN-second-table entry, compares an ASTEIN in bit positions 0-31 of general register R_1 to the ASTEIN in the entry. The two ASTEINs must be equal; otherwise, an ASTE-instance exception is recognized. This comparison is performed in both the problem and the supervisor state.

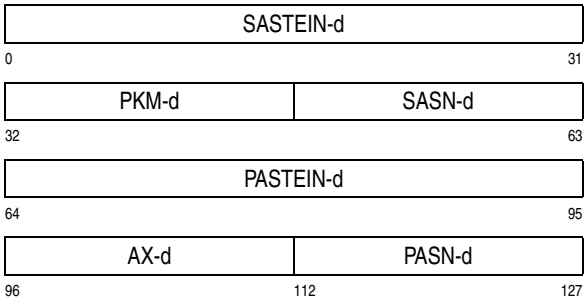
- The following operations are performed if the ASN-and-LX-reuse control is one. Each of stacking PROGRAM CALL and BRANCH AND STACK places the current primary ASTEIN in bytes 180-183 of the linkage-stack state entry that it forms, and it places the current secondary ASTEIN, bits 0-31 of control register 3, in bytes 176-179 of the state entry.

  Note: There is not a test of an ASTEIN related to the ASN in the entry-table entry used by PROGRAM CALL. There may be a test of a linkage-second-table-entry sequence number related to the linkage index used to locate the entry-table entry, as summarized in “ASN-and-LX-Reuse Control:” on page 5-30.

- The following operations are performed if the ASN-and-LX-reuse control is one. PROGRAM RETURN with space switching, after it has used the PASN in bytes 134 and 135 of the linkage-stack program-call state entry to locate an ASN-second-table entry (because that PASN is not equal to the current PASN in control register 4), compares the primary ASTEIN in bytes 180-183 of the state entry to the ASTEIN in the ASN-second-table entry. PROGRAM RETURN to current primary or with space switching, if it uses the SASN in bytes 130 and 131 of the state entry to locate an ASN-second-table entry (because that SASN is not equal to the new PASN), compares the secondary ASTEIN in bytes 176-179 of the state entry to the ASTEIN in the ASN-second-table entry. The one or two comparisons of ASTEINs must each give equal results; otherwise, an ASTE-instance exception is recognized. These operations occur independent of the controlled-ASN and reusable-ASN bits in the ASN-second-table entry. PROGRAM RETURN does not compare ASTEINs when it unstacks a branch state entry.

- The following format applies, and the operations are performed, if the ASN-and-LX-reuse control is one. The first operand of LOAD ADDRESS SPACE PARAMETERS is two consecutive doublewords having the following format:

  First Operand of LOAD ADDRESS SPACE PARAMETERS



  If PASN translation is performed to locate an ASN-second-table entry, PASTEIN-d is compared to the ASTEIN in the ASN-second-table entry, and they must be equal; otherwise, condition code 1 is set. If SASN translation is performed to locate an ASN-second-table entry, SASTEIN-d is compared to the ASTEIN in the ASN-second-table entry, and they must be equal; otherwise, condition code 2 is set. These operations are performed independent of the controlled-ASN and reusable-ASN bits in the ASN-second-table entries.

- Independent of the ASN-and-LX-reuse control, EXTRACT PRIMARY ASN AND INSTANCE places the current primary ASN, bits 48-63 of control register 4, in bit positions 48-63 of general register R_1; places the current primary ASTEIN, bits 0-31 of control register 4, in bit positions 0-31 of the general register; and places zeros in bit positions 32-47 of the general register. EXTRACT SECONDARY ASN AND INSTANCE operates the same except that it obtains the secondary ASN and secondary ASTEIN in control register 3 for placement in the general register.

- Independent of the ASN-and-LX-reuse control, a new code, code 5, of the EXTRACT STACKED STATE instruction is valid. Code 5 causes the saved secondary ASTEIN, bytes 176-179 of the state entry, to be placed in bit positions 0-31 of the R_1 general register and the saved primary ASTEIN, bytes 180-183 of the state entry, to be placed in bit positions 0-31 of general register $R_1 + 1$. Bits 32-63 of the general registers remain unchanged.

These uses of the ASTEIN allow, in all cases except one, an ASN associated with a particular ASTEIN to be made unusable when the ASN and the ASN-second-table entry it designates are reassigned to specify a conceptually different address space. The ASN-and-ASTEIN combination can be made unusable by (1) setting to one the reusable-ASN bit in the ASN-second-table entry, which prevents the use of PROGRAM TRANSFER and SET SECONDARY ASN, and (2) changing the ASTEIN in the ASN-second-table entry, which prevents the use of LOAD ADDRESS SPACE PARAMETERS, PROGRAM TRANSFER WITH INSTANCE, PROGRAM RETURN, and SET SECONDARY ASN WITH INSTANCE. The uncovered case is that of an ASN (and the corresponding ASN-second-table-entry address) in an entry-table entry, and this case should be handled by means of deletion, by the control program, of the entry-table entry. However, if the control program then reforms the entry-table entry (forms an entry-table entry located by means of the same PC number, used by PROGRAM CALL, as was the deleted entry-table entry), a PC number that was used to link to the conceptually deleted address space will then be usable to link to a different (different ASN) or conceptually different (same ASN but different contents) address space, and this error should be avoided by using the linkage-second-table-entry sequence number as described in “ASN-and-LX-Reuse Control:” on page 5-30.

Programming Notes:

1. The reusable-ASN bit of the ASN-and-LX-reuse facility provides reliability, availability, and serviceability. Most importantly, it provides availability since it allows ASNs to be reused. However it does not provide system integrity since the ASTEIN in the general register used by PROGRAM TRANSFER WITH INSTANCE or SET SECONDARY ASN WITH INSTANCE is provided by the program. (The authorization index used by those instructions normally provides system integrity but may fail to do so if an ASN authorized by an unchanged authorization index is reused.) The controlled-ASN bit provides system integrity since, if the CPU is in the problem state at the beginning of the operation, PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE are unable to proceed successfully after accessing an ASN-second-table entry in which the controlled-ASN bit is one. When the calling program and called program are both executed in the problem state, a program in an address space for which the controlled-ASN bit is one should perform linkages to programs in other address spaces only by means of stacking PROGRAM CALL. If that is done, the return by means of PROGRAM RETURN will always be successful, or will fail appropriately when the ASN of the first address space has been changed, because of the comparison to the saved primary ASTEIN in the linkage-stack state entry.

2. A program given control by a basic PROGRAM CALL operation can use EXTRACT SECONDARY ASN AND INSTANCE to obtain the ASTEIN to be used by PROGRAM TRANSFER WITH INSTANCE to return to the calling program or by SET SECONDARY ASN WITH INSTANCE to restore its secondary address space after a change of that space. This EXTRACT SECONDARY ASN AND INSTANCE instruction should be executed while the original secondary space remains continuously the secondary space; otherwise, depending on actions by the control program, EXTRACT SECONDARY ASN AND INSTANCE may return an ASTEIN that allows return to or use of a conceptually incorrect secondary space for which the ASTEIN has been changed.

3. A summary of the functions related to ASN reuse is given in Figure 3-12 on page 3-29.

4. There are certain programming errors, or situations that are not necessarily errors, that will not be detected. These cases are described in the definitions of the named instructions and are as follows:

   - In LOAD ADDRESS SPACE PARAMETERS:

     – SASN translation is performed only when, but not necessarily when, SASN-d is not equal to PASN-d. When SASN-d is equal to PASN-d, SASCE-new and SASTEIN-new are set equal to PASCE-new and PASTEIN-new, respectively. In this case, there is not a test of whether SASTEIN-d is equal to PASTEIN-d; SASTEIN-d is ignored.

     – When SASN-d is not equal to PASN-d and is equal to SASN-old, bit 61 (force ASN translation) is zero, and bit 63 (skip SASN authorization) is one, SASN translation is not performed, and SASCE-old and SASTEIN-old become SASCE-new

and SASTEIN-new, respectively. In this case, there is not a test of whether SASTEIN-d is equal to SASTEIN-old; SASTEIN-d is ignored.

- In PROGRAM RETURN, if the new SASN is equal to the new PASN, the SASCE in control register 7 is set equal to the new PASCE in control register 1. The SASTEIN, PKM, and SASN in control register 3 remain as restored from the state entry. In this case, there is not a test of whether the new SASTEIN is equal to the new PASTEIN. (There has already been a test of whether the PASTEIN saved in the entry, which becomes the new PASTEIN, equals the ASTEIN in the new PASTE.)
- In the PROGRAM TRANSFER WITH INSTANCE to-current-primary operation, there is not a test of whether the current PASTEIN equals the ASTEIN specified in bit positions 0-31 of general register R_1; the ASTEIN is ignored.
- In the SET SECONDARY ASN WITH INSTANCE to-current-primary operation, there is not a test of whether the current PASTEIN (to which the SASTEIN is set equal) equals the ASTEIN specified in bit positions 0-31 of general register R_1; the ASTEIN is ignored.

Function	Required Bit Value		
	R	CA	RA
LASP, PC, and PR update PASTEIN or SASTEIN in a control register after accessing an ASN-second-table entry	1	–	–
PT and SSAR update PASTEIN or SASTEIN in a control register after accessing an ASN-second-table entry	1	–	0[1]
BSG, LASP, PC, PR, PT, and SSAR set SASTEIN equal to PASTEIN	1	–	–
PTI and SSAIR update PASTEIN or SASTEIN in a control register after accessing an ASN-second-table entry	–	0[2]	–
PTI and SSAIR set SASTEIN equal to PASTEIN	–	–	–
PTI and SSAIR after accessing an ASN-second-table entry, compare ASTEIN in general register R_1 to ASTEIN in the entry	–	0[2]	–
Stacking PC and BAKR copy PASTEIN and SASTEIN from control registers to state entry	1	–	–
PR, after accessing an ASN-second-table entry, compares PASTEIN or SASTEIN in the state entry to ASTEIN in the ASN-second-table entry[3]	1	–	–
LASP, after accessing an ASN-second-table entry, compares PASTEIN-d or SASTEIN-d to ASTEIN in the entry[4]	1	–	–
EPAIR copies PASTEIN and PASN from control register 4 to general register R_1	–	–	–
ESAIR copies SASTEIN and SASN from control register 3 to general register R_1	–	–	–
ESTA code 5 copies PASTEIN and SASTEIN from state entry to general registers	–	–	–

Explanation:

– Bit is ignored or not applicable to the operation.
1 A special-operation exception is recognized if the bit is one.
2 A special-operation exception is recognized if the bit is one and the CPU is in the problem state at the beginning of the operation.
3 An ASTE-instance exception is recognized if the ASTEINs are not equal.
4 Condition code 1 (if PASTEIN comparison) or 2 (if SASTEIN comparison) is set if the ASTEIN is not equal.
CA Controlled-ASN bit, bit 30 of word 1 of ASN-second-table entry.
R ASN-and-LX-reuse control, bit 44 of control register 0.
RA Reusable-ASN bit, bit 31 of word 1 of ASN-second-table entry.

Figure 3-12. Summary of Functions Related to ASN Reuse (Part 1 of 2)

Figure 3-12. Summary of Functions Related to ASN Reuse (Part 2 of 2)

ASN Translation

Note: The ASN-translation concepts described in following section are applicable only to the z/Architecture architectural mode where dynamic address translation is supported. ASN translation is not supported in the ESA/390-compatibility mode.

ASN translation is the process of translating a 16-bit ASN to locate the ASN-second-table entry designated by the ASN. ASN translation is performed as part of PROGRAM TRANSFER with space switching (PT-ss) PROGRAM TRANSFER WITH INSTANCE with space switching (PTI-ss), SET SECONDARY ASN with space switching (SSAR-ss), and SET SECONDARY ASN WITH INSTANCE with space switching (SSAIR-ss), and it may be performed as part of LOAD ADDRESS SPACE PARAMETERS. For PT-ss and PTI-ss the ASN which is translated replaces the primary ASN in control register 4. For SSAR-ss and SSAIR-ss, the ASN which is translated replaces the secondary ASN in control register 3. These two translation processes are called primary ASN translation and secondary ASN translation, respectively, and both can occur for LOAD ADDRESS SPACE PARAMETERS. The ASN-translation process is the same for both primary and secondary ASN translation; only the uses of the results of the process are different.

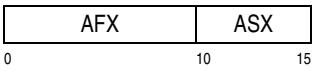
ASN translation may also be performed as part of PROGRAM RETURN. Primary ASN translation is performed as part of PROGRAM RETURN with space switching (PR-ss). Secondary ASN translation is performed if the secondary ASN restored by PROGRAM RETURN (PR-ss or PROGRAM RETURN to current primary) does not equal the primary ASN restored by PROGRAM RETURN.

PROGRAM CALL with space switching (PC-ss) performs the equivalent of primary ASN translation by obtaining a primary ASN and the address of the corresponding ASN-second-table entry from an entry-table entry.

The ASN-translation process uses two tables, the ASN first table and the ASN second table. They are used to locate the ASN-second-table entry and a third table, the authority table, which is used when ASN authorization is performed.

For the purposes of this translation, the 16-bit ASN is considered to consist of two parts: the ASN-first-table index (AFX) is the leftmost 10 bits of the ASN, and the ASN-second-table index (ASX) is the six rightmost bits. The ASN has the following format:

ASN



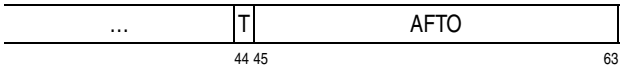
The AFX is used to select an entry from the ASN first table. The origin of the ASN first table is designated by the ASN-first-table origin in control register 14. The ASN-first-table entry contains the origin of the ASN second table. The ASX is used to select an entry from the ASN second table.

As a result of primary ASN translation and during the operation of PROGRAM CALL with space switching, the address of the located ASN-second-table entry (ASTE) is placed in control register 5 as the new primary-ASTE origin (PASTEO).

ASN-Translation Controls

ASN translation is controlled by the ASN-translation-control bit and the ASN-first-table origin, both of which reside in control register 14.

Control Register 14



ASN-Translation Control (T): Bit 44 of control register 14 is the ASN-translation-control bit. This bit provides a mechanism whereby the control program can indicate whether ASN translation can occur while a particular program is being executed, and also whether the execution of PROGRAM CALL with space switching is allowed. Bit 44 must be one to allow completion of these instructions:

- LOAD ADDRESS SPACE PARAMETERS
- PROGRAM CALL with space switching
- PROGRAM RETURN with space switching or when the restored SASN does not equal the restored PASN
- PROGRAM TRANSFER with space switching
- PROGRAM TRANSFER WITH INSTANCE with space switching
- SET SECONDARY ASN

- SET SECONDARY ASN WITH INSTANCE

Otherwise, a special-operation exception is recognized. The ASN-translation-control bit is examined in both the problem and the supervisor states.

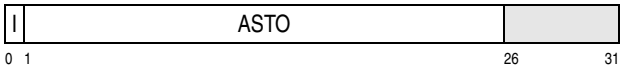
ASN-First-Table Origin (AFTO): Bits 45-63 of control register 14, with 12 zeros appended on the right, form a 31-bit real address that designates the beginning of the ASN first table.

ASN-Translation Tables

The ASN-translation process consists in a two-level lookup using two tables: an ASN first table and an ASN second table. These tables reside in real storage.

ASN-First-Table Entries

An entry in the ASN first table has the following format:



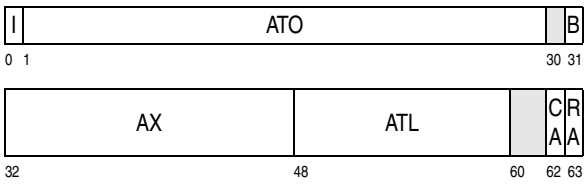
The fields in the entry are allocated as follows:

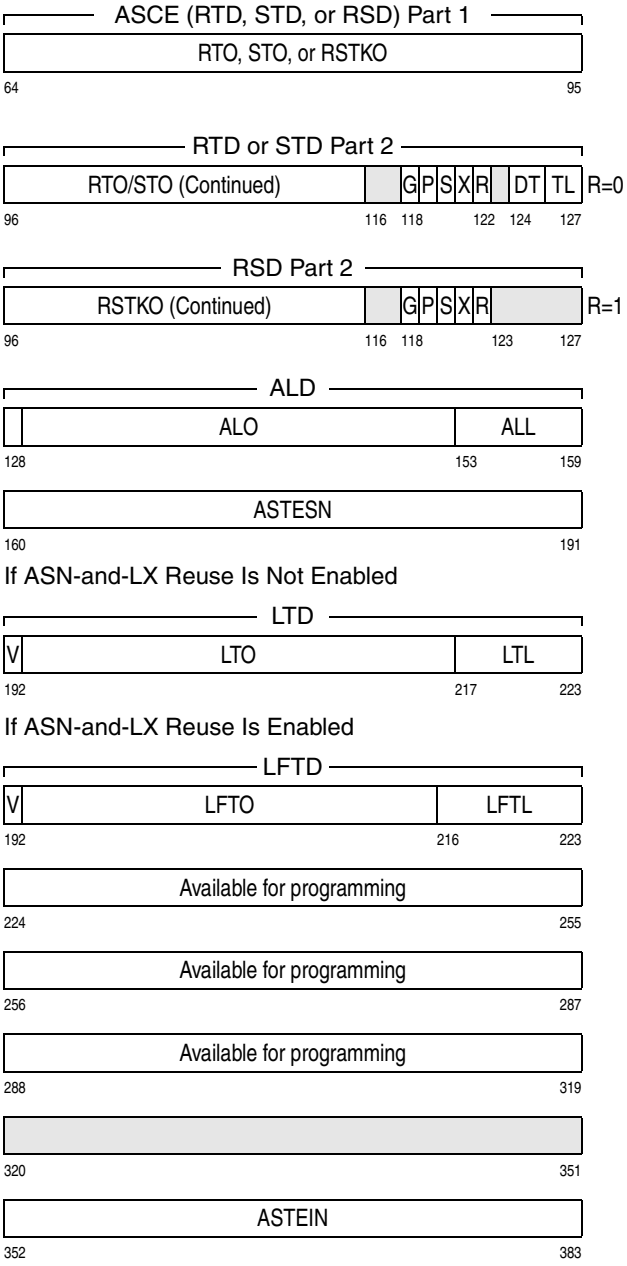
AFX-Invalid Bit (I): Bit 0 controls whether the ASN second table associated with the ASN-first-table entry is available. When bit 0 is zero, ASN translation proceeds by using the designated ASN second table. When the bit is one, the ASN translation cannot continue.

ASN-Second-Table Origin (ASTO): Bits 1-25, with six zeros appended on the right, are used to form a 31-bit real address that designates the beginning of the ASN second table.

ASN-Second-Table Entries

The ASN-second-table entry has a length of 64 bytes, with only the first 48 bytes currently in use. Bytes 0-47 of the entry have the following format:





The fields in bytes 0-47 of the ASN-second-table entry are allocated as follows. Only the fields that are used in or as a result of ASN translation or PROGRAM CALL with space switching are described in detail.

ASX-Invalid Bit (I): Bit 0 controls whether the address space associated with the ASN-second-table entry is available. When bit 0 is zero, ASN translation proceeds. When the bit is one, the ASN translation cannot continue.

Authority-Table Origin (ATO): Bits 1-29, with two zeros appended on the right, are used to form a 31-bit real address that designates the beginning of the authority table.

Base-Space Bit (B): Bit 31 specifies, when one, that the address space associated with the ASN-second-table entry is the base space of a subspace group. Bit 31 is further described in "Subspace-Group ASN-Second-Table Entries" on page 5-68.

Authorization Index (AX): Bits 32-47 are used in ASN authorization as an index to locate the authority bits in the authority table. The AX field is used as a result of primary ASN translation by PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, and, possibly, LOAD ADDRESS SPACE PARAMETERS. It is also used by PROGRAM CALL with space switching. The AX field is ignored after secondary ASN translation.

Authority-Table Length (ATL): Bits 48-59 specify the length of the authority table in units of four bytes, thus making the authority table variable in multiples of 16 entries. The length of the authority table, in units of four bytes, is one more than the ATL value. The contents of the ATL field are used to establish whether the entry designated by a particular AX falls within the authority table.

Controlled-ASN Bit (CA): PROGRAM TRANSFER WITH INSTANCE with space switching and SET SECONDARY ASN WITH INSTANCE with space switching recognize a special-operation exception if bit 62 is one and the CPU is in the problem state at the beginning of the operation. Bit 62 is ignored in the supervisor state.

Reusable-ASN Bit (RA): If the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control, bit 44 of control register 0, PROGRAM TRANSFER with space switching and SET SECONDARY ASN with space switching recognize a special-operation exception if bit 63 is one in the problem or the supervisor state.

Address-Space-Control Element (ASCE): Bits 64-127 are an eight-byte address-space-control element (ASCE) that may be a region-table designation (RTD), a segment-table designation (STD), or a real-space designation (RSD). (The term "region-table designation" is used to mean a region-first-table designation, region-second-table designation, or region-third-table designation.) The ASCE field is used as a result of ASN translation or in PROGRAM CALL with space switching to replace the primary ASCE (PASCE) or the secondary ASCE (SASCE). For PROGRAM CALL with space switching, the ASCE field replaces the PASCE, bits 0-63 of control register 1. For SET SECONDARY ASN, and SET SECONDARY ASN WITH INSTANCE, the ASCE field replaces the SASCE, bits 0-63 of control register 7. Each of these actions may occur independently for LOAD ADDRESS SPACE PARAMETERS. For PROGRAM TRANSFER, and PROGRAM TRANSFER WITH INSTANCE, the ASCE field replaces both the PASCE and the SASCE. For PROGRAM RETURN, as a result of primary ASN translation, the ASCE field replaces the PASCE, and, as a result of secondary ASN translation, the ASCE field replaces the SASCE. The contents of the entire ASCE field are placed in the appropriate control registers without being inspected for validity.

The subspace-group-control bit (G), bit 118 of the ASCE field, indicates, when one, that the ASCE specifies an address space that is the base space or a subspace of a subspace group. The bit is further described in "Subspace-Group ASN-Second-Table Entries" on page 5-68.

Bit 121 (X) of the ASCE field is the space-switch-event-control bit. When, in the space-switching operations of PROGRAM CALL, PROGRAM RETURN, and PROGRAM TRANSFER, and PROGRAM TRANSFER WITH INSTANCE, this bit is one in control register 1 either before or after the execution of the instruction, a program interruption for a space-switch event occurs after the execution of the instruction is completed. A space-switch-event program interruption also occurs after the completion of a SET ADDRESS SPACE CONTROL, SET ADDRESS SPACE CONTROL FAST, or RESUME PROGRAM instruction that changes the translation mode either to or from the home-space mode when this bit is one in either control register 1 or control register 13. When, in LOAD ADDRESS SPACE PARAMETERS, this bit is one during primary ASN translation, this fact is indicated by the condition code.

The real-space-control bit (R), bit 122 of the ASCE field, indicates, when zero, that the ASCE is a region-table or segment-table designation or, when one, that the ASCE is a real-space designation.

When bit 122 is zero, the designation-type-control bits (DT), bits 124 and 125 of the ASCE field, indicate the designation type of the ASCE. A value 11, 10, 01,

or 00 binary of bits 124 and 125 indicates a region-first-table designation, region-second-table designation, region-third-table designation, or segment-table designation, respectively.

The other fields in the ASCE (RTO, STO, P, S, TL, and RSTKO) are described in “Control Register 1” on page 3-42.

Access-List Designation (ALD): The access-list-designation (ALD) field is described in “ASN-Second-Table Entries” on page 5-57.

ASN-Second-Table-Entry Sequence Number (ASTESN): The ASTE-sequence-number (ASTESN) field is described in “ASN-Second-Table Entries” on page 5-57.

Linkage-Table Designation (LTD) or Linkage-First-Table Designation (LFTD): The linkage-table-designation (LTD) or linkage-first-table designation (LFTD) field in the ASN-second-table entry is described in “PC-Number Translation Control” on page 5-34.

ASN-Second-Table-Entry Instance Number (ASTEIN): When the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control in control register 0, bits 352-383 are compared to an ASTEIN specified along with an ASN for use by PROGRAM RETURN or LOAD ADDRESS SPACE PARAMETERS. The ASTEINs must be equal; otherwise, an ASTE-instance exception is recognized by PROGRAM RETURN or condition code 1 or 2 is set by LOAD ADDRESS SPACE PARAMETERS. This comparison and the exception result also occur in the operations of PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE except independent of the ASN-and-LX-reuse control.

Bits 224-319 in the ASN-second-table entry are available for use by programming.

Programming Note: All unused fields in the ASN-second-table entry, including the unused fields in bytes 0-31 and all of bytes 40-43 and 48-63 should be set to zeros. These fields are reserved for future extensions, and programs which place nonzero values in these fields may not operate compatibly on future machines.

ASN-Translation Process

This section describes the ASN-translation process as it is performed during the execution of the space-switching forms of PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, and SET SECONDARY ASN WITH INSTANCE, and also in PROGRAM RETURN when the restored secondary ASN does not equal the restored primary ASN. ASN translation for LOAD ADDRESS SPACE PARAMETERS is the same except that AFX-translation and ASX-translation exceptions do not occur; such conditions are instead indicated by the condition code. Translation of an ASN is performed by means of two tables, an ASN first table and an ASN second table, both of which reside in main storage.

The ASN first index is used to select an entry from the ASN first table. This entry designates the ASN second table to be used.

The ASN second index is used to select an entry from the ASN second table.

If the I bit is one in either the ASN-first-table entry or the ASN-second-table entry, the entry is invalid, and the ASN-translation process cannot be completed. An AFX-translation exception or ASX-translation exception is recognized.

Whenever access to main storage is made during the ASN-translation process for the purpose of fetching an entry from an ASN first table or ASN second table, key-controlled protection does not apply.

The ASN-translation process is shown in Figure 3-13 on page 3-34.

ASN-First-Table Lookup

The AFX portion of the ASN, in conjunction with the ASN-first-table origin, is used to select an entry from the ASN first table.

The 31-bit real address of the ASN-first-table entry is obtained by appending 12 zeros on the right to the AFT origin contained in bit positions 45-63 of control register 14 and adding the AFX portion with two rightmost and 19 leftmost zeros appended. This addition cannot cause a carry into bit position 0. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

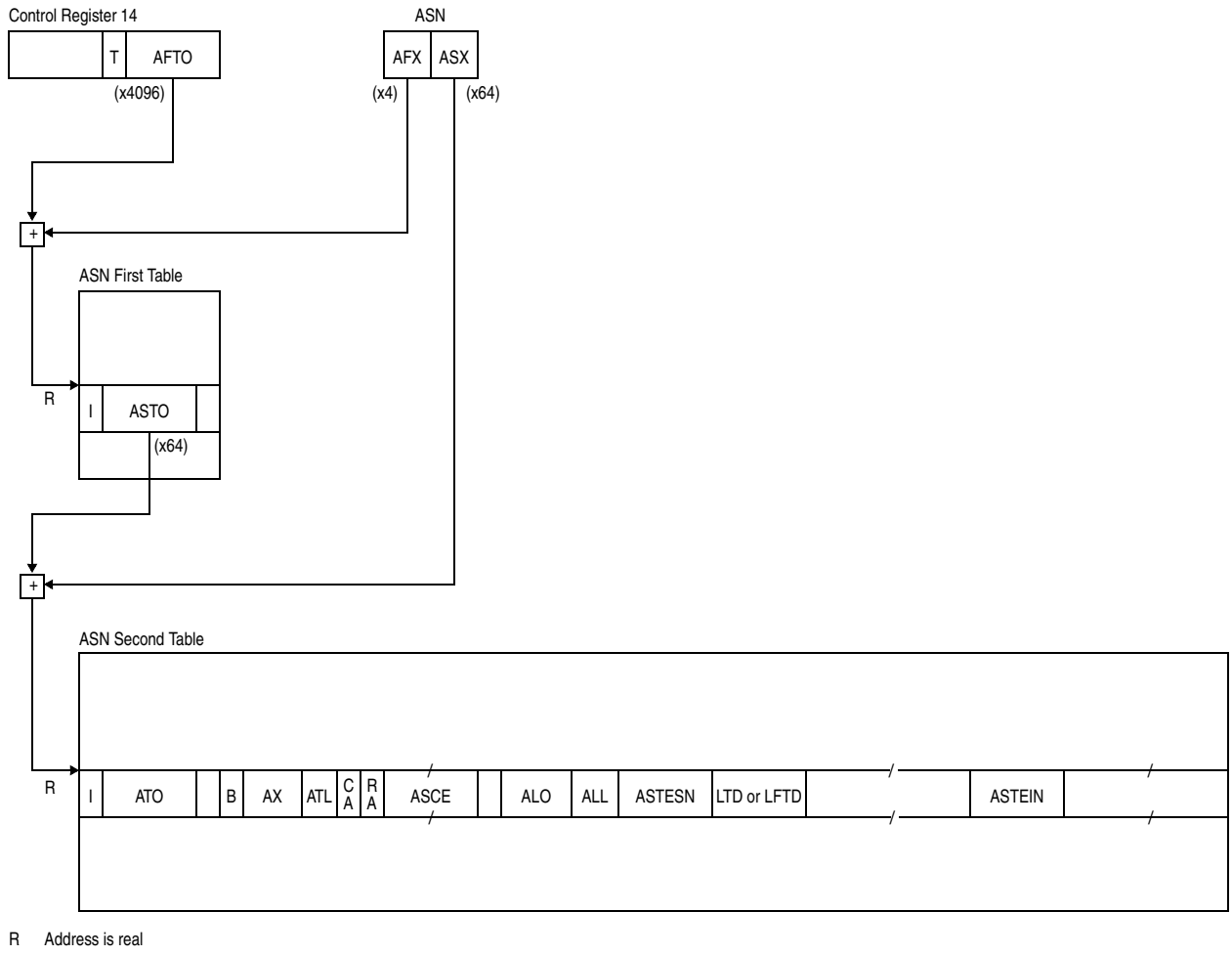


Figure 3-13. ASN Translation

All four bytes of the ASN-first-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address which is generated for fetching the ASN-first-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Bit 0 of the four-byte AFT entry specifies whether the corresponding AST is available. If this bit is one, an AFX-translation exception is recognized. The entry fetched from the AFT is used to access the AST.

ASN-Second-Table Lookup

The ASX portion of the ASN, in conjunction with the ASN-second-table origin contained in the ASN-first-table entry, is used to select an entry from the ASN second table.

The 31-bit real address of the ASN-second-table entry is obtained by appending six zeros on the right to bits 1-25 of the ASN-first-table entry and adding the ASX with six rightmost and 19 leftmost zeros appended. When a carry into bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from 2^{31} - 1 to zero. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

The fetch of the 64 bytes of the ASN-second-table entry appears to be word concurrent as observed by other CPUs, with the leftmost word fetched first. The order in which the remaining 15 words are fetched is unpredictable. The fetch access is not subject to protection. When the storage address which is generated for fetching the ASN-second-table entry

designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Bit 0 of the ASN-second-table entry specifies whether the address space is accessible. If this bit is one, an ASX-translation exception is recognized.

Recognition of Exceptions during ASN Translation

The exceptions which can be encountered during the ASN-translation process are collectively referred to as ASN-translation exceptions. A list of these exceptions and their priorities is given in Chapter 6, “Interruptions”.

ASN Authorization

Note: The ASN-authorization concepts described in following section are applicable only to the z/Architecture architectural mode where dynamic address translation is supported. ASN authorization is not supported in the ESA/390-compatibility mode.

ASN authorization is the process of testing whether the program associated with the current authorization index is permitted to establish a particular address space. The ASN authorization is performed as part of PROGRAM TRANSFER with space switching (PT-ss), PROGRAM TRANSFER WITH INSTANCE with space switching (PTI-ss), SET SECONDARY ASN with space switching (SSAR-ss), and SET SECONDARY ASN WITH INSTANCE with space switching (SSAIR-ss), and may be performed as part of LOAD ADDRESS SPACE PARAMETERS. ASN authorization is performed after the ASN-translation process for these instructions.

ASN authorization is also performed as part of PROGRAM RETURN when the restored secondary ASN does not equal the restored primary ASN. ASN authorization of the restored secondary ASN is performed after ASN translation of the restored secondary ASN.

When performed as part of PT-ss or PTI-ss, the ASN authorization tests whether the ASN can be established as the primary ASN and is called primary-ASN authorization. When performed as part of LOAD ADDRESS SPACE PARAMETERS, PROGRAM RETURN, SSAR-ss, or SSAIR-ss, the ASN authorization tests whether the ASN can be established as the secondary ASN and is called secondary-ASN authorization.

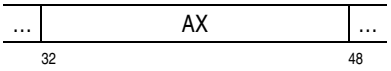
The ASN authorization is performed by means of an authority table in real storage which is designated by the authority-table-origin and authority-table-length fields in the ASN-second-table entry.

ASN-Authorization Controls

ASN authorization uses the authority-table origin and the authority-table length from the ASN-second-table entry, together with an authorization index.

Control Register 4

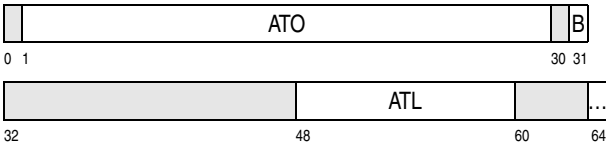
For PT-ss, PTI-ss, SSAR-ss, and SSAIR-ss, the current contents of control register 4 include the authorization index. For LOAD ADDRESS SPACE PARAMETERS and PROGRAM RETURN, the value which will become the new contents of control register 4 is used. The register has the following format:



Authorization Index (AX): Bits 32-47 of control register 4 are used as an index to locate the authority bits in the authority table.

ASN-Second-Table Entry

The ASN-second-table entry which is fetched as part of the ASN translation process contains information which is used to designate the authority table. An entry in the ASN second table has the following format:



Authority-Table Origin (ATO): Bits 1-29, with two zeros appended on the right, are used to form a 31-bit real address that designates the beginning of the authority table.

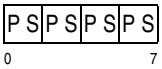
Authority-Table Length (ATL): Bits 48-59 specify the length of the authority table in units of four bytes, thus making the authority table variable in multiples of 16 entries. The length of the authority table, in units of four bytes, is equal to one more than the ATL

value. The contents of the length field are used to establish whether the entry designated by the authorization index falls within the authority table.

Authority-Table Entries

The authority table consists of entries of two bits each; accordingly, each byte of the authority table contains four entries in the following format:

The fields are allocated as follows:



Primary Authority (P): The left bit of an authority-table entry controls whether the program with the authorization index corresponding to the entry is permitted to establish the address space as a primary address space. If the P bit is one, the establishment is permitted. If the P bit is zero, the establishment is not permitted.

Secondary Authority (S): The right bit of an authority-table entry controls whether the program with the corresponding authorization index is permitted to establish the address space as a secondary address space. If the S bit is one, the establishment is permitted. If the S bit is zero, the establishment is not permitted.

The authority table is also used in the extended-authorization process, as part of access-register translation. Extended authorization is described in "Authorizing the Use of the Access-List Entry" on page 5-63.

ASN-Authorization Process

This section describes the ASN-authorization process as it is performed during the execution of PROGRAM TRANSFER with space switching, PROGRAM TRANSFER WITH INSTANCE with space switching, SET SECONDARY ASN with space switching, and SET SECONDARY ASN WITH INSTANCE with space switching. For these two instructions, the ASN-authorization process is performed by using the authorization index currently in control register 4. Secondary authorization for PROGRAM RETURN, when the restored secondary ASN does not equal the restored primary ASN, and for LOAD ADDRESS SPACE PARAMETERS is the same, except that the value which will become the new contents of control register 4 is used for the authorization index. Also, for LOAD ADDRESS SPACE PARAMETERS, a secondary-authority exception does not occur. Instead, such a condition is indicated by the condition code.

The ASN-authorization process is performed by using the authorization index, in conjunction with the authority-table origin and length from the AST entry, to select an authority-table entry. The entry is fetched, and either the primary- or the secondary-authority bit is examined, depending on whether the primary- or secondary-ASN-authorization process is being performed. The ASN-authorization process is shown in Figure 3-14.

Authority-Table Lookup

The authorization index, in conjunction with the authority-table origin contained in the ASN-second-table entry, is used to select an entry from the authority table.

The authorization index is contained in bit positions 32-47 of control register 4.

Bit positions 1-29 of the AST entry contain the leftmost 29 bits of the 31-bit real address of the authority table (ATO), and bit positions 48-59 contain the length of the authority table (ATL).

The 31-bit real address of a byte in the authority table is obtained by appending two zeros on the right to the authority-table origin and adding the 14 leftmost bits of the authorization index with 17 zeros appended on the left. When a carry into bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from 2^{31} - 1 to zero. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

As part of the authority-table-entry-lookup process, bits 0-11 of the authorization index are compared against the authority-table length. If the compared portion is greater than the authority-table length, a primary-authority exception or secondary-authority exception is recognized for PT-ss, PTI-ss, SSAR-ss, or SSAIR-ss, respectively. For LOAD ADDRESS SPACE PARAMETERS, when the authority-table length is exceeded, condition code 2 is set.

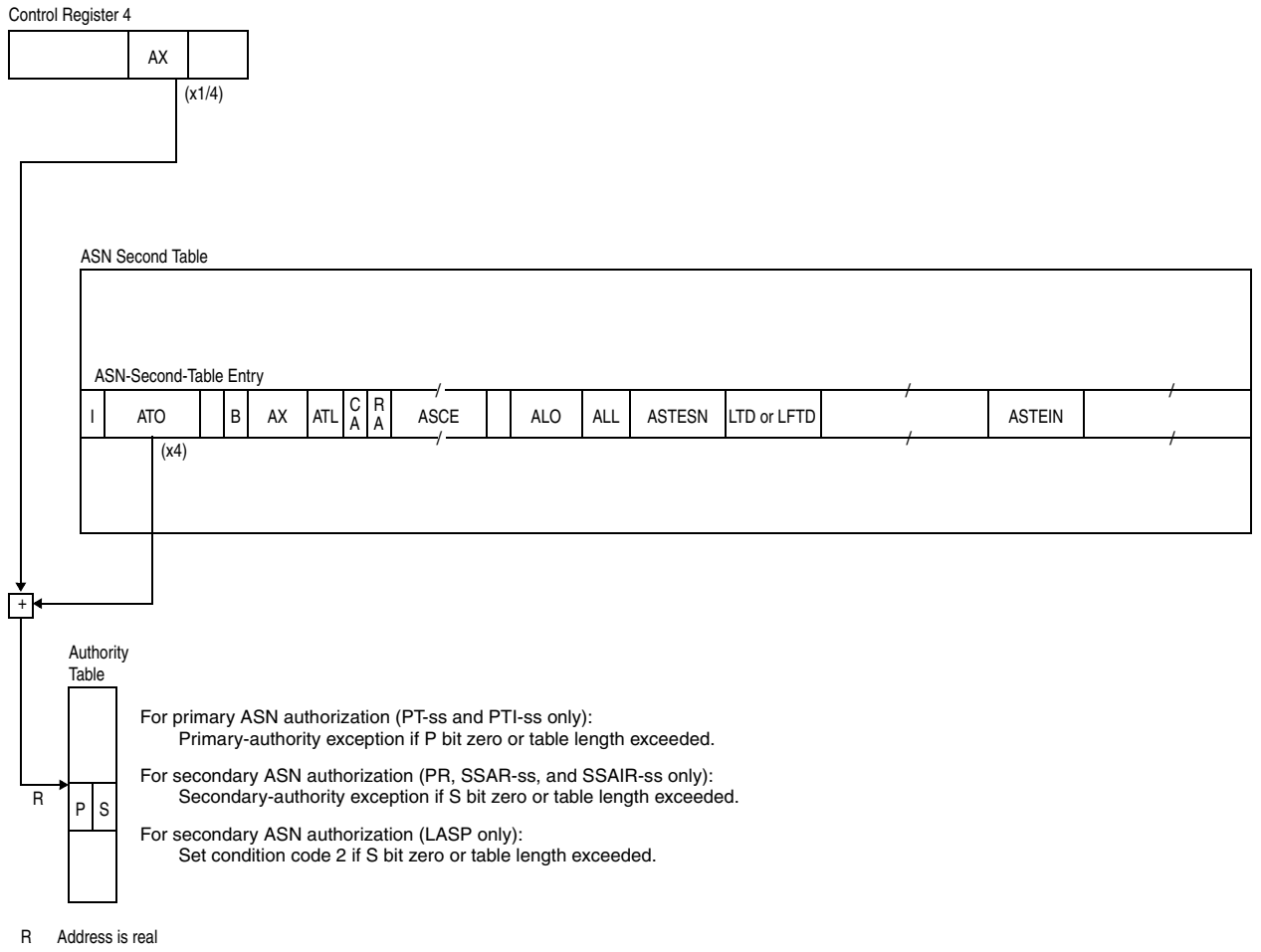


Figure 3-14. ASN Authorization

The fetch access to the byte in the authority table is not subject to protection. When the storage address which is generated for fetching the byte designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

The byte contains four authority-table entries of two bits each. The rightmost two bits of the authorization index, bits 46 and 47 of control register 4, are used to select one of the four entries. The left or right bit of the entry is then tested, depending on whether the authorization test is for a primary ASN or a secondary ASN. The following table shows the bit which is selected from the byte as a function of bits 46 and 47 of the authorization index and the instruction PT-ss, PTI-ss, SSAR-ss, SSAIR-ss, PROGRAM RETURN, or LOAD ADDRESS SPACE PARAMETERS.

Authorization-Index Bits		Bit Selected from Authority-Table Byte for Test	
46	47	P Bit (PT-ss and PTI-ss)	S Bit (SSAR-ss, SSAIR-ss, PR, or LASP)
0	0	0	1
0	1	2	3
1	0	4	5
1	1	6	7

If the selected bit is one, the ASN is authorized, and the appropriate fields in the AST entry are loaded into the appropriate control registers. If the selected bit is zero, the ASN is not authorized, and a primary-authority exception is recognized for PT-ss or PTI-ss, or a secondary-authority exception is recognized for SSAR-ss, SSAIR-ss, or PROGRAM RETURN. For

LOAD ADDRESS SPACE PARAMETERS, when the ASN is not authorized, condition code 2 is set.

Recognition of Exceptions during ASN Authorization

The exceptions which can be encountered during the primary- and secondary-ASN-authorization processes and their priorities are described in the definitions of the instructions in which ASN authorization is performed.

Programming Note: The primary- and secondary-authority exceptions cause nullification in order to permit dynamic modification of the authority table. Thus, when an address space is created or "swapped in," the authority table can first be set to all zeros and the appropriate authority bits set to one only when required.

Dynamic Address Translation

Note: The dynamic-address translation concepts described in following section are applicable only to the z/Architecture architectural mode where DAT is supported. DAT is not supported in the ESA/390-compatibility mode.

Dynamic address translation (DAT) provides the ability to interrupt the execution of a program at an arbitrary moment, record it and its data in auxiliary storage, such as a direct-access storage device, and at a later time return the program and the data to different main-storage locations for resumption of execution. The transfer of the program and its data between main and auxiliary storage may be performed piecemeal, and the return of the information to main storage may take place in response to an attempt by the CPU to access it at the time it is needed for execution. These functions may be performed without change or inspection of the program and its data, do not require any explicit programming convention for the relocated program, and do not disturb the execution of the program except for the time delay involved.

With appropriate support by an operating system, the dynamic-address-translation facility may be used to provide to a user a system wherein storage appears to be larger than the main storage which is available in the configuration. This apparent main storage is referred to as virtual storage, and the addresses used to designate locations in the virtual storage are referred to as virtual addresses. The virtual storage of a user may far exceed the size of the main storage which is available in the configuration and normally is maintained in auxiliary storage. The virtual storage is considered to be composed of blocks of addresses, called pages. Only the most recently referred-to pages of the virtual storage are assigned to occupy blocks of physical main storage. As the user refers to pages of virtual storage that do not appear in main storage, they are brought in to replace pages in main storage that are less likely to be needed. The swapping of pages of storage may be performed by the operating system without the user's knowledge.

The sequence of virtual addresses associated with a virtual storage is called an address space. With appropriate support by an operating system, the dynamic-address-translation facility may be used to provide a number of address spaces. These address spaces may be used to provide degrees of isolation between users. Such support can consist of a completely different address space for each user, thus providing complete isolation, or a shared area may be provided by mapping a portion of each address space to a single common storage area. Also, instructions are provided which permit a semiprivileged program to access more than one such address space. Dynamic address translation provides for the translation of virtual addresses from multiple different address spaces without requiring that the translation parameters in the control registers be changed. These address spaces are called the primary address space, secondary address space, and AR-specified address spaces. A privileged program can also cause the home address space to be accessed.

In the process of replacing blocks of main storage by new information from an external medium, it must be determined which block to replace and whether the block being replaced should be recorded and preserved in auxiliary storage. To aid in this decision process, a reference bit and a change bit are associated with the storage key.

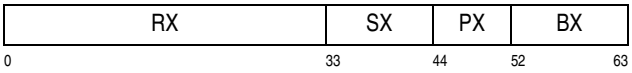
Dynamic address translation may be specified for instruction and data addresses generated by the CPU but is not available for the addressing of data and of CCWs, IDAWs, and MIDAWs in I/O operations. The CCW-indirect-data-addressing facility and modified-CCW-indirect-data-addressing facilities are provided to aid I/O operations in a virtual-storage environment.

Address computation can be carried out in the 24-bit, 31-bit, or 64-bit addressing mode. When address computation is performed in the 24-bit or 31-bit addressing mode, 40 or 33 zeros, respectively, are appended on the left to form a 64-bit address. Therefore, the resultant logical address is always 64 bits in length. The real or absolute address that is formed by dynamic address translation, and the absolute address that is then formed by prefixing, are always 64 bits in length.

Dynamic address translation is the process of translating a virtual address during a storage reference into the corresponding real or absolute address. The virtual address may be a primary virtual address, secondary virtual address, AR-specified virtual address, or home virtual address. These addresses are translated by means of the primary, the secondary, an AR-specified, or the home address-space-control element, respectively. After selection of the appropriate address-space-control element, the translation process is the same for all of the four types of virtual address. An address-space-control element may be a segment-table designation specifying a 2 G-byte address space, a region-table designation specifying a 4 T-byte, 8 P-byte, or 16 E-byte space, or a real-space designation specifying a 16 E-byte space. (The letters K, M, G, T, P, and E represent kilo, 2^{10}, mega, 2^{20}, giga, 2^{30}, tera, 2^{40}, peta, 2^{50}, and exa, 2^{60}, respectively.) A segment-table designation or region-table designation causes translation to be performed by means of tables established by the operating system in real or absolute storage. A real-space designation causes the virtual address simply to be treated as a real address, without the use of tables in storage.

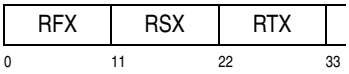
In the process of translation when using a segment-table designation or a region-table designation, three types of units of information are recognized — regions, segments, and pages. A region is a block of sequential virtual addresses spanning 2G bytes and beginning at a 2 G-byte boundary. A segment is a block of sequential virtual addresses spanning 1M bytes and beginning at a 1 M-byte boundary. A page is a block of sequential virtual addresses spanning 4K bytes and beginning at a 4 K-byte boundary.

The virtual address, accordingly, is divided into four principal fields. Bits 0-32 are called the region index (RX), bits 33-43 are called the segment index (SX), bits 44-51 are called the page index (PX), and bits 52-63 are called the byte index (BX). The virtual address has the following format:



As determined by its address-space-control element, a virtual address space may be a 2 G-byte space consisting of one region, or it may be up to a 16 E-byte space consisting of up to 8G regions. The RX part of a virtual address applying to a 2 G-byte address space must be all zeros; otherwise, an exception is recognized.

The RX part of a virtual address is itself divided into three fields. Bits 0-10 are called the region first index (RFX), bits 11-21 are called the region second index (RSX), and bits 22-32 are called the region third index (RTX). Bits 0-32 of the virtual address have the following format:



A virtual address in which the RTX is the leftmost significant part (a 42-bit address) is capable of addressing 4T bytes (2K regions), one in which the RSX is the leftmost significant part (a 53-bit address) is capable of addressing 8P bytes (4M regions), and one in which the RFX is the leftmost significant part (a 64-bit address) is capable of addressing 16E bytes (8G regions).

A virtual address in which the RX is always zero can be translated into real addresses by means of one or two translation tables, as follows:

- When enhanced DAT does not apply, or when enhanced DAT applies but the STE-format control, bit 53 of the segment-table entry, is zero, the virtual address can be translated by means of a segment table and a page table.
- When enhanced DAT applies and the STE-format control of the segment-table entry is one, the virtual address can be translated by means of a segment table only.

If the RX may be nonzero, from one to three additional translation tables are required, as follows. If the RFX may be nonzero, a region first table, region second table, and region third table are required. If the RFX is always zero but the RSX may be nonzero, a

region second table and region third table are required. If the RFX and RSX are always zero but the RTX may be nonzero, a region third table is required. An exception is recognized if the address-space-control element for an address space does not designate the highest level of table (beginning with the region first table and continuing downward to the segment table) needed to translate a reference to the address space.

A region first table, region second table, or region third table is sometimes referred to simply as a region table. Similarly, a region-first-table designation, region-second-table designation, or region-third-table designation is sometimes referred to as a region-table designation.

The region, segment, and, when applicable, page tables reflect the current assignment of real or absolute storage.

When EDAT-1 does not apply, or when EDAT-1 applies but the STE-format control, bit 53 of the segment-table entry, is zero, the assignment of real storage occurs in units of pages, the real locations being assigned contiguously within a page. The pages need not be adjacent in real storage even though assigned to a set of sequential virtual addresses.

When EDAT-1 applies and the STE-format control of the segment-table entry is one, the assignment of real storage occurs in units of segments, the absolute locations being assigned contiguously within a segment. The segments need not be adjacent in absolute storage even though assigned to a set of sequential virtual addresses.

Similarly, when EDAT-2 applies and the RTTE-format control of the region-third-table entry is one, the assignment of real storage occurs in units of regions, the absolute locations being assigned contiguously within a region. The regions need not be adjacent in absolute storage even though assigned to a set of sequential virtual addresses.

To improve performance, translation normally is performed by means of table copies maintained in a special buffer called the translation-lookaside buffer (TLB). The TLB may also contain entries that provide the virtual-equals-real translation specified by a real-space designation.

Translation Control

Address translation is controlled by three bits in the PSW and by a set of bits referred to as the translation parameters. The translation parameters are in control registers 0, 1, 7, and 13. Additional controls are located in the translation tables.

Additional controls are provided as described in Chapter 5, “Program Execution.” These controls determine whether the contents of each access register can be used to obtain an address-space-control element for use by DAT.

Translation Modes

The three bits in the PSW that control dynamic address translation are bit 5, the DAT-mode bit, and bits 16 and 17, the address-space-control bits. When the DAT-mode bit is zero, then DAT is off, and the CPU is in the real mode. When the DAT-mode bit is one, then DAT is on, and the CPU is in the translation mode designated by the address-space-control bits: 00 designates the primary-space mode, 01 designates the access-register mode, 10 designates the secondary-space mode, and 11 designates the home-space mode.

For certain instructions such as LOAD PAGE TABLE ENTRY ADDRESS, LOAD REAL ADDRESS and STORE REAL ADDRESS, and for monitor-event counting operations, dynamic address translation is performed even when the DAT-mode bit in the PSW is off.

The various modes are shown in Figure 3-15, along with the handling of addresses in each mode.

PSW Bit					Handling of Addresses	
5	16	17	DAT	Mode	Instruction Addresses	Logical Addresses
0	0	0	Off	Real mode	Real	Real
0	0	1	Off	Real mode	Real	Real
0	1	0	Off	Real mode	Real	Real
0	1	1	Off	Real mode	Real	Real
1	0	0	On	Primary-space mode	Primary virtual	Primary virtual
1	0	1	On	Access-register mode	Primary virtual	AR-specified virtual
1	1	0	On	Secondary-space mode	Primary virtual	Secondary virtual

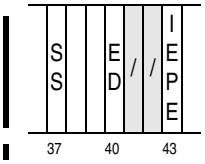
Figure 3-15. Translation Modes

PSW Bit			DAT	Mode	Handling of Addresses	
					Instruction Addresses	Logical Addresses
5	16	17				
1	1	1	On	Home-space mode	Home virtual	Home virtual

Figure 3-15. Translation Modes

Control Register 0

Bit 37 is provided in control register 0 for use in controlling dynamic address translation. When the enhanced-DAT facility is installed, bit 40 is also provided for use in controlling dynamic-address translation. The bit assignments are as follows:



Secondary-Space Control (SS): Bit 37 of control register 0 is the secondary-space-control bit. When this bit is zero and execution of MOVE TO PRIMARY, MOVE TO SECONDARY, or SET ADDRESS SPACE CONTROL is attempted, a special-operation exception is recognized. A special-operation exception is also recognized when this bit is zero, execution of MOVE WITH OPTIONAL SPECIFICATIONS is attempted, and the operand-access control for either operand designates the secondary space. When this bit is one, it indicates that the region table or segment table designated by the secondary address-space-control element has been established.

Enhanced-DAT-Enablement Control (ED): When the enhanced-DAT facility 1 is installed, bit 40 of control register 0 is the enhanced-DAT-enablement control. When this bit is zero, dynamic address translation proceeds as though the enhanced-DAT facility was not installed. When the bit is one, the following conditions apply:

- The DAT-protection bit is defined in bit position 54 of each region-table entry.
- The STE-format control is defined in bit position 53 of the segment-table entry. When the STE-format control is zero, bits 0-52 of the segment-table entry are used to locate the page table (as occurs when the enhanced-DAT facility 1 is not installed or enabled)

  When the STE-format control is one, the following apply:

  – Bits 0-43 of the segment-table entry form the segment-frame absolute address. There is no designation of a page table, and no page-table entries are used.

  – Bit 47 of the segment-table entry determines the validity of the access-control bits and fetch-protection bit (in bits 48-52 of the STE).

  – Bits 48-52 of the segment-table entry contain access-control bits and a fetch-protection bit for the segment.

When the enhanced-DAT facility 1 is not installed, bit 40 of control register 0 is reserved and should contain zero; otherwise, the program may not operate compatibly in the future.

When the enhanced-DAT-facility 2 is installed, and the enhanced-DAT-enablement control is one, the RTTE-format control is defined in bit position 53 of the region-third-table entry. When the RTTE-format control is zero, bits 0-51 of the region-third-table entry are used to locate the segment table (as occurs when the enhanced-DAT facility 2 is not installed). When the RTTE-format control is one, the following apply:

- Bits 0-32 of the region-third-table entry form the region-frame absolute address. There is no designation of a segment table, and no segment-table entries are used.
- Bit 47 of the region-third-table entry determines the validity of the access-control bits and fetch-protection bit (in bits 48-52 of the RTTE).
- Bits 48-52 of the region-third-table entry contain access-control bits and a fetch-protection bit for the region.

When the enhanced-DAT facility 2 is installed, the enhanced-DAT facility 1 is also installed.

Enhanced-DAT Terminology: For the purpose of brevity, the following terms are used in conjunction with the enhanced-DAT facilities:

- The term "*EDAT-1 applies*" refers to the case where all of the following are true:

  – The enhanced-DAT facility 1 is installed.

  - The enhanced-DAT-enablement control, bit 40 of control register 0, is one.
  - The address is translated by means of DAT-table entries.

- The term "*EDAT-2 applies"* refers to the case where both of the following are true:
  - The enhanced-DAT facility 2 is installed.
  - EDAT-1 applies.

- The term "*EDAT-1 does not apply*" refers to the case where any of the following are true:
  - The enhanced-DAT facility 1 is not installed.
  - The enhanced-DAT facility 1 is installed, but the enhanced-DAT-enablement control is zero.
  - The address is not translated by means of DAT-table entries (that is, DAT is off and is not being used implicitly; DAT is on, but the ASCE designates a real space; or the instruction uses a real address such as LOAD USING REAL ADDRESS or STORE USING REAL ADDRESS).
  - A real address is implicitly used, for example in the handling of an interruption, CPU logout, or fetching of table entries for ART, ASN translation, ASN authorization, DAT, or PC-number translation.

- The term "*EDAT-2 does not apply*" refers to the case where either of the following is true:
  - EDAT-1 does not apply
  - The enhanced-DAT facility 2 is not installed.

Instruction-Execution-Protection-Enablement (IEPE) Control: When the instruction-execution-protection facility is installed, bit 43 of control register 0 is the instruction-execution-protection-enablement control. When the instruction-execution-protection facility is not installed, or when the facility is installed and this bit is zero, instruction-execution protection does not apply.

When the instruction-execution-protection facility is installed, the IEPE control is one, and instructions are fetched using an ASCE in which the real-space control is zero, the execution of instructions is subject to the instruction-execution-protection (IEP) control in the leaf table entry used in the translation. (See "Translation Tables" on page 3-45 for a definition of "leaf.")

Programming Notes:

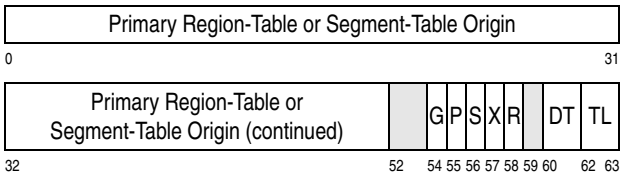
1. Before DAT is enabled or implicitly performed (such as in LOAD REAL ADDRESS), the program should ensure that the enhanced-DAT-enablement control is consistent with the enhanced-DAT-enablement control in any other CPU in the configuration that may be performing DAT for the same address space.
2. DAT should be disabled before altering the enhanced-DAT-enablement control, and the TLB of the CPU should be purged before re-enabling DAT.

Failure to follow these rules may result in unpredictable results, including the possibility of a delayed-access machine-check condition being recognized.

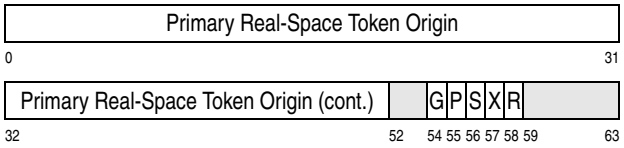
Control Register 1

Control register 1 contains the primary address-space-control element (PASCE). The register has one of the following two formats, depending on the real-space-control bit (R) in the register:

Primary Region-Table or Segment-Table Designation (R=0)



Primary Real-Space Designation (R=1)



The fields in the primary address-space-control element are allocated as follows:

Primary Region-Table or Segment-Table Origin: Bits 0-51 of the primary region-table or segment-table designation in control register 1, with 12 zeros appended on the right, form a 64-bit address that designates the beginning of the primary region table or segment table. It is unpredictable whether the address is real or absolute. This table is called the primary region table or segment table since it is used

to translate virtual addresses in the primary address space.

Primary Subspace-Group Control (G): Bit 54 of control register 1, when one, indicates that the address space specified by the PASCE is the base space or a subspace of a subspace group. When bit 54 is zero, the address space is not in a subspace group.

Primary Private-Space Control (P): If bit 55 of control register 1 is one, then (1) a one value of the common-segment bit in a translation-lookaside-buffer (TLB) representation of a segment-table entry prevents the entry and any TLB page-table copy it designates from being used when translating references to the primary address space, even with a match between the table or token origin in control register 1 and the table origin in the TLB entry, (2) low-address protection and fetch-protection override do not apply to the primary address space; and (3) a translation-specification exception is recognized if a reference to the primary address space is translated by means of a segment-table entry in storage and the common-segment bit is one in the entry. Item 2 in the above list applies even when the contents of control register 1 are a real-space designation.

When EDAT-2 applies and bit 55 of control register 1 is one, then (1) a one value of the common-region bit in a translation-lookaside-buffer (TLB) representation of a region-third-table entry prevents the entry and any TLB segment- and page-table copy it designates from being used when translating references to the primary address space, even with a match between the table or token origin in control register 1 and the table origin in the TLB entry, (2) low-address protection and fetch-protection override do not apply to the primary address space; and (3) a translation-specification exception is recognized if a reference to the primary address space is translated by means of a region-third-table entry in storage and the common-region bit is one in the entry. Item 2 in the above list applies even when the contents of control register 1 are a real-space designation.

Programming Note: With respect to item 1 in the above lists, when the contents of control register 1 are a real-space designation, a one value of the common-segment bit in a TLB representation of a segment-table entry prevents the entry and any TLB page-table copy it designates from being used regardless of the value of the private-space control in the real-space designation. Similarly, when EDAT-2 applies and the contents of control register 1 are a real-space designation, a one value of the common-region bit in a TLB representation of a region-third-table entry prevents the entry and any TLB segment- and page-table copies it designates from being used regardless of the value of the private-space control in the real-space designation.

Primary Storage-Alteration-Event Control (S): When the storage-alteration-space control in control register 9 is one, bit 56 of control register 1 specifies, when one, that the primary address space is one for which storage-alteration events can occur. Bit 56 is examined when the PASCE is used to perform dynamic-address translation for a storage-operand store reference. Bit 56 is ignored when the storage-alteration-space control is zero.

Primary Space-Switch-Event Control (X): When bit 57 of control register 1 is one:

- A space-switch-event program interruption occurs when execution of the space-switching form of PROGRAM CALL (PC-ss), PROGRAM RETURN (PR-ss), or PROGRAM TRANSFER (PT-ss) is completed. The interruption occurs if bit 57 is one either before or after the operation.
- A space-switch-event program interruption occurs upon completion of a RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST instruction that changes the address space from which instructions are fetched either to or from the home address space; that is, when instructions are fetched from the home address space either before or after the operation but not both before and after the operation.
- Condition code 3 is set by LOAD ADDRESS SPACE PARAMETERS.

Primary Real-Space Control (R): If bit 58 of control register 1 is zero, the register contains a region-table or segment-table designation. If bit 58 is one, the register contains a real-space designation. When bit 58 is one, a one value of the common-segment bit in a translation-lookaside-buffer (TLB) representation of a segment-table entry prevents the entry and any TLB page-table copy it designates from being used when translating references to the primary address space, even with a match between the token origin in control register 1 and the table origin in the TLB entry. Similarly, when EDAT-2 applies and bit 58 is

one, a one value of the common-region bit in a translation-lookaside-buffer (TLB) representation of a region-third-table entry prevents the entry and any TLB segment- and page-table copies it designates from being used when translating references to the primary address space, even with a match between the token origin in control register 1 and the table origin in the TLB entry.

Primary Designation-Type Control (DT): When R is zero, the type of table designation in control register 1 is specified by bits 60 and 61 in the register, as follows:

Bits 60 and 61	Designation Type
11	Region-first-table
10	Region-second-table
01	Region-third-table
00	Segment-table

When R is zero, bits 60 and 61 must be 11 binary when an attempt is made to use the PASCE to translate a virtual address in which the leftmost one bit is in bit positions 0-10 of the address. Similarly, bits 60 and 61 must be 11 or 10 binary when the leftmost one bit is in bit positions 11-21 of the address, and they must be 11, 10, or 01 binary when the leftmost one bit is in bit positions 22-32 of the address. Otherwise, an ASCE-type exception is recognized.

Primary Region-Table or Segment-Table Length (TL): Bits 62 and 63 of the primary region-table designation or segment-table designation in control register 1 specify the length of the primary region table or segment table in units of 4,096 bytes, thus making the length of the region table or segment table variable in multiples of 512 entries. The length of the primary region table or segment table, in units of 4,096 bytes, is one more than the TL value. The contents of the length field are used to establish whether the portion of the virtual address (RFX, RSX, RTX, or SX) to be translated by means of the table designates an entry that falls within the table.

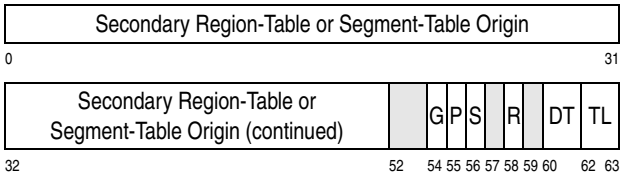
Primary Real-Space Token Origin: Bits 0-51 of the primary real-space designation in control register 1, with 12 zeros appended on the right, form a 64-bit address that may be used in forming and using TLB entries that provide a virtual-equals-real translation for references to the primary address space. Although this address is used only as a token and is not used to perform a storage reference, it still must be a valid address; otherwise, an incorrect TLB entry may be used when the contents of control register 1 are used.

The following bits of control register 1 are not assigned and are ignored: bits 52, 53, and 59 if the register contains a region-table designation or segment-table designation, and bits 52, 53 and 59-63 if the register contains a real-space designation.

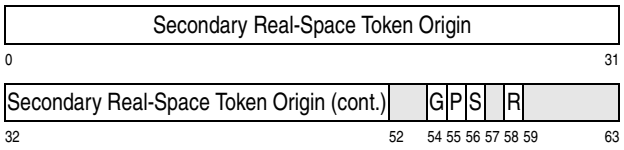
Control Register 7

Control register 7 contains the secondary address-space-control element (SASCE). The register has one of the following two formats, depending on the real-space-control bit (R) in the register:

Secondary Region-Table or Segment-Table Designation (R=0)



Secondary Real-Space Designation (R=1)



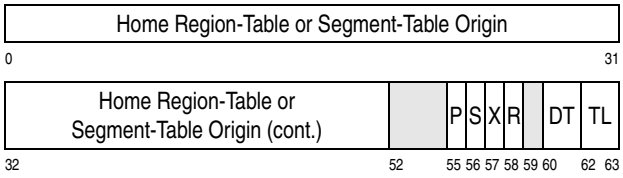
The secondary region-table origin, secondary segment-table origin, secondary subspace-group control (G), secondary private-space control (P), secondary storage-alteration-event control (S), secondary real-space control (R), secondary designation-type control (DT), secondary region-table or segment-table length (TL), and secondary real-space token origin in control register 7 are defined the same as the fields in the same bit positions in control register 1, except that control register 7 applies to the secondary address space.

The following bits of control register 7 are not assigned and are ignored: bits 52, 53, 57, and 59 if the register contains a region-table designation or segment-table designation, and bits 52, 53, 57, and 59-63 if the register contains a real-space designation.

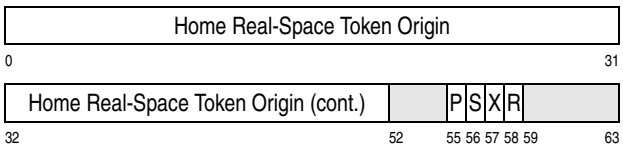
Control Register 13

Control register 13 contains the home address-space-control element (HASCE). The register has one of the following two formats, depending on the real-space-control bit (R) in the register:

Home Region-Table or Segment-Table Designation (R=0)



Home Real-Space Designation (R=1)



Home Space-Switch-Event Control (X): When bit 57 of control register 13 is one, a space-switch-event program interruption occurs upon completion of a RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST instruction that changes the address space from which instructions are fetched either to or from the home address space; that is, when instructions are fetched from the home address space either before or after the operation but not both before and after the operation.

The home region-table origin, home segment-table origin, home private-space control (P), home storage-alteration-event control (S), home real-space control (R), home designation-type control (DT), home region-table or segment-table length (TL), and home real-space token origin in control register 13 are defined the same as the fields in the same bit positions in control register 1, except that control register 13 applies to the home address space.

The following bits of control register 13 are not assigned and are ignored: bits 52-54 and 59 if the register contains a region-table designation or segment-table designation, and bits 52-54 and 59-63 if the register contains a real-space designation.

Programming Notes:

1. The validity of the information loaded into a control register, including that pertaining to dynamic address translation, is not checked at the time the register is loaded. This information is checked and the program exception, if any, is indicated at the time the information is used.

2. The information pertaining to dynamic address translation is considered to be used when an instruction is executed with DAT on, or when the enhanced-monitor counting array is accessed, or when COMPARE AND REPLACE DAT TABLE ENTRY, INVALIDATE DAT TABLE ENTRY, INVALIDATE PAGE TABLE ENTRY, LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, or STORE REAL ADDRESS is executed. The information is not considered to be used when the PSW specifies translation but an I/O, external, restart, or machine-check interruption occurs before an instruction is executed, or when the PSW specifies the wait state.

Translation Tables

When the address-space-control element (ASCE) used in a translation is a region-first-table designation, the translation process consists in a five-level lookup using five tables: a region first table, a region second table, a region third table, a segment table, and a page table. These tables reside in real or absolute storage. When the ASCE is a region-second-table designation, region-third-table designation, or segment-table designation, the lookups in the levels of tables above the designated level are omitted, and the higher-level tables themselves are omitted.

Notes:

1. The terms *higher* and *lower* are used to describe the level of DAT tables or table entries, either in storage or in the translation-lookaside buffer. A higher-level table entry maps a larger portion of virtual storage than a lower-level table entry. Thus, a region-first-table entry is the highest level of DAT-table entries, and a page-table entry is the lowest level.

2. Higher DAT-table entries (if any) may be referred to as *limb*-table entries. Limb-table entries comprise region-first-table entries, region-second-table entries, region-third-table entries in which

the format control is zero, and segment-table entries in which the format control is zero.

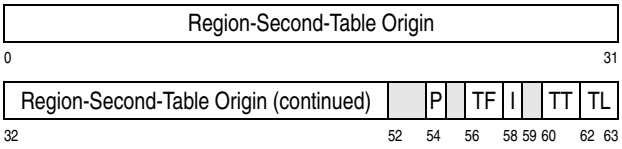
The lowest (or only) DAT-table entry used in a translation is referred to as a *leaf*-table entry. Leaf-table entries comprise region-third-table entries and segment-table entries in which the format control is one, and page-table entries.

Region-Table Entries

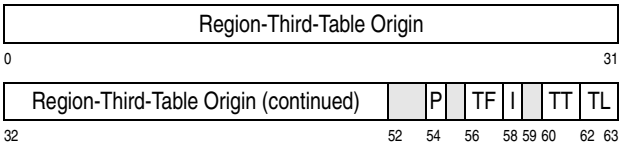
The term "region-table entry" means a region-first-table entry, region-second-table entry, or region-third-table entry.

The entries fetched from the region first table, region second table, and region third table have the following formats. The level (first, second, or third) of the table containing an entry is identified by the table-type (TT) bits in the entry.

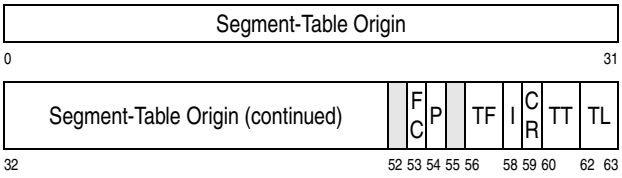
Region-First-Table Entry (TT=11)



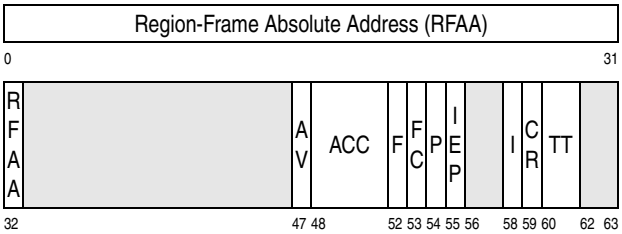
Region-Second-Table Entry (TT=10)



Region-Third-Table Entry (TT=01, FC=0)



Region-Third-Table Entry (TT=01, FC=1)



The fields in the three levels of region-table entries are allocated as follows:

Region-Second-Table Origin, Region-Third-Table Origin, and Segment-Table Origin: A region-first-table entry contains a region-second-table origin. A region-second-table entry contains a region-third-table origin. When EDAT-2 does not apply, or when EDAT-2 applies and the RTTE-format control in the region-third table entry is zero, a region-third-table entry contains a segment-table origin. The following description applies to each of the three origins. Bits 0-51 of the entry, with 12 zeros appended on the right, form a 64-bit address that designates the beginning of the next-lower-level table. It is unpredictable whether the address is real or absolute.

Region-Frame Absolute Address (RFAA): When EDAT-2 applies and the RTTE-format control is one, bits 0-32 of the entry, with 31 zeros appended on the right, form the 64-bit absolute address of the region.

ACCF-Validity Control (AV): When EDAT-2 applies and the RTTE-format control is one, bit 47 is the access-control-bits and fetch-protection bit (ACCF) validity control. When the AV control is zero, bits 48-52 of the region-third-table entry are ignored. When the AV control is one, bits 48-52 are used as described below.

Access-Control Bits (ACC): When EDAT-2 applies, and both the RTTE-format control and the AV control in the region-third-table-entry are one, bits 48-51 of the entry contain the access-control bits that may be used for any key-controlled access checking that applies to the address. It is unpredictable whether the CPU uses these bits or the access-control bits in the storage key of the 4 K-byte block corresponding to the address.

Fetch-Protection Bit (F): When EDAT-2 applies, and both the RTTE-format control and the AV control in the region-third-table-entry are one, bit 52 of the entry contains the fetch-protection bit that may be used for any key-controlled access checking that applies to the address. It is unpredictable whether the CPU uses this bit or the fetch-protection bit in the storage key of the 4 K-byte block corresponding to the address.

Format Control (FC): When EDAT-2 applies, bit 53 of the region-third table entry is the RTTE-format control for the entry, as follows:

- When the FC bit is zero, bits 0-51 of the entry form the segment-table origin, and bits 52 and 55 are reserved.
- When the FC bit is one, bits 0-32 of the entry form the region-frame absolute address, bit 47 is the ACCF-validity control, bits 48-51 are the access-control bits, bit 52 is the fetch-protection bit, and bits 56-57, and 62-63 are reserved. When the instruction-execution-protection facility is enabled (that is, bit 43 of control register 0 is one), bit 55 of the format-1 RTTE is the instruction-execution-protection control; otherwise bit 55 of the format-1 RTTE is reserved.

When EDAT-2 does not apply, bit 53 of the region-third table entry is reserved. Bit 53 is reserved in region-first- and region-second-table entries.

DAT-Protection Bit (P): When EDAT-1 applies, bit 54 is treated as being ORed with the DAT-protection bit in each subsequent region-table entry, and, when applicable, segment-table entry, and page-table entry used in the translation. Thus, when the bit is one, DAT protection applies to the entire region or regions specified by the region-table entry.

When the enhanced-DAT facility 1 is not installed, or when the facility is installed but the enhanced-DAT-enablement control is zero, bit 54 of the region-table entry is ignored.

Instruction-Execution-Protection (IEP) Control: When the instruction-execution-protection facility is enabled, and the RTTE-format control is one, bit 55 of the RTTE is the instruction-execution-protection control. When the IEP control is zero, instruction-execution protection does not apply to instructions executed in the region frame. When the IEP control is one, instruction-execution protection applies to the region frame; in the absence of higher-priority exception conditions, any attempt to execute instructions from the region frame causes a protection exception to be recognized.

Region-Second-Table Offset, Region-Third-Table Offset, and Segment-Table Offset (TF): A region-first-table entry contains a region-second-table offset. A region-second-table entry contains a region-third-table offset. When EDAT-2 does not apply, or when EDAT-2 applies but the RTTE-format control in the region-third table entry is zero, a region-third-table entry contains a segment-table offset. The following description applies to each of the three offsets. Bits 56 and 57 of the entry specify the length of a portion of the next-lower-level table that is missing at the beginning of the table, that is, the bits specify the location of the first entry actually existing in the next-lower-level table. The bits specify the length of the missing portion in units of 4,096 bytes, thus making the length of the missing portion variable in multiples of 512 entries. The length of the missing portion, in units of 4,096 bytes, is equal to the TF value. The contents of the offset field, in conjunction with the length field, bits 62 and 63, are used to establish whether the portion of the virtual address (RSX, RTX, or SX) to be translated by means of the next-lower-level table designates an entry that actually exists in the table.

When EDAT-2 applies and the RTTE-format-control of a region-third-table entry is one, bits 56-57 of the entry are reserved.

Region-Invalid Bit (I): Bit 58 in a region-first-table entry or region-second-table entry controls whether the set of regions associated with the entry is available. Bit 58 in a region-third-table entry controls whether the single region associated with the entry is available. When bit 58 is zero, address translation proceeds by using the region-table entry. When the bit is one, the entry cannot be used for translation.

When the region-invalid bit is one, all other bits in the region-table entry are available for use by programming.

Common-Region Bit (CR): When EDAT-2 applies, bit 59 controls the use of the translation-lookaside-buffer (TLB) copies of the region-third-table entry. When EDAT-2 applies but the RTTE-format control is zero, bit 59 also controls the use of the TLB copies of any segment table designated by the region-third-table entry, and any page table designated by the segment-table entry.

A zero identifies a private region; in this case, the region-third-table entry, and any lower-level table entries it designates may be used only in association with the region-third-table origin that designates the region-third table in which the region-third-table entry resides, except that on models that implement a TLB composite table entry that includes the region-third-table entry, the TLB composite table entry and any lower-level tables it designates may be used only when (a) the table origin in the TLB composite table entry matches the ASCE table origin or table origin of a higher-level region-table entry, and (b) the region

indices in the TLB composite table entry match the corresponding indices of the virtual address.

A one identifies a common region; in this case, the region-third-table entry and any lower-level tables it designates may continue to be used for translating addresses corresponding to the region-third index, even though a different region-third table is specified. However, TLB copies of the region-third-table entry and lower-level tables for a common region are not usable if the private-space control, bit 55, is one in the address-space-control element used in the translation or if that address-space-control element is a real-space designation. The common-region bit must be zero if the region-third-table entry is fetched from storage during a translation when the private-space control is one in the address-space-control element being used; otherwise, a translation-specification exception is recognized.

Bit 59 is ignored in the region-first- and region-second-table entries, and, when EDAT-2 does not apply, in the region-third-table entry.

Table-Type Bits (TT): Bits 60 and 61 of the region-first-table entry, region-second-table entry, and region-third-table entry identify the level of the table containing the entry, as follows:

Bits 60 and 61	Region-Table Level
11	First
10	Second
01	Third

Bits 60 and 61 must identify the correct table level, considering the type of table designation that is the address-space-control element being used in the translation and the number of table levels that have so far been used; otherwise, a translation-specification exception is recognized.

Region-Second-Table Length, Region-Third-Table Length, and Segment-Table Length (TL): A region-first-table entry contains a region-second-table length. A region-second-table entry contains a region-third-table length. When EDAT-2 does not apply, or when EDAT-2 applies but the RTTE-format control in the region-third table entry is zero, a region-third-table entry contains a segment-table length. The following description applies to each of the three lengths. Bits 62 and 63 of the entry specify the length of the next-lower-level table in units of 4,096 bytes, thus making the length of the table variable in multiples of 512 entries. The length of the next-lower-level table, in units of 4,096 bytes, is one more than the TL value. The contents of the length field, in conjunction with the offset field, bits 56 and 57, are used to establish whether the portion of the virtual address (RSX, RTX, or SX) to be translated by means of the next-lower-level table designates an entry that actually exists in the table.

When EDAT-2 applies and the RTTE-format-control of a region-third-table entry is one, bits 62-63 of the entry are available for programming.

All other bit positions of the region-table entry are reserved for possible future extensions and should contain zeros; otherwise, the program may not operate compatibly in the future.

Programming Note: When the common-region (CR) bit is one in a valid region-third-table entry fetched from storage during dynamic-address translation for a non-private address space, that is, for an address space in which the private-space control is zero in the address-space-control element being used, then the following conditions apply:

1. The region-third-table entries corresponding to the same virtual address in all other non-private address spaces must be identical.

   If the program alters such a region-third-table entry in one non-private address space, then it must also (a) identically alter all other region-third-table entries corresponding to the same virtual address in all other non-private address spaces, and (b) ensure that the affected entries are cleared from the TLBs of all CPUs in the configuration. Further information on clearing TLB entries may be found in “Modification of Translation Tables” on page 3-67.

2. The program must ensure that region-third-table entries in which the CR bit is set to one, and any region-first- and region-second-table entries used by the DAT process to locate such region-third-table entries, are consistent across all address spaces. That is, if the DAT process can successfully locate a region-third-table entry in which the CR bit is one in one non-private address space, then there must be no exception condition that prevents DAT from locating the region-third-table entry corresponding to the same virtual address in any other non-private address spaces.

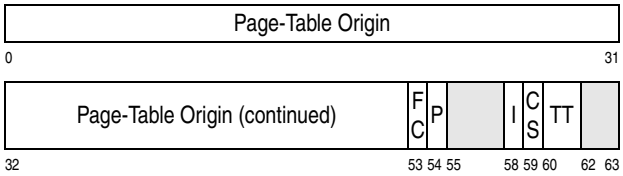
If the program alters such a region-table entry in one non-private address space, then it must also (a) perform consistent alteration to all corresponding table entries in all other non-private address spaces, and (b) ensure that the affected entries are cleared from the TLBs of all CPUs in the configuration.

3. If the program fails to maintain consistent DAT table entries as described above, results are unpredictable and may include the presentation of a delayed-access-exception machine check, as described in “Delayed Access Exception” on page 11-17.

Segment-Table Entries

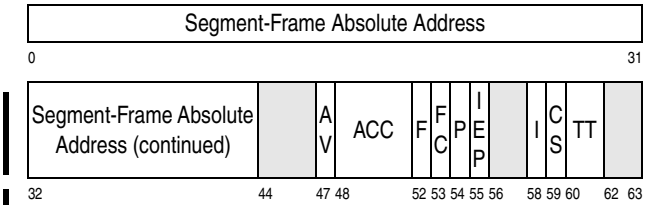
When EDAT-1 does not apply, or when enhanced DAT applies and the STE-format control, bit 53 of the segment-table entry is zero, the entry fetched from the segment table has the following format:

Segment-Table Entry (TT=00, FC=0)



When EDAT-1 applies and the STE-format control is one, the entry fetched from the segment table has the following format:

Segment-Table Entry (TT=00, FC=1)



The fields in the segment-table entry are allocated as follows:

Page-Table Origin: When EDAT-1 does not apply, or when EDAT-1 applies but the STE-format control, bit 53 of the segment-table entry, is zero, bits 0-52, with 11 zeros appended on the right, form a 64-bit address that designates the beginning of a page table. It is unpredictable whether the address is real or absolute.

Segment-Frame Absolute Address (SFAA): When EDAT-1 applies and the STE-format control is one, bits 0-43 of the entry, with 20 zeros appended on the right, form the 64-bit absolute address of the segment.

ACCF-Validity Control (AV): When EDAT-1 applies and the STE-format control is one, bit 47 is the access-control-bits and fetch-protection bit (ACCF) validity control. When the AV control is zero, bits 48-52 of the segment-table entry are ignored. When the AV control is one, bits 48-52 are used as described below.

Access-Control Bits (ACC): When EDAT-1 applies, and both the STE-format control and the AV control in the segment-table entry are one, bits 48-51 of the segment-table entry contain the access-control bits that may be used for any key-controlled access checking that applies to the address. It is unpredictable whether the CPU uses these bits or the access-control bits in the storage key of the 4 K-byte block corresponding to the address.

Fetch-Protection Bit (F): When EDAT-1 applies, and both the STE-format control and the AV control in the segment-table entry are one, bit 52 of the segment-table entry contains the fetch-protection bit that may be used for any key-controlled access checking that applies to the address. It is unpredictable whether the CPU uses this bit or the fetch-protection bit in the storage key of the 4 K-byte block corresponding to the address.

STE-Format Control (FC): When EDAT-1 applies, bit 53 is the format control for the segment-table entry, as follows:

- When the FC bit is zero, bits 0-52 of the entry form the page-table origin, and bit 55 is reserved.
- When the FC bit is one, bits 0-43 of the entry form the segment-frame absolute address, bit 47 is the ACCF-validity control, bits 48-51 are the access-control bits, and bit 52 is the fetch-protection bit. When the instruction-execution-protection facility is enabled, bit 55 of the STE is the instruction-execution-protection control; otherwise bit 55 of the STE is reserved.

When EDAT-1 does not apply, bit 53 is ignored.

DAT-Protection Bit (P): Bit 54, when one, indicates that DAT protection applies to the entire segment.

When EDAT-1 does not apply, bit 54 is treated as being ORed with the DAT-protection bit in the page-table entry used in the translation.

When EDAT-1 applies, the DAT-protection bit in any and all region-table entries used in the translation are treated as being ORed with the DAT-protection bit in the segment-table entry; when the STE-format control is zero, the DAT-protection bit in the STE is further treated as being ORed with the DAT-protection bit in the page-table entry.

Instruction-Execution-Protection (IEP) Control: When the instruction-execution-protection facility is enabled, and the STE-format control is one, bit 55 of the STE is the instruction-execution-protection control. When the IEP control is zero, instruction-execution protection does not apply to instructions executed in the segment frame. When the IEP control is one, instruction-execution protection applies to the segment frame; in the absence of higher-priority exception conditions, any attempt to execute instructions from the segment frame causes a protection exception to be recognized.

Segment-Invalid Bit (I): Bit 58 controls whether the segment associated with the segment-table entry is available. When the bit is zero, address translation proceeds by using the segment-table entry. When the bit is one, the segment-table entry cannot be used for translation.

When the segment-invalid bit is one, all other bits in the segment-table entry are available for use by programming.

Common-Segment Bit (CS): Bit 59 controls the use of the translation-lookaside-buffer (TLB) copies of the segment-table entry. When EDAT-1 does not apply or when EDAT-1 applies but the format control is zero, bit 59 also controls the use of the TLB copies of the page table designated by the segment-table entry.

A zero identifies a private segment; in this case, the segment-table entry and any page table it designates may be used only in association with the segment-table origin that designates the segment table in which the segment-table entry resides, except that on models that implement a TLB composite table entry that includes the STE , the TLB composite table entry and any page table it designates may be used only when (a) the table origin in the TLB composite table entry matches the ASCE table origin or table origin of a higher-level region-table entry, and (b) the region and segment indices in the TLB composite table entry match the corresponding indices of the virtual address.

A one identifies a common segment; in this case, the segment-table entry and any page table it designates may continue to be used for translating addresses corresponding to the segment index, even though a different segment table is specified. However, TLB copies of the segment-table entry and any page table for a common segment are not usable if the private-space control, bit 55, is one in the address-space-control element used in the translation or if that address-space-control element is a real-space designation. The common-segment bit must be zero if the segment-table entry is fetched from storage during a translation when the private-space control is one in the address-space-control element being used; otherwise, a translation-specification exception is recognized.

When EDAT-2 applies, the effective value of the common-segment bit is the logical OR of bits 59 of the segment-table entry and any region-third-table entry designating the segment-table entry.

Table-Type Bits (TT): Bits 60 and 61 of the segment-table entry are 00 binary to identify the level of the table containing the entry. The meanings of all possible values of bits 60 and 61 in a region-table entry or segment-table entry are as follows:

Bits 60 and 61	Table Level
11	Region-first
10	Region-second
01	Region-third
00	Segment

Bits 60 and 61 must identify the correct table level, considering the type of table designation that is the address-space-control element being used in the translation and the number of table levels that have so far been used; otherwise, a translation-specification exception is recognized.

Bits 62-63 are available for programming.

All other bit positions of the segment-table entry are reserved for possible future extensions and should contain zeros; otherwise, the program may not operate compatibly in the future.

Programming Note: When the common-segment (CS) bit is one in a valid segment-table entry fetched from storage during dynamic-address translation for a non-private address space, that is, for an address space in which the private-space control is zero in the address-space-control element being used, then the following conditions apply:

1. The segment-table entries corresponding to the same virtual address in all other non-private address spaces must be identical.

   If the program alters such a segment-table entry in one non-private address space, then it must also (a) identically alter all other segment-table entries corresponding to the same virtual address in all other non-private address spaces, and (b) ensure that the affected entries are cleared from the TLBs of all CPUs in the configuration. Further information on clearing TLB entries may be found in “Modification of Translation Tables” on page 3-67.

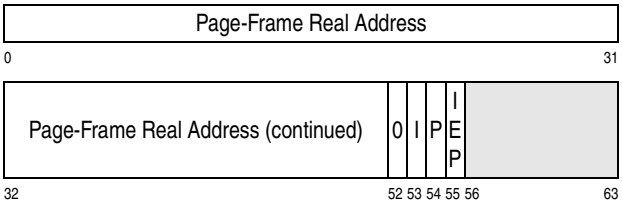
2. The program must ensure that segment-table entries in which the CS bit is set to one, and any region-table entries used by the DAT process to locate such segment-table entries, are consistent across all address spaces. That is, if the DAT process can successfully locate a segment-table entry in which the CS bit is one in one non-private address space, then there must be no exception condition that prevents DAT from locating the segment-table entry corresponding to the same virtual address in any other non-private address spaces.

   If the program alters such a segment-table entry or region-table entry in one non-private address space, then it must also (a) perform consistent alteration to all corresponding table entries in all other non-private address spaces, and (b) ensure that the affected entries are cleared from the TLBs of all CPUs in the configuration.

3. If the program fails to maintain consistent DAT table entries as described above, results are unpredictable and may include the presentation of a delayed-access-exception machine check, as described in “Delayed Access Exception” on page 11-17.

Page-Table Entries

The entry fetched from the page table entry has the following format:



The fields in the page-table entry are allocated as follows:

Page-Frame Real Address (PFRA): Bits 0-51 provide the leftmost bits of a real storage address. When these bits are concatenated with the 12-bit byte-index field of the virtual address on the right, a 64-bit real address is obtained.

Page-Invalid Bit (I): Bit 53 controls whether the page associated with the page-table entry is available. When the bit is zero, address translation proceeds by using the page-table entry. When the bit is one, the page-table entry cannot be used for translation.

When the page-invalid bit is one, all other bits in the page-table entry are available for use by programming.

DAT-Protection Bit (P): Bit 54 controls whether store accesses can be made in the page. This protection mechanism is in addition to the key-controlled-protection and low-address-protection mechanisms. The bit has no effect on fetch accesses. If the bit is zero, stores are permitted to the page, subject to the following additional constraints:

- The DAT-protection bit being zero in the segment-table entry used in the translation,
- When EDAT-1 applies, the DAT-protection bit being zero in all region-table entries used in the translation,
- Other protection mechanisms

If the bit is one, stores are disallowed. When no higher priority exception conditions exist, an attempt to store when the DAT-protection bit is one causes a protection exception to be recognized. The DAT-protection bit in the segment-table entry is treated as being ORed with bit 54 when determining whether

DAT protection applies to the page. When EDAT-1 applies, the DAT-protection bits in any region-table entries used in translation are also treated as being ORed with bit 54 when determining whether DAT-protection applies.

Bit position 52 of the entry must contain zero; otherwise, a translation-specification exception is recognized as part of the execution of an instruction using that entry for address translation.

Instruction-Execution-Protection (IEP) Control: When the instruction-execution-protection facility is enabled, bit 55 of the PTE is the instruction-execution-protection control. When the IEP control is zero, instruction-execution protection does not apply to instruction executed in the page frame. When the IEP control is one, instruction-execution protection applies to the page frame; in the absence of higher-priority exception conditions, any attempt to execute instructions from the page frame causes a protection exception to be recognized.

When the instruction-execution-protection (IEP) facility is not installed in the configuration and EDAT-1 does not apply, it is unpredictable whether a translation-specification exception is recognized when bit 55 of the page-table entry is one. When the IEP facility is not installed in the configuration and EDAT-1 applies, or when the IEP facility is installed but disabled, bit 55 of the page-table entry is ignored.

Bit positions 56-63 are not assigned and are ignored.

Programming Note: Bit 55 of the format-1 region-third-table entry, format-1 segment-table entry, and page table entry was formerly defined to be the change-recording override (CO). The change-recording override was never implemented on any IBM processor, and this feature has been removed from the architecture. Bit 55 of the respective table entries is now defined to be the instruction-execution-protection control.

Architecture Notes:

Translation Process

This section describes the translation process as it is performed implicitly before a virtual address is used to access main storage. Explicit translation, which is the process of translating the operand address of LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, STORE REAL ADDRESS, and TEST PROTECTION, is the same, except that, for LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS and TEST PROTECTION, region-first-translation, region-second-translation, region-third-translation, and segment-translation exceptions are not recognized, and for LOAD REAL ADDRESS and TEST PROTECTION, page-translation exceptions are not recognized; such conditions are instead indicated by the condition code. Translation of the operand address of LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS and STORE REAL ADDRESS also differs in that the CPU may be in the real mode.

Translation of a virtual address is controlled by the DAT-mode bit and address-space-control bits in the PSW, by bits in control register 0, and by the address-space-control elements (ASCEs) in control registers 1, 7, and 13 and as specified by the access registers. When the ASCE used in a translation is a region-first-table designation, the translation is performed by means of a region first table, region second table, region third table, and, when applicable, a segment table and page table, all of which reside in real or absolute storage. When the ASCE is a lower-level type of table designation (region-second-table designation, region-third-table designation, or segment-table designation) the translation is performed by means of only the table levels beginning with the designated level, and the virtual-address bits that would, if nonzero, require use of a higher level or levels of table must be all zeros; otherwise, an ASCE-type exception is recognized. When the ASCE is a real-space designation, the virtual address is treated as a real address, and table entries in real or absolute storage are not used.

The address-space-control element (ASCE) used for a particular address translation is called the effective ASCE. Accordingly, when a primary virtual address is translated, the contents of control register 1 are used as the effective ASCE. Similarly, for a secondary virtual address, the contents of control register 7 are used; for an AR-specified virtual address, the ASCE specified by the access register is used; and for a home virtual address, the contents of control register 13 are used.

When the real-space control in the effective ASCE is zero, the designation-type control in the ASCE specifies the table-designation type of the ASCE: region-first-table designation, region-second-table designation, region-third-table designation, or segment-table

designation. The corresponding portion of the virtual address (region first index, region second index, region third index, or segment index) is checked against the table-length field in the designation, and it is added to the origin in the designation to select an entry in the designated table. If the selected entry is outside its table, as determined by the table-length field in the designation, or if the I bit is one in the selected entry, a region-first-translation, region-second-translation, region-third-translation, or segment-translation exception is recognized, depending on the table level specified by the designation. If the table-type bits in the selected entry do not indicate the expected table level, a translation-specification exception is recognized.

The table entry selected by means of the effective ASCE designates the next-lower-level table to be used. If the current table is a region first table, region second table, or region third table, the next portion of the virtual address (region second index, region third index, or, when applicable, segment index, respectively) is checked against the table-offset and table-length fields in the current table entry, and it is added to the origin in the entry to select an entry in the next-lower-level table. If the selected entry in the next table is outside its table, as determined by the table-offset and table-length fields in the current table entry, or if the I bit is one in the selected entry, a region-second-translation, region-third-translation, or, when applicable, segment-translation exception is recognized, depending on the level of the next table. If the table-type bits in the selected entry do not indicate the expected table level, a translation-specification exception is recognized.

Processing of portions of the virtual address continues by means of any successive table levels until a leaf table entry is reached. When EDAT-1 applies, the DAT-protection bit in any and all region-table entries used during the translation are treated as being ORed with the respective bit in any segment-table entry used in the translation.

When EDAT-2 applies, the region-third-table entry contains a common-region bit that controls the use of the TLB copies of any lower-level tables designated by the region-third-table entry. The common-region bit is logically ORed with the common-segment bit of any segment-table entry used in the translation.

When EDAT-2 applies and a region-third-table entry in which the RTTE-format control is one is selected, the region-third-table entry is the leaf table entry, and the following conditions are in effect:

- The region-third-table entry contains the leftmost 33 bits of the absolute address that represents the translation of the virtual address.
- The segment-index, page-index and byte-index fields of the virtual address are used unchanged as the rightmost 31 bit positions of the real address.
- The region-third-table entry also contains the ACCF-validity control, the access-control bits, the fetch-protection bit, the DAT-protection bit, and the common-region bit that apply to the region. When the instruction-execution-protection facility is enabled, the region-third-table entry also contains the instruction-execution-protection control that applies to the region.

When EDAT-1 applies and a segment-table entry in which the STE-format control is one is selected, the segment-table entry is the leaf table entry, the following conditions are in effect:

- The segment-table entry contains the leftmost bits of the absolute address that represents the translation of the virtual address.
- The page-index and byte-index fields of the virtual address are used unchanged as the rightmost bit positions of the real address.
- The segment-table entry also contains the ACCF-validity control, the access-control bits, the fetch-protection bit, the DAT-protection bit, and common-segment bit that apply to the segment. When the instruction-execution-protection facility is enabled, the segment-table entry also contains the instruction-execution-protection control that applies to the segment.

When EDAT-1 does not apply, or when EDAT-1 applies but the STE-format control is zero, the following conditions are in effect:

- The segment-table entry contains a DAT-protection bit that applies to all pages in the specified segment; the segment-table entry also contains a common-segment bit that controls the use of the TLB copies of the page table designated by the segment-table entry.

- The segment-table entry designates the page table to be used, and the page-table entry is the leaf table entry.
- The page-index portion of the virtual address is added to the page-table origin in the segment-table entry to select an entry in the page table. If the I bit is one in the page-table entry, a page-translation exception is recognized. The page-table entry contains the leftmost bits of the real address that represents the translation of the virtual address, and it contains a DAT-protection bit that applies only to the page specified by the page-table entry. When the instruction-execution-protection facility is enabled, the page-table entry also contains the instruction-execution-protection control that applies to the page.
- The byte-index field of the virtual address is used unchanged as the rightmost bit positions of the real address.

In order to eliminate the delay associated with references to translation tables in real or absolute storage, the information fetched from the tables normally is also placed in a special buffer, the translation-lookaside buffer (TLB), and subsequent translations involving the same table entries may be performed by using the information recorded in the TLB. The TLB may also record virtual-equals-real translations related to a real-space designation. The operation of the TLB is described in "Translation-Lookaside Buffer" on page 3-62.

Whenever access to real or absolute storage is made during the address-translation process for the purpose of fetching an entry from a region table, segment table, or page table, key-controlled protection does not apply.

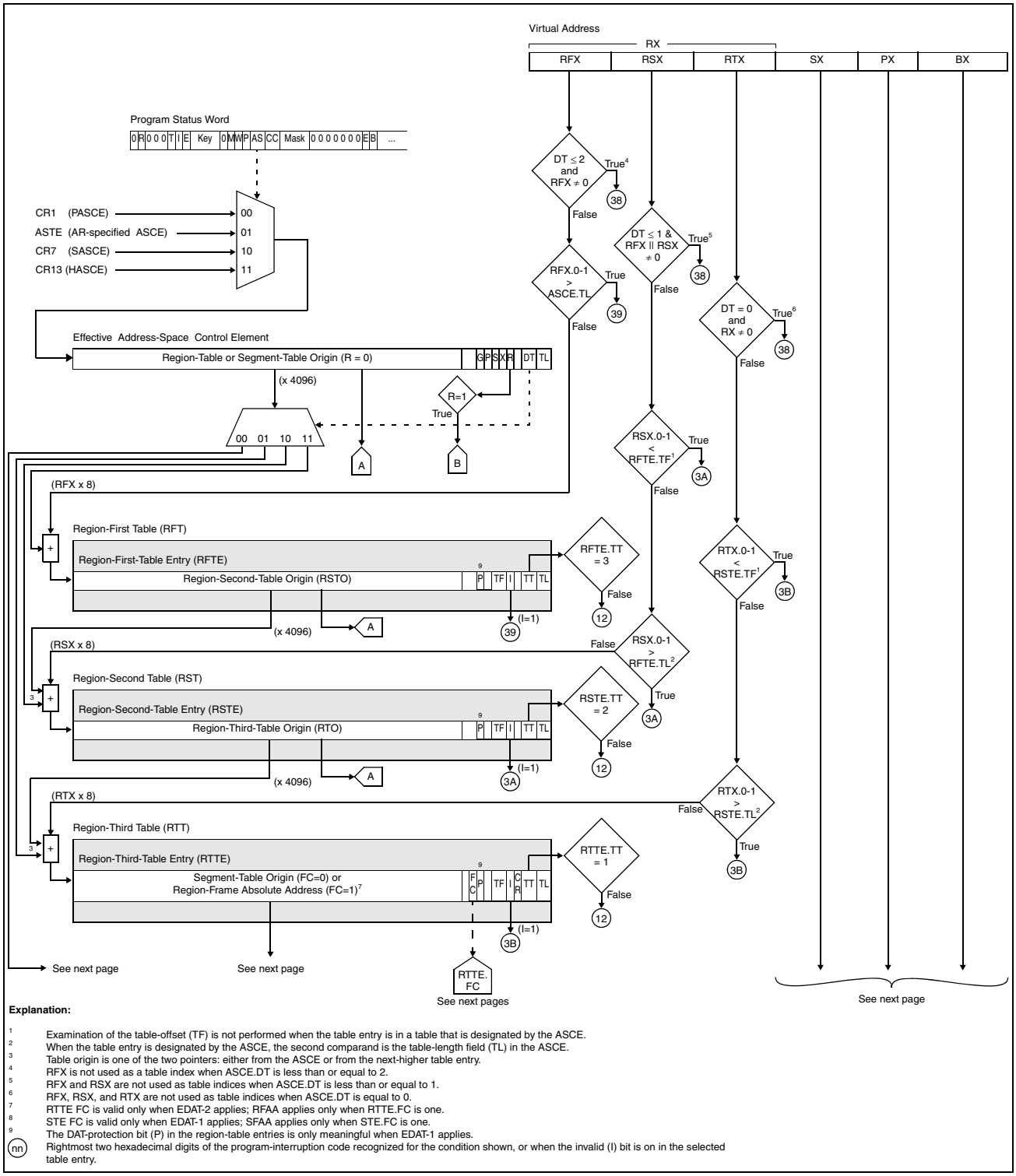
The translation process, including the effect of the TLB, is shown graphically in Figure 3-16 on page 3-55.

Inspection of Real-Space Control

When the effective address-space-control element (ASCE) contains a real-space control, bit 58, having the value zero, the ASCE is a region-table or segment-table designation. When the real-space control is one, the ASCE is a real-space designation.

Inspection of Designation-Type Control

When the real-space control is zero, the designation-type control, bits 60 and 61 of the effective address-space-control element (ASCE), specifies the table-designation type of the ASCE. Depending on the type, some number of leftmost bits of the virtual address being translated must be zeros; otherwise, an ASCE-type exception is recognized. For each possible value of bits 60 and 61, the table-designa-



Explanation:

1. Examination of the table-offset (TF) is not performed when the table entry is in a table that is designated by the ASCE.
2. When the table entry is designated by the ASCE, the second comparand is the table-length field (TL) in the ASCE.
3. Table origin is one of the two pointers: either from the ASCE or from the next-higher table entry.
4. RFX is not used as a table index when ASCE.DT is less than or equal to 2.
5. RFX and RSX are not used as table indices when ASCE.DT is less than or equal to 1.
6. RFX, RSX, and RTX are not used as table indices when ASCE.DT is equal to 0.
7. RTTE FC is valid only when EDAT-2 applies; RFAA applies only when RTTE.FC is one.
8. STE FC is valid only when EDAT-1 applies; SFAA applies only when STE.FC is one.
9. The DAT-protection bit (P) in the region-table entries is only meaningful when EDAT-1 applies.

(nn) Rightmost two hexadecimal digits of the program-interruption code recognized for the condition shown, or when the invalid (I) bit is on in the selected table entry.

Figure 3-16. Translation Process (Part 1 of 3)

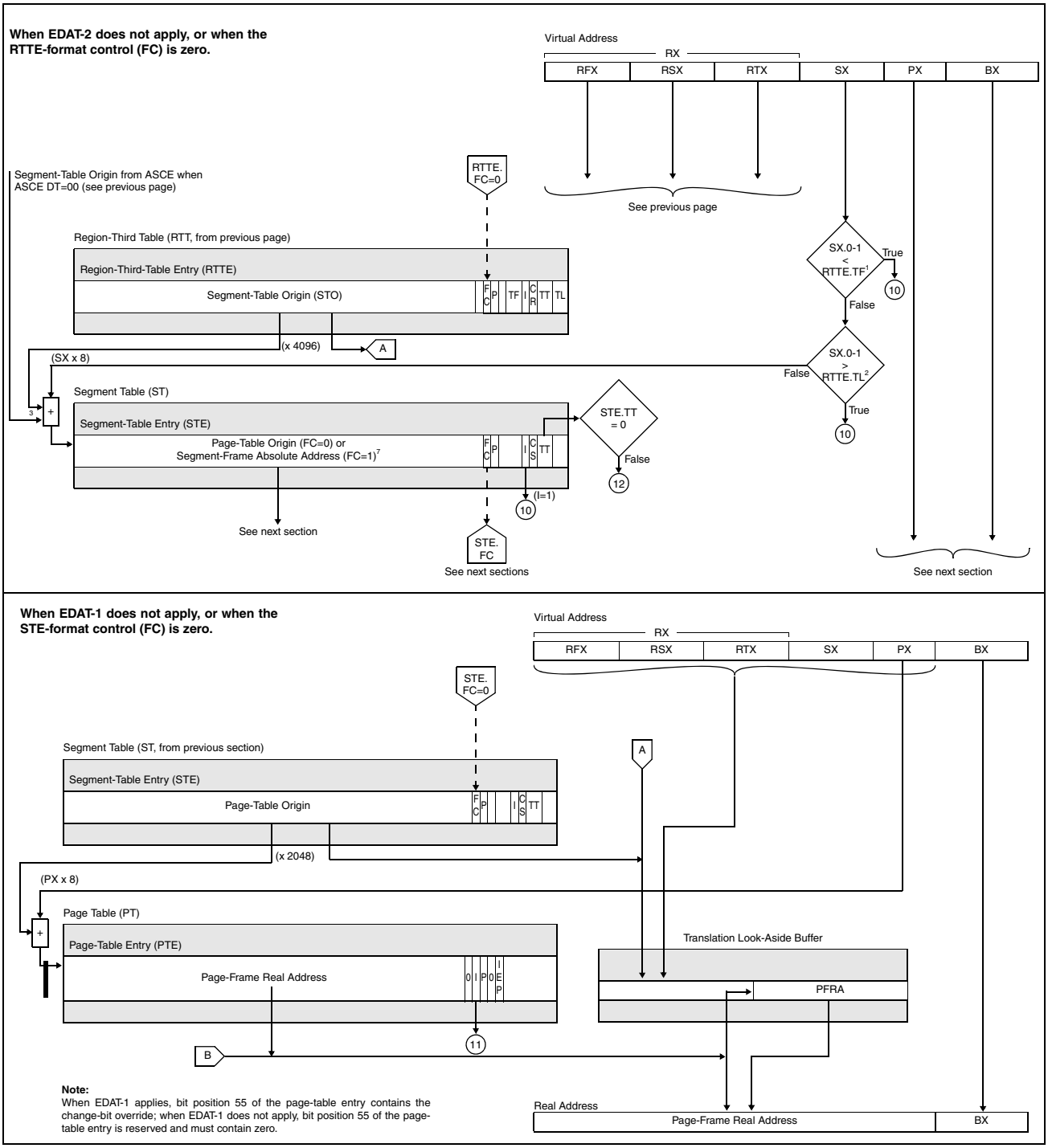


Figure 3-16. Translation Process (Part 2 of 3)

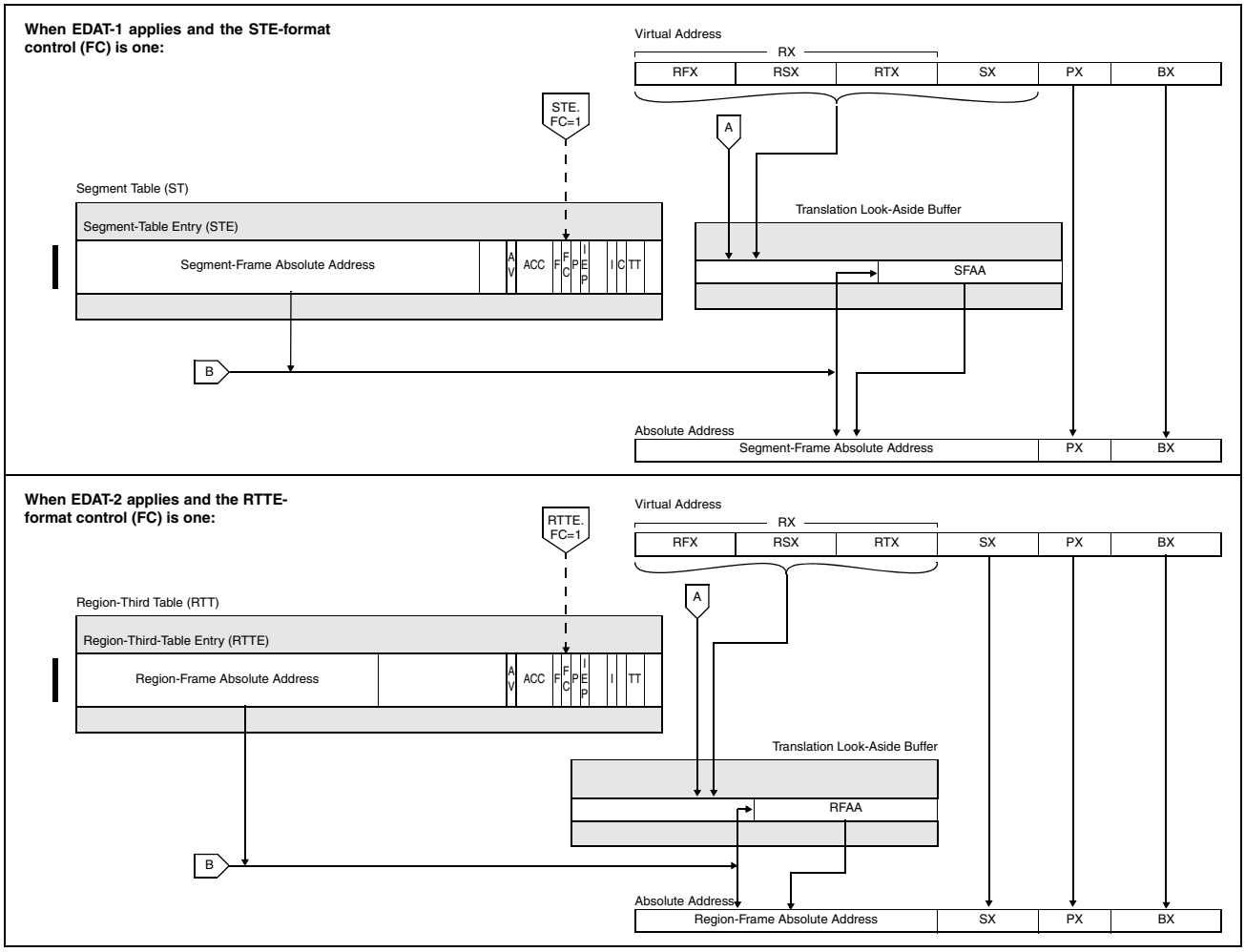


Figure 3-16. Translation Process (Part 3 of 3)

tion type and the virtual-address bits required to be zeros are as follows:

Bits 60 and 61	Designation Type	Virtual-Address Bits Required to be Zeros
11	Region-first-table	None
10	Region-second-table	0-10
01	Region-third-table	0-21
00	Segment-table	0-32

Lookup in a Table Designated by an Address-Space-Control Element

The designation-type control, bits 60 and 61 of the effective address-space-control element (ASCE), specifies both the table-designation type of the ASCE and the portion of the virtual address that is to be translated by means of the designated table, as follows:

Bits 60 and 61	Designation Type	Virtual-Address Portion Translated by the Table
11	Region-first-table	Region first index (bits 0-10)
10	Region-second-table	Region second index (bits 11-21)
01	Region-third-table	Region third index (bits 22-32)
00	Segment-table	Segment index (bits 33-43)

Region-First-Table Lookup: When bits 60 and 61 have the value 11 binary, the region-first-index portion of the virtual address, in conjunction with the

region-first-table origin contained in the ASCE, is used to select an entry from the region first table.

The 64-bit address of the region-first-table entry in real or absolute storage is obtained by appending 12 zeros to the right of bits 0-51 of the region-first-table designation and adding the region first index with three rightmost and 50 leftmost zeros appended. When a carry out of bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from 2^{64} - 1 to zero. All 64 bits of the address are used, regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode. When forming the address of a region-first-, region-second-, region-third-, or segment-table entry, it is unpredictable whether prefixing, if any, is applied to the respective table origin contained in the ASCE before the addition of the table index value, or prefixing is applied to the table-entry address that is formed by the addition of the table origin and table index value.

As part of the region-first-table-lookup process, bits 0 and 1 of the virtual address (which are bits 0 and 1 of the region first index) are compared against the table length, bits 62 and 63 of the region-first-table designation, to establish whether the addressed entry is within the region first table. If the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a region-first-translation exception is recognized. The comparison against the table length may be omitted if the equivalent of a region-first-table entry in the translation-lookaside buffer is used in the translation.

All eight bytes of the region-first-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address generated for fetching the region-first-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the unit of operation is suppressed.

Bit 58 of the entry fetched from the region first table specifies whether the corresponding set of regions is available. This bit is inspected, and, if it is one, a region-first-translation exception is recognized.

A translation-specification exception is recognized if the table-type bits, bits 60 and 61, in the region-first-table entry do not have the same value as bits 60 and 61 of the ASCE.

When no exceptions are recognized in the process of region-first-table lookup, the entry fetched from the region first table designates the beginning and specifies the offset and length of the corresponding region second table.

Region-Second-Table Lookup: When bits 60 and 61 of the ASCE have the value 10 binary, the region-second-index portion of the virtual address, in conjunction with the region-second-table origin contained in the ASCE, is used to select an entry from the region second table. Bits 11 and 12 of the virtual address (which are bits 0 and 1 of the region second index) are compared against the table length in the ASCE. If the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a region-second-translation exception is recognized. The comparison against the table length may be omitted if the equivalent of a region-second-table entry in the translation-lookaside buffer is used in the translation. The region-second-table-lookup process is otherwise the same as the region-first-table-lookup process, except that a region-second-translation exception is recognized if bit 58 is one in the region-second-table entry. When no exceptions are recognized, the entry fetched from the region second table designates the beginning and specifies the offset and length of the corresponding region third table.

Region-Third-Table Lookup: When bits 60 and 61 of the ASCE have the value 01 binary, the region-third-index portion of the virtual address, in conjunction with the region-third-table origin contained in the ASCE, is used to select an entry from the region third table. Bits 22 and 23 of the virtual address (which are bits 0 and 1 of the region third index) are compared against the table length in the ASCE. If the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a region-third-translation exception is recognized. The comparison against the table length may be omitted if the equivalent of a region-third-table entry in the translation-lookaside buffer is used in the translation. When EDAT-2 applies, a translation-specification exception is recognized if (1) the private-space control, bit 55, in the ASCE is one and (2) the common-region bit, bit 59, in the entry fetched from the region-third table is one. The region-third-table-lookup process is otherwise the same as the region-first-table-lookup process, including the checking of the table-type bits in the region-third-table entry, except that a region-third-translation exception is recognized if bit 58 is one in the region-third-table entry.

When no exceptions are recognized, processing is as follows:

- When EDAT-2 does not apply, or when EDAT-2 applies but the RTTE-format control is zero, the entry fetched from the region third table designates the beginning and specifies the offset and length of the corresponding segment table.
- When EDAT-2 applies and the RTTE-format control is one, the entry fetched from the region-third table is a leaf-table entry and contains the leftmost bits of the region-frame absolute address. In this case, if (a) the instruction-execution-protection facility is enabled, (b) the storage reference for which the translation is being performed is for instruction execution, and (c) the instruction-execution-protection control, bit 55, is one, then a protection exception is recognized.
- Regardless of the RTTE-format control, if (a) EDAT-2 applies, (b) the DAT-protection bit, bit 54, is one either in any region-table entry used in the translation, and (c) the storage reference for which the translation is being performed is a store, then a protection exception is recognized.

Segment-Table Lookup: When bits 60 and 61 of the ASCE have the value 00 binary, the segment-index portion of the virtual address, in conjunction with the segment-table origin contained in the ASCE, is used to select an entry from the segment table. Bits 33 and 34 of the virtual address (which are bits 0 and 1 of the segment index) are compared against the table length in the ASCE. If the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a segment-translation exception is recognized. The comparison against the table length may be omitted if the equivalent of a segment-table entry in the translation-lookaside buffer is used in the translation. A translation-specification exception is recognized if (1) the private-space control, bit 55, in the ASCE is one and (2) the common-segment bit, bit 59, in the entry fetched from the segment table is one. When EDAT-2 applies and the common-region bit is one in a region-third-table entry designating the segment table, then the common-segment bit is assumed to be one in all entries in the segment table. The segment-table-lookup process is otherwise the same as the region-first-table-lookup process, including the checking of the table-type bits in the segment-table entry, except that a segment-translation exception is recognized if bit 58 is one in the segment-table entry. When no exceptions are recognized, processing is as follows:

- When EDAT-1 does not apply, or when EDAT-1 applies but the STE-format control is zero, the entry fetched from the segment table designates the beginning of the corresponding page table, and processing continues as described in “Page-Table Lookup”, below.
- When EDAT-1 applies and the STE-format control is one, the entry fetched from the segment table is a leaf-table entry and contains the leftmost bits of the segment-frame absolute address. In this case, if (a) the instruction-execution-protection facility is enabled, (b) the storage reference for which the translation is being performed is for instruction execution, and (c) the instruction-execution-protection control, bit 55, is one, then a protection exception is recognized.
- Regardless of the STE-format control, if (a) EDAT-1 applies, (b) the DAT-protection bit, bit 54, is one either in any region-table entry used in the translation or in the segment-table entry, and (c) the storage reference for which the translation is being performed is a store, then a protection exception is recognized.

Lookup in a Table Designated by a Region-Table Entry

When the effective address-space-control element (ASCE) is a region-table designation, a region-table entry is selected as described in the preceding section. Then the contents of the selected entry and the next index portion of the virtual address are used to select an entry in the next-lower-level table, which may be another region table or a segment table.

Region-Second-Table Lookup: When the table entry selected by means of the ASCE is a region-first-table entry, the region-second-index portion of the virtual address, in conjunction with the region-second-table origin contained in the region-first-table entry, is used to select an entry from the region second table.

The 64-bit address of the region-second-table entry in real or absolute storage is obtained by appending 12 zeros to the right of bits 0-51 of the region-first-table entry and adding the region second index with three rightmost and 50 leftmost zeros appended. When a carry out of bit position 0 occurs during the addition, an addressing exception may be recog-

nized, or the carry may be ignored, causing the table to wrap from 2^{64} - 1 to zero. All 64 bits of the address are used, regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode. When forming the address of a region-second-, region-third-, or segment-table entry, it is unpredictable whether prefixing, if any, is applied to the respective table origin contained in the higher-level table entry before the addition of the table index value, or prefixing is applied to the table-entry address that is formed by the addition of the table origin and table index value.

As part of the region-second-table-lookup process, bits 11 and 12 of the virtual address (which are bits 0 and 1 of the region second index) are compared against the table offset, bits 56 and 57 of the region-first-table entry, and against the table length, bits 62 and 63 of the region-first-table entry, to establish whether the addressed entry is within the region second table. If the value in the table-offset field is greater than the value in the corresponding bit positions of the virtual address, or if the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a region-second-translation exception is recognized.

All eight bytes of the region-second-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address generated for fetching the region-second-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the unit of operation is suppressed.

Bit 58 of the entry fetched from the region second table specifies whether the corresponding set of regions is available. This bit is inspected, and, if it is one, a region-second-translation exception is recognized.

A translation-specification exception is recognized if the table-type bits, bits 60 and 61, in the region-second-table entry do not have a value that is one less than the value of those bits in the next-higher-level table.

When no exceptions are recognized in the process of region-second-table lookup, the entry fetched from the region second table designates the beginning and specifies the offset and length of the corresponding region third table.

Region-Third-Table Lookup: When the table entry selected by means of the ASCE is a region-second-table entry, or if a region-second-table entry has been selected by means of the contents of a region-first-table entry, the region-third-index portion of the virtual address, in conjunction with the region-third-table origin contained in the region-second-table entry, is used to select an entry from the region third table. Bits 22 and 23 of the virtual address (which are bits 0 and 1 of the region third index) are compared against the table offset and table length in the region-second-table entry. A region-third-translation exception is recognized if the table offset is greater than bits 22 and 23 or if the table length is less than bits 22 and 23. When EDAT-2 applies, a translation-specification exception is recognized if (1) the private-space control, bit 55, in the ASCE is one and (2) the common-region bit, bit 59, in the entry fetched from the region-third table is one. The region-third-table-lookup process is otherwise the same as the region-second-table-lookup process, including the checking of the table-type bits in the region-third-table entry, except that a region-third-translation exception is recognized if bit 58 is one in the region-third-table entry.

When no exceptions are recognized, processing is as follows:

- When EDAT-2 does not apply, or when EDAT-2 applies but the RTTE-format control is zero, the entry fetched from the region third table designates the beginning and specifies the offset and length of the corresponding segment table.
- When EDAT-2 applies and the RTTE-format control is one, the entry fetched from the region-third table is a leaf-table entry and contains the left-most bits of the region-frame absolute address. In this case, if (a) the instruction-execution-protection facility is enabled, (b) the storage reference for which the translation is being performed is for instruction execution, and (c) the instruction-execution-protection control, bit 55, is one, then a protection exception is recognized.
- Regardless of the RTTE-format control, if (a) EDAT-2 applies, (b) the DAT-protection bit, bit 54, is one either in any region-table entry used in the translation, and (c) the storage reference for which the translation is being performed is a store, then a protection exception is recognized.

Segment-Table Lookup: When (1) the table entry selected by means of the ASCE is a region-third-table entry, or a region-third-table entry has been selected by means of the contents of a region-second-table entry, and (2) EDAT-2 does not apply, or EDAT-2 applies but the RTTE-format control is zero, then the segment-index portion of the virtual address, in conjunction with the segment-table origin contained in the region-third-table entry, is used to select an entry from the segment table. Bits 33 and 34 of the virtual address (which are bits 0 and 1 of the segment index) are compared against the table offset and table length in the region-third-table entry. A segment-translation exception is recognized if the table offset is greater than bits 33 and 34 or if the table length is less than bits 33 and 34. A translation-specification exception is recognized if (1) the private-space control, bit 55, in the ASCE is one and (2) the common-segment bit, bit 59, in the entry fetched from the segment table is one. When EDAT-2 applies and the common-region bit is one in a region-third-table entry designating the segment table, then the common-segment bit is assumed to be one in all entries in the segment table. The segment-table-lookup process is otherwise the same as the region-second-table-lookup process, including the checking of the table-type bits in the segment-table entry, except that a segment-translation exception is recognized if bit 58 is one in the segment-table entry. When no exceptions are recognized, processing is as follows:

- When EDAT-1 does not apply, or when EDAT-1 applies but the STE-format control is zero, the entry fetched from the segment table designates the beginning of the corresponding page table, and processing continues as described in "Page-Table Lookup", below.
- When EDAT-1 applies and the STE-format control is one, the entry fetched from the segment table is a leaf-table entry and contains the leftmost bits of the segment-frame absolute address. In this case, if (a) the instruction-execution-protection facility is enabled, (b) the storage reference for which the translation is being performed is for instruction execution, and (c) the instruction-execution-protection control, bit 55, is one, then a protection exception is recognized.
- Regardless of the STE-format control, if (a) EDAT-1 applies, (b) the DAT-protection bit, bit 54, is one either in any region-table entry used in the translation or in the segment-table entry, and (c) the storage reference for which the translation is being performed is a store, then a protection exception is recognized.

Page-Table Lookup

When EDAT-1 does not apply, or when EDAT-1 applies but the STE-format control is zero, the page-index portion of the virtual address, in conjunction with the page-table origin contained in the segment-table entry, is used to select an entry from the page table.

The 64-bit address of the page-table entry in real or absolute storage is obtained by appending 11 zeros to the right of the page-table origin and adding the page index, with three rightmost and 53 leftmost zeros appended. A carry out of bit position 0 cannot occur. All 64 bits of the address are used, regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

All eight bytes of the page-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address generated for fetching the page-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the unit of operation is suppressed.

The entry fetched from the page table is a leaf-table entry and indicates the availability of the page. If the page is available, the entry contains the leftmost bits of the page-frame real address.

- The page-invalid bit, bit 53, is inspected to establish whether the corresponding page is available. If this bit is one, a page-translation exception is recognized.
- If bit position 52 contains a one, a translation-specification exception is recognized.
- When EDAT-1 does not apply, and the instruction-execution-protection facility is not installed in the machine, a translation-specification exception is also recognized if bit position 55 contains a one. When EDAT-1 applies, but either the instruction-execution-protection facility is not installed in the machine, or the facility is installed but not enabled, bit 55 is ignored. Regardless of whether EDAT-1 applies, if (a) the instruction-execution-protection facility is enabled, (b) the storage reference for which the translation is being performed is for instruction execution, and

(c) the instruction-execution-protection control, bit 55, is one in the leaf-table entry, then a protection exception is recognized.

- If the DAT-protection bit, bit 54, is one either in the segment-table entry used in the translation, in the page-table entry, or, when EDAT-1 applies, in any region-table entry used during the translation, and the storage reference for which the translation is being performed is a store, a protection exception is recognized.

Formation of the Real and Absolute Addresses

When the effective address-space-control element (ASCE) is a real-space designation, bits 0-63 of the virtual address are used directly as the real storage address. The real address is then subjected to prefixing to form an absolute address.

When the effective ASCE is not a real-space designation and no exceptions in the translation process are encountered, the following conditions apply:

- When EDAT-2 applies and the RTTE-format control is one, the region-frame absolute address and the segment-index, page-index and byte-index portions of the virtual address are concatenated, left to right, respectively, to form the absolute address which corresponds to the virtual address.
- When EDAT-1 applies and the STE-format control is one, the segment-frame absolute address and the page-index and byte-index portions of the virtual address are concatenated, left to right, respectively, to form the absolute address which corresponds to the virtual address.
- When EDAT-1 does not apply, or when EDAT-1 applies but the STE format control is zero, the page-frame real address is obtained from the page-table entry. The page-frame real address and the byte-index portion of the virtual address are concatenated, with the page-frame real address forming the leftmost part. The result is the real storage address which corresponds to the virtual address. The real address is then subjected to prefixing to form an absolute address.

All 64 bits of the real and absolute addresses are used, regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

Recognition of Exceptions during Translation

Invalid addresses and invalid formats can cause exceptions to be recognized during the translation process. Exceptions are recognized when information contained in table entries is used for translation and is found to be incorrect. When the storage reference for which the translation is being performed is a store, a protection exception can be recognized when the DAT-protection bit is on in the segment, or page-table entry, or, when EDAT-1 applies, in any region-table entry used in the translation. When the instruction-execution-protection facility is enabled, a protection exception can be recognized when the instruction-execution-protection control is one in the leaf-table entry used in the translation.

The information pertaining to DAT is considered to be used when an instruction is executed with DAT on, when the enhanced-monitor counting array is accessed, or when COMPARE AND REPLACE DAT TABLE ENTRY, INVALIDATE DAT TABLE ENTRY, INVALIDATE PAGE TABLE ENTRY, LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, or STORE REAL ADDRESS is executed. The information is not considered to be used when the PSW specifies DAT on but an I/O, external, restart, or machine-check interruption occurs before an instruction is executed, or when the PSW specifies the wait state. Only that information required in order to translate a virtual address is considered to be in use during the translation of that address, and, in particular, addressing exceptions that would be caused by the use of an address-space-control element are not recognized when that address-space-control element is not the one actually used in the translation.

A list of translation exceptions, with the action taken for each exception and the priority in which the exceptions are recognized when more than one is applicable, is provided in “Recognition of Access Exceptions” on page 6-47.

Translation-Lookaside Buffer

To enhance performance, the dynamic-address-translation mechanism normally is implemented such that some of the information specified in the region tables, segment tables, and page tables is maintained in a special buffer, referred to as the translation-lookaside buffer (TLB). The CPU necessarily refers to a DAT-table entry in real or absolute storage only for the initial access to that entry. This informa-

tion may be placed in the TLB, and subsequent translations may be performed by using the information in the TLB. For consistency of operation, the virtual-equals-real translation specified by a real-space designation also may be performed by using information in the TLB. The presence of the TLB affects the translation process to the following extent:

1. A modification of the contents of a table entry in real or absolute storage does not necessarily have an immediate effect, if any, on the translation.

2. A region-first-table origin, region-second-table origin, region-third-table origin, segment-table origin, or real-space token origin in an address-space-control element (ASCE) may select a TLB entry that was formed by means of an ASCE containing a table or token origin of the same value even when the two origins were obtained from either (a) ASCEs having differing real-space controls or (b) ASCEs having differing designation-type controls (when the real-space controls are zero).

3. The comparison against the table length in an address-space-control element may be omitted if a TLB equivalent of the designated table is used. In a multiple-CPU configuration, each CPU has its own TLB.

Entries within the TLB are not explicitly addressable by the program.

Information is not necessarily retained in the TLB under all conditions for which such retention is permissible. Furthermore, information in the TLB may be cleared under conditions additional to those for which clearing is mandatory.

TLB Structure

The description of the logical structure of the TLB covers the implementation by all systems operating as defined by z/Architecture. The TLB entries are considered as being of six types: TLB region-first-table entries, TLB region-second-table entries, TLB region-third-table entries (collectively called TLB region-table entries), TLB segment-table entries, TLB page-table entries, and TLB real-space entries. A TLB region-table entry, TLB segment-table entry, or TLB page-table entry is considered as containing within it both the information obtained from the table entry in real or absolute storage and the attributes used to fetch this information from storage. A TLB real-space entry is considered as containing a page-frame real address and the real-space token origin and region, segment, and page indexes used to form the entry.

Any applicable TLB region-table entries, the TLB segment-table entry, and the TLB page-table entry may be merged into a single entry called a TLB composite table entry. In a similar manner, an implementation may combine any contiguous subset of table levels. When this happens, the intermediate table origins, offsets, and lengths need not be buffered.

An equivalent to the TLB combined region-and-segment-table entry (CRSTE), described in previous versions of the architecture, may be formed which maps a table origin, region index, segment index, and common-segment bit to a segment-frame absolute address or page-table origin (and other designated fields).

The token origin in a TLB real-space entry is indistinguishable from the table origin in a TLB composite-, region-, or segment-table entry.

Note: The following sections describe the conditions under which information may be placed in the TLB, the conditions under which information from the TLB may be used for address translation, and how changes to the translation tables affect the translation process.

Formation of TLB Entries

The formation of TLB region-table entries, TLB segment-table entries and TLB page-table entries from table entries in real or absolute storage, and the effect of any manipulation of the contents of table entries in storage by the program, depend on whether the entries in storage are attached to a particular CPU and on whether the entries are valid.

The *attached* state of a table entry denotes that the CPU to which it is attached can attempt to use the table entry for implicit address translation, except that a table entry for the primary or home address space may be attached even when the CPU does not fetch from either of those spaces. A table entry may be attached to more than one CPU at a time.

The *valid* state of a table entry denotes that the region, segment, or page associated with the table entry is available. An entry is valid when the region-

invalid, segment-invalid, or page-invalid bit in the entry is zero.

A region-table entry, segment-table entry, or page-table entry may be placed in the TLB whenever the entry is attached and valid and would not cause a translation-specification exception if used for translation.

The region-table entries, if any, and the segment-table entry, if any, used to translate a virtual address are called a *translation path*. The highest-level table entry in a translation path is attached when it is within a table designated by an attaching address-space-control element (ASCE). "Within a table" means as determined by the origin and length fields in the ASCE. An ASCE is an attaching ASCE when all of the following conditions are met:

1. The current PSW specifies DAT on.
2. The current PSW contains no errors that would cause an early specification exception to be recognized.
3. The ASCE meets the requirements in a, b, c, or d, below.

   a. The ASCE is the primary ASCE in control register 1.

   b. The ASCE is the secondary ASCE in control register 7, and any of the following requirements is met:

      - The CPU is in the secondary-space mode or access-register mode.
      - The CPU is in the primary-space mode, and the secondary-space control, bit 37 of control register 0, is one.
      - The M_4 operand of LOAD PAGE TABLE ENTRY ADDRESS explicitly allows access to the secondary space or explicitly allows access-register translation.
      - Either operand's operand-access control (OAC) of MOVE WITH OPTIONAL SPECIFICATIONS explicitly allows access to the secondary space or explicitly allows access-register translation.

      For further explanation of the term "explicitly allows," used in the above two items, see the programming note, below.

   c. The ASCE is in either an attached and valid ASN-second-table entry (ASTE) or a usable ALB ASTE, and any of the following requirements is met:

      - The CPU is in the access-register mode.
      - The M_4 field of LOAD PAGE TABLE ENTRY ADDRESS explicitly allows access-register translation to be performed.
      - Either operand's OAC of MOVE WITH OPTIONAL SPECIFICATIONS explicitly allows access-register translation to be performed.

      See the programming note, below, for further explanation. See "ART-Lookaside Buffer" on page 5-64 for the meaning of the terminology used here.

   d. The ASCE is the home ASCE in control register 13.

Regardless of whether DAT is on or off, an ASCE is also an attaching ASCE when the current PSW contains no errors that would cause an early specification exception to be recognized, and any of the following conditions is met:

- The home ASCE is considered to be an attaching ASCE when a monitor-event counting operation occurs.

Each of the remaining table entries in a translation path is attached when it is within the table designated either by an attached and valid entry of the next higher level which would not cause a translation-specification exception if used for translation or by a usable TLB entry of the next higher level. "Within the table" means as determined by the origin, offset, and length fields in the next-higher-level entry. A usable TLB entry is explained in the next section.

A page-table entry is attached when it is within the page table designated by either an attached and valid segment-table entry that would not cause a translation-specification exception if used for translation or a usable TLB segment-table entry.

A region-table entry or segment-table entry causes a translation-specification exception if the table-type bits, bits 60 and 61, in the entry are inconsistent with the level at which the entry would be encountered

when using the translation path in the translation process. A segment-table entry also causes a translation-specification exception if the private-space-control bit is one in the address-space-control element used to select it and the common-segment bit is one in the entry. When EDAT-2 applies, a region-third-table entry also causes a translation-specification exception if the private-space-control bit is one in the address-space-control element used to select it and the common-region bit is one in the entry. A page-table entry causes a translation-specification exception if bit 52 in the entry is one. When EDAT-1 does not apply, a page-table entry also causes a translation-specification exception if bit 55 in the entry is one.

When the instruction-execution-protection facility is enabled, and the instruction-execution-protection control is one in the leaf table entry, a TLB entry or composite TLB entry of the appropriate level may be formed even if a protection exception is recognized. Alternatively, when an instruction-execution-protection exception is recognized, the formation of the TLB entry may be bypassed.

A TLB real-space entry may be formed whenever an attaching real-space designation exists. The entry is formed using the real-space token origin in the designation and any value of bits 0-51 of a virtual address.

Subject to the attached and valid constraints defined above, the CPU may form TLB entries in anticipation of future storage references or as a result of the speculative execution of instructions. See "Overlapped Operation of Instruction Execution" on page 5-114 for additional details.

Programming Note: In the above list of conditions for an ASCE to be attaching, item 3.b and 3.c use the term "explicitly allows …", as explained below:

- LPTEA explicitly allows access to the secondary space when the M_4 field is 0010 binary.
- LPTEA explicitly allows ART to be performed when the M_4 field is 0001 binary.
- Either operand of MVCOS explicitly allows access to the secondary space when bits 8-9 of the operand's OAC are 10 binary, bit 15 of the OAC is one, and the secondary-space control, bit 37 of control register 0, is one.
- Either operand of MVCOS explicitly allows ART to be performed when bits 8-9 of the operand's OAC are 01 binary, and bit 15 of the OAC is one.

Use of TLB Entries

The *usable* state of a TLB entry denotes that the CPU can attempt to use the TLB entry for implicit address translation. A usable TLB entry attaches the next-lower-level table, if any, and may be usable for a particular instance of implicit address translation.

With reference to a TLB entry, the term "current level" refers to the level of translation table (region first table, region second table, region third table, segment table, or page table) from which the TLB entry was formed. Likewise, the "current-level index" is that portion of the virtual address used as an index into the current level of translation table.

A TLB region- or segment-table entry is in the usable state when all of the following conditions are met:

1. The current PSW specifies DAT on.
2. The current PSW contains no errors that would cause an early specification exception to be recognized.
3. The TLB entry meets at least one of the following requirements:
   a. The common-region bit is one in a TLB region-third-table entry.
   b. The common-segment bit is one in a TLB segment-table entry.
   c. The ASCE-table-origin (ASCETO) field in the TLB entry matches the table- or token-origin field in an attaching address-space-control element.
   d. The TLB entry is a TLB region-second-table, region-third-table, or segment-table entry, and the current-level table-origin field in the TLB entry matches one of the following:
      - The table-origin field in an attaching ASCE which directly designates the current table level (as indicated by the R and DT bits)
      - The table-origin field in an attached region-table entry of the next higher level

- The table-origin field of the same level in a usable TLB region-table entry of the next higher level

A TLB region-table entry may be used for a particular instance of implicit address translation only when the entry is in the usable state, the current-level index field in the TLB entry matches the corresponding index field of the virtual address being translated, and any of the following conditions is met:

1. The ASCE-table-origin (ASCETO) field in the TLB entry matches the table- or token-origin field in the address-space-control element being used in the translation, and the portion of the virtual address being translated which is to the left of the current-level index matches the corresponding index fields in the TLB entry.
2. The address-space-control element being used in the translation designates a table of the current level, and the current-level table-origin field in the TLB entry matches the table origin in that address-space-control element.
3. The current-level table-origin field in the TLB entry matches the table origin of the same level in the next-higher-level table entry or TLB entry being used in the translation.
4. For a TLB region-third-table entry, the common-region bit is one in the TLB entry, and the region-first-index and region-second-index fields in the TLB entry matches those of the virtual address being translated.

However, when EDAT-2 applies, the TLB region-third-table entry is not used if the common-region bit is one in the entry and either the private-space-control bit is one in the address-space-control element being used in the translation or that address-space-control element is a real-space designation. In both these cases, the TLB entry is not used even if the ASCE-table-origin (ASCETO) field in the entry and the table- or token-origin field in the address-space-control element match.

A TLB segment-table entry may be used for a particular instance of implicit address translation only when the entry is in the usable state, the segment-index field in the TLB entry matches that of the virtual address being translated, and any of the following conditions is met:

1. The ASCE-table-origin (ASCETO) field in the TLB entry matches the table- or token-origin field in the address-space-control element being used in the translation, and the region-index field in the TLB entry matches that of the virtual address being translated.
2. The segment-table-origin field in the TLB entry matches the table-origin field in the address-space-control element being used in the translation, and that address-space-control element is a segment-table designation.
3. The segment-table-origin field in the TLB entry matches the segment-table-origin field in the region-third-table entry or TLB region-third-table entry being used in the translation.
4. The common-segment bit is one in the TLB entry, and the region-index field in the TLB entry matches that of the virtual address being translated.

However, the TLB segment-table entry is not used if the common-segment bit is one in the entry and either the private-space-control bit is one in the address-space-control element being used in the translation or that address-space-control element is a real-space designation. In both these cases, the TLB entry is not used even if the ASCE-table-origin (ASCETO) field in the entry and the table- or token-origin field in the address-space-control element match.

A TLB page-table entry may be used for a particular instance of implicit address translation only when the page-table-origin field in the entry matches the page-table-origin field in the segment-table entry or TLB segment-table entry being used in the translation and the page-index field in the TLB page-table entry matches the page index of the virtual address being translated.

A TLB real-space entry may be used for implicit address translation only when the token-origin field in the TLB entry matches the table- or token-origin field in the address-space-control element being used in the translation and the region-index, segment-index, and page-index fields in the TLB entry match those of the virtual address being translated

The operand addresses of LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, and STORE REAL ADDRESS, and accesses to the enhanced-monitor counting array may be translated

with the use of the TLB contents whether DAT is on or off. However, for LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, and STORE REAL ADDRESS, TLB entries still are formed only if DAT is on.

Programming Notes:

1. Although contents of a table entry may be copied into the TLB only when the table entry is both attached and valid, the copy may remain in the TLB even when the table entry itself is no longer attached or valid.

2. Except when translations are performed as a result of enhanced-monitor counting operations, no contents can be copied into the TLB when DAT is off because the table entries at this time are not attached. In particular, translation of the operand address of LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS and STORE REAL ADDRESS with DAT off does not cause entries to be placed in the TLB.

   Conversely, when DAT is on, information may be copied into the TLB from all translation-table entries that could be used for address translation, given the current translation parameters, the setting of the address-space-control bits, and the contents of the access registers. The loading of the TLB does not depend on whether the entry is used for translation as part of the execution of the current instruction, and such loading can occur when the CPU is in the wait state.

3. More than one copy of contents of a table entry may exist in the TLB. For example, some implementations may cause a copy of contents of a valid table entry to be placed in the TLB for the table origin in each address-space-control element by which the entry becomes attached.

Modification of Translation Tables

When an attached and invalid table entry is made valid and no entry usable for translation of the associated virtual address is in the TLB, the change takes effect no later than the end of the current unit of operation. Similarly, when an unattached and valid table entry is made attached and no usable entry for the associated virtual address is in the TLB, the change takes effect no later than the end of the current unit of operation.

When a valid and attached table entry is changed, and when, before the TLB is cleared of entries that qualify for substitution for that entry, an attempt is made to refer to storage by using a virtual address requiring that entry for translation, unpredictable results may occur, to the following extent. The use of the new value may begin between instructions or during the execution of an instruction, including the instruction that caused the change. Moreover, until the TLB is cleared of entries that qualify for substitution for that entry, the TLB may contain both the old and the new values, and it is unpredictable whether the old or new value is selected for a particular access. If both old and new values of a higher-level table entry are present in the TLB, a lower-level table entry may be fetched by using one value and placed in the TLB associated with the other value. If the new value of the entry is a value that would cause an exception, the exception may or may not cause an interruption to occur. If an interruption does occur, the result fields of the instruction may be changed even though the exception would normally cause suppression or nullification.

Entries are cleared from the TLB in accordance with the following rules:

1. All entries are cleared from the TLB by the execution of PURGE TLB or SET PREFIX and by CPU reset.

2. All entries may be cleared from all TLBs in the configuration by the execution of COMPARE AND SWAP AND PURGE by any of the CPUs in the configuration, depending on a bit in a general register used by the instruction.

3. Selected entries are cleared from all TLBs in the configuration by the execution of COMPARE AND REPLACE DAT TABLE ENTRY, INVALIDATE DAT TABLE ENTRY or INVALIDATE PAGE TABLE ENTRY by any of the CPUs in the configuration when the local-TLB-clearing facility is not installed, or when the facility is installed but the instruction does not specify local clearing.

   Only the entries in the TLB of the CPU executing COMPARE AND REPLACE DAT TABLE ENTRY, INVALIDATE DAT TABLE ENTRY or INVALIDATE PAGE TABLE ENTRY are cleared when the local-TLB-clearing facility is installed and the instruction specifies local clearing.

4. Some or all TLB entries may be cleared at times other than those required by the preceding rules.

Programming Notes:

1. Entries in the TLB may continue to be used for translation after the table entries from which they have been formed have become unattached or invalid. These TLB entries are not necessarily removed unless explicitly cleared from the TLB.

   A change made to an attached and valid entry or a change made to a table entry that causes the entry to become attached and valid is reflected in the translation process for the next instruction, or earlier than the next instruction, unless a TLB entry qualifies for substitution for that table entry. However, a change made to a table entry that causes the entry to become unattached or invalid is not necessarily reflected in the translation process until the TLB is cleared of entries that qualify for substitution for that table entry.

2. Exceptions associated with dynamic address translation may be established by a pretest for operand accessibility that is performed as part of the initiation of instruction execution. Consequently, a region-first-translation, region-second-translation, region-third-translation, segment-translation, or page-translation exception may be indicated when a table entry is invalid at the start of execution even if the instruction would have validated the table entry it uses and the table entry would have appeared valid if the instruction was considered to process the operands one byte at a time.

3. A change made to an attached table entry, except to set the I bit to zero or to alter the rightmost byte of a page-table entry, may produce unpredictable results if that entry is used for translation before the TLB is cleared of all copies of contents of that entry. The use of the new value may begin between instructions or during the execution of an instruction, including the instruction that caused the change. When an instruction, such as MOVE (MVC), makes a change to an attached table entry, including a change that makes the entry invalid, and subsequently uses the entry for translation, a changed entry is being used without a prior clearing of the entry from the TLB, and the associated unpredictability of result values and of exception recognition applies.

   Manipulation of attached table entries may cause spurious table-entry values to be recorded in a TLB. For example, if changes are made piecemeal, modification of a valid attached entry may cause a partially updated entry to be recorded, or, if an intermediate value is introduced in the process of the change, a supposedly invalid entry may temporarily appear valid and may be recorded in the TLB. Such an intermediate value may be introduced if the change is made by an I/O operation that is retried, or if an intermediate value is introduced during the execution of a single instruction.

   As another example, if a segment-table entry is changed to designate a different page table and used without clearing the TLB, the new page-table entries may be fetched and associated with the old page-table origin. In such a case, execution of INVALIDATE PAGE TABLE ENTRY designating the new page-table origin will not necessarily clear the page-table entries fetched from the new page table.

4. To facilitate the manipulation of page tables, the INVALIDATE PAGE TABLE ENTRY instruction is provided. This instruction sets the I bit in a page-table entry to one and clears one or more TLBs in the configuration of entries formed from those table entries as follows:

   a. All TLBs in the configuration are cleared when the local-TLB-clearing facility is not installed, or when the facility is installed and the instruction specifies the clearing of all TLBs (that is, the local-clearing control in the instruction is zero).

   b. Only the TLB in the CPU executing the INVALIDATE PAGE TABLE ENTRY instruction is cleared when the local-TLB-clearing facility is installed and the instruction specifies the clearing of only the local TLB (that is, the local-clearing control in the instruction is one).

   The following aspects of the TLB operation should be considered when using INVALIDATE PAGE TABLE ENTRY. (See also the programming notes for INVALIDATE PAGE TABLE ENTRY.)

   a. INVALIDATE PAGE TABLE ENTRY should be executed before making any change to a page-table entry other than changing the rightmost byte; otherwise, the selective-clearing portion of INVALIDATE PAGE

TABLE ENTRY may not clear the TLB copies of the entry.

b. Invalidation of all the page-table entries within a page table by means of INVALIDATE PAGE TABLE ENTRY does not necessarily clear the TLB of any segment-table entry designating the page table. When it is desired to invalidate and clear the TLB of a region- or segment-table entry, the rules in note 5 below must be followed.

   Similarly, invalidation of all the lower-level table entries within a region or segment table by means of INVALIDATE DAT TABLE ENTRY does not necessarily clear the TLB of any higher-level table entry designating the lower-level table. When it is desired to invalidate and clear the TLB of a higher-level table entry, the rules in note 5 below must be followed.

c. When a large number of page-table entries are to be invalidated at a single time, the overhead involved in using COMPARE AND SWAP AND PURGE (one that purges the TLB), INVALIDATE DAT TABLE ENTRY, or PURGE TLB and in following the rules in note 5 below may be less than in issuing INVALIDATE PAGE TABLE ENTRY for each page-table entry.

5. Manipulation of table entries should be in accordance with the following rules. If these rules are complied with, translation is performed as if the table entries from real or absolute storage were always used in the translation process.

   a. A valid table entry must not be changed while it is attached to any CPU and may be used for translation by that CPU except to (1) invalidate the entry by using INVALIDATE PAGE TABLE ENTRY or INVALIDATE DAT TABLE ENTRY, (2) alter bits 56-63 of a page-table entry, (3) make a change by means of a COMPARE AND SWAP AND PURGE instruction that purges the TLB, or (4) replace an entry by using COMPARE AND REPLACE DAT TABLE ENTRY.

   b. When any change is made to an invalid table entry in such a way as to allow intermediate valid values to appear in the entry, each CPU to which the entry is attached must be caused to purge its TLB after the change occurs and prior to the use of the entry for implicit address translation by that CPU.

   c. When any change is made to an offset or length specified for a table, each CPU which may have a TLB entry formed from a table entry that no longer lies within its table must be caused to purge its TLB after the change occurs and prior to the use of the table for implicit translation by that CPU.

   Note that when an invalid page-table entry is made valid without introducing intermediate valid values, the TLB need not be cleared in a CPU which does not have any TLB entries formed from that entry. Similarly, when an invalid region-table or segment-table entry is made valid without introducing intermediate valid values, the TLB need not be cleared in a CPU which does not have any TLB entries formed from that validated entry and which does not have any TLB entries formed from entries in a page table attached by means of that validated entry.

   The execution of PURGE TLB, COMPARE AND SWAP AND PURGE, or SET PREFIX may have an adverse effect on the performance of some models. Use of these instructions should, therefore, be minimized in conformance with the above rules.

6. In addition to the constraints described in programming note 3 on page 3-68, the following considerations are in effect when EDAT-1 applies:

   a. When the STE-format and ACCF-validity controls are both one, it is unpredictable whether the CPU inspects the access-control bits and the fetch-protection bit in the segment-table entry or in the storage key of the corresponding 4 K-byte block for any given key-controlled-protection check. Therefore, the program should ensure that the access-control bits and fetch-protection bit in the segment-table entry are identical to the respective fields in all 256 storage keys for the constituent 4 K-byte blocks of the segment, before setting the invalid bit in the STE to zero.

   b. Prior to changing the ACCF-validity control, the access-control bits, or the fetch-protection bit in the segment-table entry, and prior to changing the access-control bits or fetch-protection bit in any of the segment's 256

storage keys, the program should first set the invalid bit to one in the segment-table entry and clear all entries in all TLBs in the configuration, as described previously in this section.

When EDAT-2 applies, the following additional considerations are in effect:

a. When the RTTE-format control and ACCF-validity control are both one, it is unpredictable whether the CPU inspects the access-control bits and the fetch-protection bit in the region-third-table entry or in the storage key of the corresponding 4 K-byte block for any given key-controlled-protection check. Therefore, the program should ensure that the access-control bits and fetch-protection bit in the region-third-table entry are identical to the respective fields in all 524,288 storage keys for the constituent 4 K-byte blocks of the region, before setting the invalid bit in the RTTE to zero.

b. Prior to changing the ACCF-validity control, the access-control bits, or the fetch-protection bit in the region-third-table entry, and prior to changing the access-control bits or fetch-protection bit in any of the region's 524,288 storage keys, the program should first set the invalid bit to one in the region-third-table entry and clear all entries in all TLBs in the configuration, as described previously in this section.

Failure to observe these procedures may lead to unpredictable results, possibly including a delayed-access-exception machine-check or failure to record a change.

Address Summary

Addresses Translated

Most addresses that are explicitly specified by the program and are used by the CPU to refer to storage are instruction or logical addresses and are subject to implicit translation when DAT is on. Analogously, the corresponding addresses indicated to the program on an interruption or as the result of executing an instruction are instruction or logical addresses. The operand address of LOAD PAGE TABLE ENTRY ADDRESS, LOAD REAL ADDRESS, and STORE REAL ADDRESS is explicitly translated, regardless of whether the PSW specifies DAT on or off.

Translation is not applied to quantities that are formed from the values specified in the B and D fields of an instruction but that are not used to address storage. This includes operand addresses in EXTRACT CPU ATTRIBUTE, LOAD ADDRESS, LOAD ADDRESS EXTENDED, MONITOR CALL, the second operand address in LOAD ADDRESS SPACE PARAMETERS and SHIFT AND ROUND PACKED, and the shifting instructions. This also includes the addresses in control registers 10 and 11 designating the starting and ending locations for PER.

Except as noted below, the addresses explicitly designating storage keys (operand addresses in INSERT STORAGE KEY EXTENDED, PERFORM FRAME MANAGEMENT FUNCTION, RESET REFERENCE BIT EXTENDED, and SET STORAGE KEY EXTENDED) are real addresses.

- For INSERT VIRTUAL STORAGE KEY, the address designating the storage key is virtual.
- For INSERT REFERENCE BITS MULTIPLE, for PERFORM FRAME MANAGEMENT FUNCTION when the set-key control is 1 and the frame-size code is 1 or 2, for RESET REFERENCE BITS MULTIPLE, and for SET STORAGE KEY EXTENDED when the multiple-block control is one, the address designating the storage keys is absolute.
- For TEST PROTECTION, the address designating the block to be tested is logical.

The addresses implicitly used by the CPU for such sequences as interruptions are real addresses.

The addresses used by channel programs to transfer data and to refer to CCWs, IDAWs, or MIDAWs are absolute addresses.

The handling of storage addresses associated with DIAGNOSE is model-dependent.

The processing of addresses, including dynamic address translation and prefixing, is discussed in "Address Types" on page 3-4. Prefixing, when provided, is applied after the address has been translated by means of the dynamic-address-translation

facility. For a description of prefixing, see “Prefixing” on page 3-21.

Handling of Addresses

The handling of addresses is summarized in Figure 3-17. This figure lists all addresses that are encountered by the program and specifies the address type.

Virtual Addresses

- Address of storage operand for INSERT VIRTUAL STORAGE KEY
- Operand address in LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, and STORE REAL ADDRESS
- Addresses of storage operands for MOVE TO PRIMARY, MOVE TO SECONDARY, and MOVE WITH OPTIONAL SPECIFICATIONS
- Address stored in the doubleword at real location 168 on a program interruption for ASCE-type, region-first-translation, region-second-translation, region-third-translation, segment-translation, or page-translation exception
- Linkage-stack-entry address in control register 15
- Backward stack-entry address in linkage-stack header entry
- Forward-section-header address in linkage-stack trailer entry
- Trap-control-block address in dispatchable-unit-control table
- Trap-save-area address and trap-program address in trap control block
- Address of the enhanced-monitor counter array in the doubleword at real location 256

Instruction Addresses

- Instruction address in PSW
- Branch address
- Target of EXECUTE and EXECUTE RELATIVE LONG
- Address stored in the doubleword at real location 152 on a program interruption for PER
- Address placed in general register by BRANCH AND LINK, BRANCH AND SAVE, BRANCH AND SAVE AND SET MODE, BRANCH AND STACK, BRANCH IN SUBSPACE GROUP, BRANCH RELATIVE AND SAVE, BRANCH RELATIVE AND SAVE LONG, and PROGRAM CALL
- Address used in general register by BRANCH AND STACK
- Address placed in general register by BRANCH AND SET AUTHORITY executed in reduced-authority state
- Addresses used by BRANCH PREDICTION PRELOAD and BRANCH PREDICTION RELATIVE PRELOAD
- Aborted-transaction instruction address (in the transaction diagnostic block)

Logical Addresses

- Addresses of storage operands for instructions not otherwise specified
- Address placed in general register 1 by EDIT AND MARK and TRANSLATE AND TEST
- Addresses in general registers updated by MOVE LONG, MOVE LONG EXTENDED, COMPARE LOGICAL LONG, and COMPARE LOGICAL LONG EXTENDED
- Addresses in general registers updated by CHECKSUM, COMPARE AND FORM CODEWORD, and UPDATE TREE
- Address for TEST PENDING INTERRUPTION when the second-operand address is nonzero
- Address of parameter list of RESUME PROGRAM
- Address of the TBEGIN-specified transaction-diagnostic block

Figure 3-17. Handling of Addresses (Part 1 of 3).

Real Addresses

- Address of storage key for INSERT STORAGE KEY EXTENDED, and RESET REFERENCE BIT EXTENDED
- Address of storage key for SET STORAGE KEY EXTENDED (when the enhanced-DAT facility is not installed, or when the enhanced-DAT facility is installed but the multiple-block control is zero)
- Address of second operand for PERFORM FRAME MANAGEMENT FUNCTION when frame-size code is 0.
- Address of storage operand for LOAD USING REAL ADDRESS, STORE USING REAL ADDRESS, and TEST BLOCK
- The translated address generated by LOAD REAL ADDRESS and STORE REAL ADDRESS
- Page-frame real address in page-table entry
- Trace-entry address in control register 12
- ASN-first-table origin in control register 14
- ASN-second-table origin in ASN-first-table entry
- Authority-table origin in ASN-second-table entry, except when used by access-register translation
- Linkage-table origin in primary ASN-second-table entry
- Entry-table origin in linkage-table entry
- Dispatchable-unit-control-table origin in control register 2
- Primary-ASN-second-table-entry origin in control register 5
- Base-ASN-second-table-entry origin and subspace-ASN-second-table-entry origin in dispatchable-unit control table
- ASN-second-table-entry address in entry-table entry and access-list entry

Permanently Assigned Real Addresses

- Address of the doubleword into which TEST PENDING INTERRUPTION stores when the second-operand address is zero
- Addresses of PSWs, interruption codes, and the associated information used during interruption
- Addresses used for machine-check logout and save areas
- Address of STORE FACILITY LIST operand

Addresses which Are Unpredictably Real or Absolute

- Region-first-table origin, region-second-table origin, region-third-table origin, or segment-table origin in control registers 1, 7, and 13, in access-register-specified address-space-control element, and in region-first-table entry, region-second-table entry, or region-third-table entry
- Page-table origin in segment-table entry and in INVALIDATE PAGE TABLE ENTRY
- Address of segment-table entry or page-table entry provided by LOAD REAL ADDRESS
- Address of region-first-table entry, region-second-table entry, region-third-table entry, segment-table entry, or page-table entry provided by LOAD PAGE-TABLE-ENTRY ADDRESS
- The dispatchable-unit or primary-space access-list origin and the authority-table origin (in the ASTE designated by the ALE used) used by access-register translation

Figure 3-17. Handling of Addresses (Part 2 of 3).

Absolute Addresses

- Prefix value
- Channel-program address in ORB
- Data address in CCW
- IDAW address in a CCW specifying indirect data addressing
- MIDAW address in a CCW specifying modified indirect data addressing
- CCW address in a CCW specifying transfer in channel
- Data address in IDAW
- Data address in MIDAW
- Measurement-block origin specified by SET CHANNEL MONITOR
- Address limit specified by SET ADDRESS LIMIT
- Addresses used by the store-status-at-address SIGNAL PROCESSOR order
- Address of storage key for INSERT REFERENCE BITS MULTIPLE and RESET REFERENCE BITS MULTIPLE
- Address of storage key for SET STORAGE KEY EXTENDED (when the enhanced-DAT facility is installed and the multiple-block control is one)
- Address of second operand for PERFORM FRAME MANAGEMENT FUNCTION when frame-size code is 1 or 2.
- Failing-storage address stored in the doubleword at real location 248
- CCW address in SCSW

Permanently Assigned Absolute Addresses

- Addresses used for the store-status function
- Addresses of PSW and first two CCWs used for initial program loading

Addresses Not Used to Reference Storage

- PER starting address in control register 10
- PER ending address in control register 11
- Address stored in the doubleword at real location 176 for a monitor event
- Address in shift instructions and other instructions specified not to use the address to reference storage
- Real-space token origin in real-space designation

Figure 3-17. Handling of Addresses (Part 3 of 3).

Assigned Storage Locations

Figure 3-19 on page 3-87 shows the format and extent of the assigned locations in storage in both the z/Architecture architectural mode and ESA/390-compatibility mode.

Assigned Storage Locations in the z/Architecture Architectural Mode

0-7 . . . . . Absolute Address

IPL PSW: The first eight bytes read during a CCW-type initial-program-loading (IPL) initial-read operation are stored at locations 0-7. The list-directed IPL process also stores eight bytes at locations 0-7. The contents of these locations are used to form the new PSW at the completion of the IPL operation. These locations may also be used for temporary storage at the initiation of the IPL operation.

When the configuration-z/Architecture-architectural-mode (CZAM) facility is not installed, the IPL new PSW has the format of an ESA/390 PSW. When the CZAM facility is installed, the IPL PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

8-15 (08-0F hex) . . . . . Absolute Address

CCW-Type IPL CCW1: Bytes 8-15 read during a CCW-type initial-program-loading (IPL) initial-read operation are stored at locations 8-15. The contents of these locations are ordinarily used as the next CCW in an IPL CCW chain after completion of the IPL initial-read operation.

16-23 (10-17 hex). . . . . .Absolute Address

CCW-Type IPL CCW2: Bytes 16-23 read during a CCW-type initial-program loading (IPL) initial-read operation are stored at locations 16-23. The contents of these locations may be used as another CCW in the IPL CCW chain to follow IPL CCW1.

LD IPL Machine-Loader Execution-Space Size: During a list-directed initial-program loading (IPL) operation, the machine-loader execution-space size is stored at locations 16-19.

Available for Use by Programming: Locations 16-19 are available for use by programming after IPL.

LD IPL System-Parameter-Block Address: During a list-directed initial-program loading (IPL) operation, the absolute address of the first byte of the system-parameter block is stored at locations 20-23.

128-131 (80-83 hex). . . . . Real Address

External-Interruption Parameter: During an external interruption due to service signal or timing alert, the parameter associated with the interruption is stored at locations 128-131.

132-133 (84-85 hex). . . . . Real Address

CPU Address: During an external interruption due to malfunction alert, emergency signal, or external call, the CPU address associated with the source of the interruption is stored at locations 132-133. For all other external-interruption conditions, zeros are stored at locations 132-133.

134-135 (86-87 hex). . . . . Real Address

External-Interruption Code: During an external interruption, the interruption code is stored at locations 134-135.

136-139 (88-8B hex) . . . . . Real Address

Supervisor-Call-Interruption Identification: During a supervisor-call interruption, the instruction-length code is stored in bit positions 5 and 6 of location 137, and the interruption code is stored at locations 138-139. Zeros are stored at location 136 and in the remaining bit positions of location 137.

140-143 (8C-8F hex) . . . . . Real Address

Program-Interruption Identification: During a program interruption, the instruction-length code is stored in bit positions 5 and 6 of location 141, and the interruption code is stored at locations 142-143. Zeros are stored at location 140 and in the remaining bit positions of location 141.

When a transaction is aborted due to a program interruption, the instruction-length code is respective to the instruction at which the exception condition was detected.

144-147 (90-93 hex) . . . . . Real Address

Data-Exception Code (DXC): During a program interruption due to a data exception, the data-exception code is stored at location 147, and zeros are stored at locations 144-146. The DXC is described in “Data-Exception Code (DXC)” on page 6-17.

Vector-Exception Code (VXC): During a program interruption due to a vector-processing exception, the vector-exception code is stored at location 147, and zeros are stored at locations 144-146. The VXC is described in “Vector-Exception Code” on page 6-20.

148-149 (94-95 hex) . . . . . Real Address

Monitor-Class Number: During a program interruption due to a monitor event, the monitor-class number is stored at location 149, and zeros are stored at location 148.

150-151 (96-97 hex) . . . . . Real Address

PER Code: During a program interruption due to a PER event the PER code is stored in bit positions 0-7 of locations 150-151, and other information is or may be stored as described in “Identification of Cause” on page 4-29.

152-159 (98-9F hex) . . . . . Real Address

PER Address: During a program interruption due to a PER event, the PER address is stored at locations 152-159.

160 (A0 hex) . . . . . Real Address

Exception Access Identification (EAID): During a program interruption due to an ASCE-type, region-first-translation, region-second-translation, region-third-translation, segment-translation, or page-translation exception, an indication

of the address space to which the exception applies may be stored at location 160, as follows:

- If the access was a storage-operand reference that used an AR-specified address-space-control element, the number of the access register used is stored in bit positions 4-7 of location 160, and zeros are stored in bit positions 0-3.
- If the CPU was in the access-register mode and either (a) the access was an instruction fetch, including a fetch of the target of an execute-type instruction (EXECUTE or EXECUTE RELATIVE LONG), or (b) the access was to the second operand of the relative-long form of COMPARE, COMPARE HALFWORD, COMPARE LOGICAL, COMPARE LOGICAL HALFWORD, LOAD, LOAD HALFWORD, LOAD LOGICAL, LOAD LOGICAL HALFWORD, STORE or STORE HALFWORD, then zeros are stored at location 160.

In either of the two cases described above, storing at location 160 occurs regardless of the value stored in bit positions 62 and 63 of the translation-exception identification (TEID) at real locations 168-175. Otherwise, the contents of location 160 are unpredictable.

During a program interruption due to a protection exception, the contents of real location 160 are dependent on both the type of suppression-on-protection facility installed and the type of protection exception recognized. In the following cases, the information stored in real location 160 is the same as that described above for a page-translation exception; otherwise, the contents of real location 160 are unpredictable.

Facility	Protection-Exception Conditions Producing EAID Identical to that of a Page-Translation Exception
SOP	TEID bit 61 is one, and PSW bit 5 is one.
ESOP-1	TEID bit 61 is one.
ESOP-2	The protection code in TEID bits 56, 60, and 61 is nonzero, and the address in TEID bits 0-51 is virtual.
Explanation: SOP Basic suppression-on-protection ESOP Enhanced suppression-on-protection (facility 1 or 2)	

During a program interruption due to an ALEN-translation, ALE-sequence, ASTE-validity, ASTE-sequence, or extended-authority exception recognized during access-register translation, the number of the access register used is stored in bit positions 4-7 of location 160, and zeros are stored in bit positions 0-3. During a program interruption due to an ASTE-validity or ASTE-sequence exception recognized during a subspace-replacement operation, all zeros are stored at location 160.

During a program interruption due to an ASTE-instance exception recognized due to use of the ASN-and-LX-reuse facility, (1) a one is stored in bit position 2, and zeros are stored in bit positions 0, 1, and 3-7, if the exception was recognized after primary ASN translation in PROGRAM TRANSFER WITH INSTANCE or PROGRAM RETURN, or (2) a one is stored in bit position 3, and zeros are stored in bit positions 0-2 and 4-7, if the exception was recognized after secondary ASN translation in SET SECONDARY ASN WITH INSTANCE or PROGRAM RETURN.

161 (A1 hex) . . . . . .Real Address

PER Access Identification: During a program interruption due to a PER storage-alteration event or PER zero-address-detection event, an indication of the address space to which the event applies may be stored at location 161. If the access used an AR-specified address-space-control element, the number of the access register used is stored in bit positions 4-7 of location 161, and zeros are stored in bit positions 0-3. The contents of location 161 are unpredictable if the access did not use an AR-specified address-space-control element.

Programming Note: The PER ASCE identification may be inspected to determine whether the PER storage-alteration event or PER zero-address-detection event used an AR-specified ASCE. See “PER ASCE Identification (AI)” on page 4-31 for further details.

162 (A2 hex) . . . . . .Real Address

Operand Access Identification: When EDAT-1 does not apply, and a program interruption due to a page-translation exception is recognized by the MOVE PAGE instruction, the contents of the R_1 field of the instruction are stored in bit positions

0-3 of location 162, and the contents of the R_2 field are stored in bit positions 4-7. If the page-translation exception was recognized during the execution of an instruction other than MOVE PAGE, or if an ASCE-type, region-first-translation, region-second-translation, region-third-translation, or segment-translation exception was recognized, the contents of location 162 are unpredictable.

When EDAT-1 applies, and a program interruption due to a region-first-translation, region-second-translation, region-third-translation, segment-translation, or page-translation exception is recognized by the MOVE PAGE instruction, the contents of the R_1 and R_2 fields are stored in location 162 as described above. If any of the exceptions listed in the preceding sentence was recognized during the execution of an instruction other than MOVE PAGE, or if an ASCE-type exception was recognized, the contents of location 162 are unpredictable.

163 (A3 hex). . . . . .Absolute Address

Store-Status Architectural-Mode Identification: During the execution of the store-status operation, zeros are stored in bit positions 0-6 of location 163, and a one is stored in bit position 7. A zero stored in bit position 7 indicates the ESA/390 architectural mode, and a one indicates the z/Architecture architectural mode.

163 (A3 hex). . . . . Real Address

Machine-Check Architectural-Mode Identification: During a machine-check interruption, zeros are stored in bit positions 0-6 of location 163, and a one is stored in bit position 7. A zero stored in bit position 7 indicates the ESA/390 architectural mode, and a one indicates the z/Architecture architectural mode.

168-175 (A8-AF hex) . . . . . Real Address

Translation-Exception Identification (TEID): In the z/Architecture architectural mode, a translation-exception identification is stored at real locations 168-175 for various types of program interruptions, as described below.

Translation-Exception Identification for DAT Exceptions: During a program interruption due to an ASCE-type, region-first-translation, region-second-translation, region-third-translation, segment-translation, or page-translation exception, bits 0-51 of the virtual address causing the exception are stored in bit positions 0-51 of locations 168-175. This address is sometimes referred to as the translation-exception address.

When the access-exception-fetch/store-indication facility is installed, bits 52 and 53 of locations 168-175 are set as described below.

Access-Exception Fetch/Store Indication (AEFSI): Bit positions 52-53 of locations 168-175 contain an indication of whether the exception was due to a fetch or a store operation, as follows:

Bit 52	Bit 53	Meaning
0	0	Could not determine whether the exception was due to a fetch or store
0	1	Exception was due to a store operation
1	0	Exception was due to a fetch operation
1	1	Reserved

When the access-exception-fetch/store-indication facility is not installed, bits 52 and 53 of locations 168-175 are unpredictable.

The access-exception fetch/store indication is also stored for access-list-controlled-protection and DAT-protection exceptions. When the enhanced suppression-on-protection facility 2 is installed, the access-exception fetch-store indication is also stored for key-controlled-protection and low-address-protection exceptions. When the instruction-execution-protection facility is installed, the access-exception fetch-store indication is also stored for instruction-execution-protection exceptions.

For DAT-protection and access-list-controlled-protection exceptions, the field indicates a store operation. For instruction-execution-protection exceptions, the field indicates a fetch operation. For a storage-operand update reference, it is unpredictable whether a fetch or store access is indicated. When multiple access-exception conditions exist — for one or more operands — the condition recognized is determined based on the discussion of “Multiple Program-Interruption Conditions” on page 6-50.

The enhanced-suppression-on-protection facility 1 is a prerequisite for the access-exception-fetch/store-indication and instruction-execution-protection facilities.

Side-Effect-Access Indication: When the side-effect-access facility is installed, bit position 54 of locations 168-175 contains an indication of whether the exception was due to a side-effect access or not, as follows:

Bit 54	Meaning
0	Recognized exception is not due to a side-effect access.
1	Recognized exception is due to a side-effect access.

A side-effect access is an implied access not directly associated with a storage operand of an instruction, is not an instruction fetch, is not a fetch of table information during ART or DAT, and is not a store of a trace entry. When the side-effect-access facility is not installed, bit 54 is unpredictable.

The location of a side-effect access is not specified by any operand of an instruction for which the exception is recognized. Instead, the location was established by previous loading of the relevant controls.

MOVE PAGE Indication: When EDAT-1 does not apply, and the exception was a page-translation exception that was recognized during the execution of MOVE PAGE, bit 61 of locations 168-175 is set to one. If the exception was a page-translation exception recognized during the execution of an instruction other than MOVE PAGE, bit 61 is set to zero. If the exception was an ASCE-type, region-first-translation, region-second-translation, region-third-translation, or segment-translation exception, bit 61 of locations 168-175 is unpredictable. See the definition of real location 162 for related information.

When EDAT-1 applies, and the exception was a region-first-, region-second, region-third-, segment-, or page-translation exception that was recognized during the execution of MOVE PAGE, bit 61 of locations 168-175 is set to one. If the exception was a region-first-, region-second, region-third-, segment-, or page-translation exception recognized during the execution of an instruction other than MOVE PAGE, bit 61 is set to zero. If the exception was an ASCE-type exception, bit 61 of locations 168-175 is unpredictable. See the definition of real location 162 for related information.

ASCE Identifier: Bits 62 and 63 of locations 168-175 are set to identify the address-space-control element (ASCE) used in the translation, as follows:

Bit 62	Bit 63	Meaning
0	0	Primary ASCE was used.
0	1	CPU was in the access-register mode, and either the access was an instruction fetch or it was a storage-operand reference that used an AR-specified ASCE (the access was not an implicit reference to the linkage stack). The exception access id, real location 160, can be examined to determine the ASCE used. However, if the primary, secondary, or home ASCE was used, bits 62 and 63 may be set to 00, 10, or 11, respectively, instead of 01.
1	0	Secondary ASCE was used.
1	1	Home ASCE was used (includes the case of an implicit reference to the linkage stack).

The CPU may avoid setting bits 62 and 63 to 01 by recognizing that the access was an instruction fetch, that access-list-entry token 00000000 or 00000001 hex was used, or that the access-list-entry token designated, through an access-list entry, an ASN-second-table entry containing an ASCE equal to the primary ASCE, secondary ASCE, or home ASCE.

Bits 55 and 57-59 are unpredictable.

Translation-Exception Identification for Protection Exceptions: The information stored in the TEID for a protection exception depends on which of the three suppression-on-protection facilities is installed, as described below.

Basic Suppression-on-Protection: When TEID bit 61 is zero, the remainder of the TEID is unpredictable. When TEID bit 61 is one, the following applies:

- TEID bit positions 0-51 contain the effective address that caused the protection exception.
- TEID bits 52-59 are unpredictable.
- If DAT was on, the following applies:
  - The effective address that caused the protection exception is a virtual address.
  - When TEID bit 60 is zero, access-list-controlled protection is not the cause of

the exception. When TEID bit 60 is one, access-list-controlled protection is the cause of the exception.

– TEID bit positions 62-63 contain information identifying ASCE used in the translation, as described in "ASCE Identifier: Bits 62 and 63 of locations 168-175 are set to identify the address-space-control element (ASCE) used in the translation, as follows:" on page 3-77.

If DAT was off, TEID bits 60, 62, and 63 are unpredictable.

Enhanced Suppression-on-Protection Facility 1: When TEID bit 61 is zero, the exception was due to either key-controlled protection or low-address protection, and the remainder of the TEID is unpredictable. When TEID bit position 61 is one, the following applies:

- The exception is due to either access-list-controlled protection or DAT protection, as determined by TEID bit 60.
- TEID bit positions 0-51 contain the virtual address that caused the protection exception.
- When the access-exception-fetch/store-indication facility is installed, TEID bit positions 52-53 contain the access-exception fetch/store indication as described in "Access-Exception Fetch/Store Indication (AEFSI): Bit positions 52-53 of locations 168-175 contain an indication of whether the exception was due to a fetch or a store operation, as follows:" on page 3-76.
- TEID bit 54-59 are unpredictable.
-
- When TEID bit 60 is zero, DAT protection is the cause of the exception. When TEID bit 60 is one, access-list-controlled-protection is the cause of the exception.
- TEID bit positions 62-63 contain information identifying ASCE used in the translation, as described in "ASCE Identifier: Bits 62 and 63 of locations 168-175 are set to identify the address-space-control element (ASCE) used in the translation, as follows:" on page 3-77.

Enhanced Suppression-on-Protection Facility 2: The contents of TEID bit positions 56, 60, and 61 form a three-bit binary code that identifies the cause of the exception. When the protection code is zero, the remainder of the TEID is unpredictable. When the protection code is nonzero, the following applies:

- The three-bit code identifies the cause of the protection exception, as follows:

Code	Meaning
000	Key-controlled or low-address protection
001	DAT protection
010	Key-controlled protection
011	Access-list-controlled protection
100	Low-address protection
101	Instruction-execution-protection (when the instruction-execution-protection facility is installed.

- Except as noted below, TEID bit positions 0-51 contain the effective address that caused the protection exception. For access-list-controlled-protection, DAT-protection, and instruction-execution-protection exceptions, the effective address is virtual. The effective address may be virtual for key-controlled-protection and low-address-protection exceptions.

  For key-controlled-protection or instruction-execution-protection exceptions recognized when fetching an instruction, it is unpredictable whether bit positions 0-51 of the TEID contain the address of the instruction or zeros.
- TEID bits 52-53 are the access-exception fetch/store indication as described in "Access-Exception Fetch/Store Indication (AEFSI): Bit positions 52-53 of locations 168-175 contain an indication of whether the exception was due to a fetch or a store operation, as follows:" on page 3-76.
- TEID bit 54 is the side-effect-access indication, as described in "Side-Effect-Access Indication: When the side-effect-access facility is installed, bit position 54 of locations 168-175 contains an indication of whether the exception was due to a side-effect access or not, as follows:" on page 3-77.
- TEID bit 55 is unpredictable.

- TEID bits 57-59 are unpredictable.
- When DAT is on, TEID bit positions 62-63 contain information identifying ASCE used in the translation, as described in "ASCE Identifier: Bits 62 and 63 of locations 168-175 are set to identify the address-space-control element (ASCE) used in the translation, as follows:" on page 3-77. When DAT is off, or when DAT is on but the effective address is not virtual, TEID bits 62-63 are unpredictable.

Translation-Exception Identification for ASN-Translation Exceptions: During a program interruption due to an AFX-translation, ASX-translation, primary-authority, or secondary-authority exception, the ASN being translated is stored at locations 174 and 175, zeros are stored at locations 172 and 173, and the contents of locations 168-171 remain unchanged.

Translation-Exception Identification for Space-Switch Events: During a program interruption due to a space-switch event, an identification of the old instruction space is stored at locations 174 and 175, the old instruction-space space-switch-event-control bit is placed in bit position 0 and zeros are placed in bit positions 1-15 of locations 172 and 173, and the contents of locations 168-171 remain unchanged. The identification and bit stored are as follows:

- If the CPU was in the primary-space, secondary-space, or access-register mode before the operation, the old PASN, bits 48-63 of control register 4 before the operation, is stored at locations 174 and 175, and the old primary space-switch-event-control bit, bit 57 of control register 1 before the operation, is placed in bit position 0 of locations 172 and 173.
- If the CPU was in the home-space mode before the operation, zeros are stored at locations 174 and 175, and the home space-switch-event-control bit, bit 57 of control register 13, is placed in bit position 0 of locations 172 and 173.

Translation-Exception Identification During PC-Number Translation: During a program interruption due to an EX-translation or LX-translation exception recognized by PROGRAM CALL when ASN-and-LX reuse is not installed or is not enabled by a one value of the ASN-and-LX-reuse control in control register 0, bits 44-63 of the second-operand address used by PROGRAM CALL (a 20-bit PC number), with 12 zeros appended on the left, are stored at locations 172-175, and the contents of locations 168-171 remain unchanged.

During a program interruption due to an EX-translation, LFX-translation, LSTE-sequence, or LSX-translation exception recognized by PROGRAM CALL when ASN-and-LX reuse is enabled and bit 44 of the second-operand address used by PROGRAM CALL is zero, bits 44-63 of the second-operand address (a 20-bit PC number), with 12 zeros appended on the left, are stored at locations 172-175. If bit 44 of the second-operand address is one, bits 32-63 of the address (a 32-bit PC number) are stored at locations 172-175. In either of these cases, the contents of locations 168-171 remain unchanged.

176-183 (B0-B7 hex). . . . . .Real Address

Monitor Code: During a program interruption due to a monitor event, the monitor code is stored at locations 176-183.

184-187 (B8-BB hex) . . . . . .Real Address

Subsystem-Identification Word: During an I/O interruption, the subsystem-identification word is stored at locations 184-187.

188-191 (BC-BF hex) . . . . . .Real Address

I/O-Interruption Parameter: During an I/O interruption, the interruption parameter from the associated subchannel is stored at locations 188-191.

192-195 (C0-C3 hex) . . . . . .Real Address

I/O-Interruption-Identification Word: During an I/O interruption, the I/O-interruption-identification word, which further identifies the source of the I/O interruption, is stored at locations 192-195.

200-203 (C8-CB hex) . . . . . .Real Address

STFL Facility List: The STORE FACILITY LIST instruction stores information at real locations 200-203. The information describes which facilities are provided by the configuration. The information stored is identical in format to the first 32 bits stored by the STORE FACILITY LIST EXTENDED instruction. Figure 4-36, "Assigned

Facility Bits" on page 4-98 shows the meanings of the assigned facility bits.

232-239 (E8-EF hex) . . . . . Real Address

Machine-Check-Interruption Code: During a machine-check interruption, the machine-check-interruption code is stored at locations 232-239.

244-247 (F4-F7 hex) . . . . . Real Address

External-Damage Code: During a machine-check interruption due to certain external-damage conditions, depending on the model, an external-damage code may be stored at locations 244-247.

248-255 (F8-FF hex) . . . . . Real Address

Failing-Storage Address: During a machine-check interruption, a 64-bit failing-storage address may be stored at locations 248-255.

256-263 (100-107 hex). . . . . Real Address

Enhanced-Monitor Counter-Array Origin: Bits 0-60 of the doubleword at location 256, appended on the right with three binary zeros, form the 64-bit virtual address of the enhanced-monitor counter array in the home address space.

During the execution of the MONITOR CALL instruction when the enhanced-monitor facility is installed, both the monitor-mask bit and enhanced-monitor-mask bit in control register 8 corresponding to the monitor class are ones, and the monitor code derived from the first-operand address is less than the enhanced-monitor counter-array size in location 264, then the enhanced-monitor counter designated by the first-operand location is incremented by one.

Programming Note: It is recommended that the enhanced-monitor counter array be allocated on a cache-line boundary of the CPU's first-level data cache. The cache line size may be determined by the EXTRACT CPU ATTRIBUTES (ECAG) instruction.

264-267 (108-10B hex) . . . . . Real Address

Enhanced-Monitor Counter-Array Size: The word at location 264 contains a 32-bit unsigned binary value that is referenced during the execution of the MONITOR CALL instruction when the enhanced-monitor facility is installed, and both the monitor-mask bit and enhanced-monitor-mask bit in control register 8 corresponding to the monitor class are both ones. In this case, if the monitor code derived from the first-operand address is less than the enhanced-monitor counter-array size in location 264, then the counter designated by the first-operand location is incremented by one; otherwise, the enhanced-monitor exception counter at location 268 is incremented.

268-271 (10C-10F hex) . . . . . Real Address

Enhanced-Monitor Exception Count: The word at location 268 contains a 32-bit unsigned binary value that may be updated during the execution of the MONITOR CALL instruction when the enhanced-monitor facility is installed, and the monitor-class bit and corresponding enhanced-monitor-class bit in control register 8 are both ones. The word is updated in either of the following two cases:

- The monitor code derived from the first-operand address is greater than or equal to than the enhanced-monitor counter-array size in location 264.
- The counter designated by the first-operand location is inaccessible.

272-279 (110-117 hex) . . . . . Real Address

Breaking-Event Address: If the PER-3 facility is installed, then, during a program interruption, the contents of the breaking-event-address register are stored in locations 272-279. If the breaking-event-address-recording facility is not installed, this location remains unchanged.

288-303 (120-12F hex) . . . . . Real Address

Restart Old PSW: The current PSW is stored as the old PSW at locations 288-303 during a restart interruption.

304-319 (130-13F hex) . . . . . Real Address

External Old PSW: The current PSW is stored as the old PSW at locations 304-319 during an external interruption.

320-335 (140-14F hex) . . . . . Real Address

Supervisor-Call Old PSW: The current PSW is stored as the old PSW at locations 320-335 during a supervisor-call interruption.

336-351 (150-15F hex) . . . . . Real Address

Program Old PSW: The current PSW is stored as the old PSW at locations 336-351 during a program interruption.

352-367 (160-16F hex) . . . . . Real Address

Machine-Check Old PSW: The current PSW is stored as the old PSW at locations 352-367 during a machine-check interruption.

368-383 (170-17F hex) . . . . . Real Address

Input/Output Old PSW: The current PSW is stored as the old PSW at locations 368-383 during an I/O interruption.

416-431 (1A0-1AF hex) . . . . . Real Address

Restart New PSW: The new PSW is fetched from locations 416-431 during a restart interruption.

432-447 (1B0-1BF hex) . . . . . Real Address

External New PSW: The new PSW is fetched from locations 432-447 during an external interruption.

448-463 (1C0-1CF hex). . . . . Real Address

Supervisor-Call New PSW: The new PSW is fetched from locations 448-463 during a supervisor-call interruption.

464-479 (1D0-1DF hex). . . . . Real Address

Program New PSW: The new PSW is fetched from locations 464-479 during a program interruption.

480-495 (1E0-1EF hex) . . . . . Real Address

Machine-Check New PSW: The new PSW is fetched from locations 480-495 during a machine-check interruption.

496-511 (1F0-1FF hex) . . . . . Real Address

Input/Output New PSW: The new PSW is fetched from locations 496-511 during an I/O interruption.

4528-4535 (11B0-11B7 hex) . . . . . Real Address

Machine-Check-Extended-Save-Area Designation (MCESAD): During a machine check interruption, additional information may be stored at the absolute storage location designated by the doubleword at locations 4528-4535. The leftmost bits of the doubleword, called the machine-check-extended-save-area origin (MCESAO), appended on the right with binary zeros, forms the address of the area.

When the guarded-storage facility is not installed, bit positions 0-53 of the doubleword form the MCESAO. The MCESAO, with 10 zeros appended on the right, is used as the absolute address of a 1,024-byte extended-save area. Bit positions 54-63 of the doubleword are reserved and should contain zeros.

When the guarded-storage facility is installed, bit positions 60-63 of the doubleword contain a length characteristic (LC) that specifies the size and alignment of the extended-save area as a power of two; an LC value of zero is treated as 10. Bits 0 through 63-LC of the doubleword form the MCESAO. The MCESAO, with LC zeros appended on the right, is used as the absolute address of a 2^{LC}-byte extended-save area on a 2^{LC}-byte boundary. Bit positions 64-LC through 59 of the doubleword are reserved and should contain zeros. Figure 3-18 shows the length-characteristic values and corresponding extended-save-area attributes. If a reserved LC value is specified, a machine-check extended save area is not stored.

LC	MCESA Attributes	
	MCESAO	Alignment & Size
0*	Bits 0-53	1,024 bytes
1-9†	N/A	0 bytes
10	Bits 0-53	1,024 bytes
11	Bits 0-52	2,048 bytes
12	Bits 0-51	4,096 bytes
13-15†	N/A	0 bytes

Explanation:

* An LC value of zero is treated as if a value of 10 was specified.
† LC values 1-9 and 13-15 are reserved.
N/A Not applicable; for reserved values, the MCESAO is treated as if all bits are zeros.

Figure 3-18. Machine-Check Extended-Save-Area Attributes based on Length Characteristic

When the value of the MCESAO is zero, storing is not performed in the extended save area.

The format of the machine-check extended save area is described in the section “Machine-Check Extended Save Area (MCESA)” on page 11-24.

4544-4607 (11C0-11FF hex) . . . . . Real Address

Available for Programming: Locations 4544-4607 are available for use by programming.

4608-4735 (1200-127F hex) . . . . . Absolute Address

Store-Status Floating-Point-Register Save Area: During the execution of the store-status operation, the contents of the floating-point registers are stored at locations 4608-4735.

4608-4735 (1200-127F hex) . . . . . Real Address

Machine-Check Floating-Point-Register Save Area: During a machine-check interruption, the contents of the floating-point registers are stored at locations 4608-4735.

4736-4863 (1280-12FF hex) . . . . . Absolute Address

Store-Status General-Register Save Area: During the execution of the store-status operation, the contents of the general registers are stored at locations 4736-4863.

4736-4863 (1280-12FF hex) . . . . . Real Address

Machine-Check General-Register Save Area: During a machine-check interruption, the contents of the general registers are stored at locations 4736-4863.

4864-4879 (1300-130F hex) . . . . . Absolute Address

Store-Status PSW Save Area: During the execution of the store-status operation, the contents of the current PSW are stored at locations 4864-4879.

4864-4879 (1300-130F hex) . . . . . Real Address

Fixed-Logout Area: Depending on the model, logout information may be stored at locations 4864-4879 during a machine-check interruption.

4880-4887 (1310-1317 hex). . . . . Real Address

Assigned to IBM internal use.

4888-4891 (1318-131B hex) . . . . . Absolute Address

Store-Status Prefix Save Area: During the execution of the store-status operation, the contents of the prefix register are stored at locations 4888-4891.

4892-4895 (131C-131F hex) . . . . . Absolute Address

Store-Status Floating-Point-Control-Register Save Area: During the execution of the store-status operation, the contents of the floating-point control register are stored at locations 4892-4895.

4892-4895 (131C-131F hex) . . . . . Real Address

Machine-Check Floating-Point-Control-Register Save Area: During a machine-check interruption, the contents of the floating-point control register are stored at locations 4892-4895.

4900-4903 (1324-1327 hex) . . . . . Absolute Address

Store-Status TOD-Programmable-Register Save Area: During the execution of the store-status operation, the contents of the TOD programmable register are stored at locations 4900-4903.

4900-4903 (1324-1327 hex) . . . . . Real Address

Machine-Check TOD-Programmable-Register Save Area: During a machine-check interruption, the contents of the TOD programmable register are stored at locations 4900-4903.

4904-4911 (1328-132F hex) . . . . . Absolute Address

Store-Status CPU-Timer Save Area: During the execution of the store-status operation, the contents of the CPU timer are stored at locations 4904-4911.

4904-4911 (1328-132F hex) . . . . . Real Address

Machine-Check CPU-Timer Save Area: During a machine-check interruption, the contents of the CPU timer are stored at locations 4904-4911.

4913-4919 (1331-1337 hex) . . . . . Absolute Address

Store-Status Clock-Comparator Save Area: During the execution of the store-status operation, the contents of bit positions 0-55 of the clock comparator are stored at locations 4913-4919. When this store occurs, zeros are stored at location 4912.

4913-4919 (1331-1337 hex) . . . . . Real Address

Machine-Check Clock-Comparator Save Area: During a machine-check interruption, the contents of bit positions 0-55 of the clock comparator are stored at locations 4913-4919. When this store occurs, zeros are stored at location 4912.

4928-4991 (1340-137F hex) . . . . . Absolute Address

Store-Status Access-Register Save Area: During the execution of the store-status operation, the contents of the access registers are stored at locations 4928-4991.

4928-4991 (1340-137F hex) . . . . . Real Address

Machine-Check Access-Register Save Area: During a machine-check interruption, the contents of the access registers are stored at locations 4928-4991.

4992-5119 (1380-13FF hex) . . . . . Absolute Address

Store-Status Control-Register Save Area: During the execution of the store-status operation, the contents of the control registers are stored at locations 4992-5119.

4992-5119 (1380-13FF hex) . . . . . Real Address

Machine-Check Control-Register Save Area: During a machine-check interruption, the contents of the control registers are stored at locations 4992-5119.

6144-6399 (1800-18FF hex) . . . . . Real Address

Program-Interruption Transaction Diagnostic Block: During a program interruption while the CPU is in the transactional-execution mode, the transaction-diagnostic block is stored at locations 6,144-6,399. The format of the transaction-diagnostic block is described in "Transaction Diagnostic Block (TDB)" on page 5-93.

Assigned Storage Locations in the ESA/390-Compatibility Mode

0-7 . . . . . Absolute Address

IPL PSW: The first eight bytes read during a CCW-type initial-program-loading (IPL) initial-read operation are stored at locations 0-7. The list-directed IPL process also stores eight bytes at locations 0-7. The contents of these locations are used to form the new PSW at the completion of the IPL operation. These locations may also be used for temporary storage at the initiation of the IPL operation.

The IPL PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

0-7 . . . . . Real Address

Restart New PSW: The new PSW is fetched from locations 0-7 during a restart interruption. The restart new PSW has the short-PSW format as shown in Figure 4-3 on page 4-8.

8-15 (08-0F hex) . . . . . Absolute Address

CCW-Type IPL CCW1: Bytes 8-15 read during a CCW-type initial-program-loading (IPL) initial-read operation are stored at locations 8-15. The contents of these locations are ordinarily used as the next CCW in an IPL CCW chain after completion of the IPL initial-read operation.

8-15 (08-0F hex) . . . . . Real Address

Restart Old PSW: The current PSW is stored as the old PSW at locations 288-303 during a restart interruption. The restart old PSW has the short-PSW format as shown in Figure 4-3 on page 4-8.

16-23 (10-17 hex) . . . . . Absolute Address

CCW-Type IPL CCW2: Bytes 16-23 read during a CCW-type initial-program loading (IPL) initial-read operation are stored at locations 16-23. The contents of these locations may be used as another CCW in the IPL CCW chain to follow IPL CCW1.

LD IPL Machine-Loader Execution-Space Size: During a list-directed initial-program loading (IPL) operation, the machine-loader execution-space size is stored at locations 16-19.

Available for Use by Programming: Locations 16-19 are available for use by programming after IPL.

LD IPL System-Parameter-Block Address: During a list-directed initial-program loading (IPL) operation, the absolute address of the first byte of the system-parameter block is stored at locations 20-23.

24-31 (18-1F hex) . . . . . Real Address

External Old PSW: The current PSW is stored as the old PSW at locations 24-31 during an external interruption. The external old PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

32-39 (20-27 hex). . . . . Real Address

Supervisor-Call Old PSW: The current PSW is stored as the old PSW at locations 32-39 during a supervisor-call interruption. The supervisor-call old PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

40-47 (28-2F hex) . . . . . Real Address

Program Old PSW: The current PSW is stored as the old PSW at locations 40-47 during a program interruption. The program old PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

48-55 (30-37 hex). . . . . Real Address

Machine-Check Old PSW: The current PSW is stored as the old PSW at locations 48-55 during a machine-check interruption. The machine-check old PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

56-63 (38-3F hex) . . . . . Real Address

Input/Output Old PSW: The current PSW is stored as the old PSW at locations 56-63 during an input/output interruption. The input/output old PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

88-95 (58-5F hex) . . . . . Real Address

External New PSW: The new PSW is fetched from locations 88-95 during an external interruption. The external new PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

96-103 (60-67 hex). . . . . Real Address

Supervisor-Call New PSW: The new PSW is fetched from locations 96-103 during a supervisor-call interruption. The supervisor-call new PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

104-111 (68-6F hex) . . . . . Real Address

Program New PSW: The new PSW is fetched from locations 104-111 during a program interruption. The program new PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

112-119 (70-77 hex). . . . . Real Address

Machine-Check New PSW: The new PSW is fetched from locations 112-119 during a machine-check interruption. The machine-check new PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

120-127 (78-7F hex) . . . . . Real Address

Input/Output New PSW: The new PSW is fetched from locations 120-127 during an input/output interruption. The input/output new PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

128-131 (80-83 hex) . . . . . Real Address

External-Interruption Parameter: During an external interruption due to service signal or timing alert, the parameter associated with the interruption is stored at locations 128-131.

132-133 (84-85 hex) . . . . . Real Address

CPU Address: During an external interruption due to malfunction alert, emergency signal, or external call, the CPU address associated with the source of the interruption is stored at locations 132-133. For all other external-interruption conditions, zeros are stored at locations 132-133.

134-135 (86-87 hex) . . . . . Real Address

External-Interruption Code: During an external interruption, the interruption code is stored at locations 134-135.

136-139 (88-8B hex) . . . . . Real Address

Supervisor-Call-Interruption Identification: During a supervisor-call interruption, the instruction-length code is stored in bit positions 5 and 6 of location 137, and the interruption code is stored at locations 138-139. Zeros are stored at location 136 and in the remaining bit positions of location 137.

140-143 (8C-8F hex) . . . . . Real Address

Program-Interruption Identification: During a program interruption, the instruction-length code is stored in bit positions 5 and 6 of location 141, and the interruption code is stored at locations 142-143. Zeros are stored at location 140 and in the remaining bit positions of location 141.

144-147 (90-93 hex) . . . . . Real Address

Data-Exception Code (DXC): During a program interruption due to a data exception, the data-

exception code is stored at location 147, and zeros are stored at locations 144-146. The DXC is described in "Data-Exception Code (DXC)" on page 6-17.

Protection-Exception Identification (PEID): Information may stored at real locations 144-147 during a program interruption due to a protection exception. The information has the same format as bits 32-63 of the TEID stored at real locations 172-175 in z/Architecture architectural mode. See "Translation-Exception Identification (TEID)" on page 3-76 for details. See Reference [12.] on page xxx for further details on locations 144-147 in ESA/XC virtual machines.

148-149 (94-95 hex) . . . . . Real Address

Monitor-Class Number: During a program interruption due to a monitor event, the monitor-class number is stored at location 149, and zeros are stored at location 148.

150-151 (96-97 hex) . . . . . Real Address

PER Code: During a program interruption due to a PER event the PER code is stored in bit positions 0-7 of locations 150-151, and other information is or may be stored as described in "Identification of Cause" on page 4-29.

152-155 (98-9B hex) . . . . . Real Address

PER Address: During a program interruption due to a PER event, the PER address is stored at locations 152-155.

156-159 (9C-9F hex) . . . . . Real Address

Monitor Code: During a program interruption due to a monitor event, the monitor code is stored at locations 156-159.

160 (A0 hex) . . . . . Real Address

Exception Access Identification (EAID): During a program interruption due to a protection exception, information is stored at location 160 as described in "Suppression on Protection" on page 3-15.

In ESA/XC virtual machines under z/VM, during a program interruption doe to an ALEN-translation or addressing-capability exception, if the ALET being translated was obtained from an access register, the number of the access register used is stored in bit positions 4-7 of location 160, and zeros are stored in bit positions 0-3. If the ALET being translated was not obtained from an access register, then zeros are stored at location 160. See Reference [12.] on page xxx for details.

161 (A1 hex) . . . . . .Real Address

PER Access Identification: During a program interruption due to a PER storage-alteration event or PER zero-address-detection event, the contents of location 161 are unpredictable.

163 (A3 hex) . . . . . Absolute Address

Store-Status Architectural-Mode Identification: During the execution of the store-status operation, zeros are stored in location 163.

163 (A3 hex) . . . . . .Real Address

Machine-Check Architectural-Mode Identification: During a machine-check interruption, zeros are stored in location 163.

168-171 (A8-AB hex) . . . . . .Real Address

Exception ALET: In ESA/XC virtual machines under z/VM, during a program interruption due to an ALEN-translation or addressing-capability exception, the access-list-entry token used in the translation is stored at locations 168-171. See Reference [12.] on page xxx for details.

184-187 (B8-BB hex) . . . . . .Real Address

Subsystem-Identification Word: During an I/O interruption, the subsystem-identification word is stored at locations 184-187.

188-191 (BC-BF hex) . . . . . .Real Address

I/O-Interruption Parameter: During an I/O interruption, the interruption parameter from the associated subchannel is stored at locations 188-191.

192-195 (C0-C3 hex) . . . . . .Real Address

I/O-Interruption-Identification Word: During an I/O interruption, the I/O-interruption-identification word, which further identifies the source of the I/O interruption, is stored at locations 192-195.

200-203 (C8-CB hex) . . . . . .Real Address

STFL Facility List: The STORE FACILITY LIST instruction stores information at real locations

200-203. The information describes which facilities are provided by the configuration. The information stored is identical in format to the first 32 bits stored by the STORE FACILITY LIST EXTENDED instruction. Figure 4-36, “Assigned Facility Bits” on page 4-98 shows the meanings of the assigned facility bits.

212-215 (D4-D7 hex) . . . . . .Absolute Address

Store-Status Extended-Save-Area Address: During the execution of the store-status operation when the extended-save-area control, bit 34 of control register 14, is one, bits 1-19 of locations 212-215, with 33 zeros appended on the left and 12 zeros appended on the right, are used as the absolute address of a 4,096-byte extended save area. Bits 0 and 20-31 of the locations are reserved and should be zeros. They are ignored when forming the address of the extended save area. When bits 1-19 are all zeros, storing is not performed in the extended save area.

212-215 (D4-D7 hex) . . . . . Real Address

Machine-Check Extended-Save-Area Address: During a machine-check interruption when the extended-save-area control, bit 34 of control register 14, is one, bits 1-19 of locations 212-215, with 33 zeros appended on the left and 12 zeros appended on the right, are used as the absolute address of a 4,096-byte extended save area. Bits 0 and 20-31 of the locations are reserved and should be zeros. They are ignored when forming the address of the extended save area. When bits 1-19 are all zeros, storing is not performed in the extended save area.

216-223 (D8-DF hex) . . . . . .Absolute Address

Store-Status CPU-Timer Save Area: During the execution of the store-status operation, the contents of the CPU timer are stored at locations 216-223.

216-223 (D8-DF hex) . . . . . Real Address

Machine-Check CPU-Timer Save Area: During a machine-check interruption, the contents of the CPU timer are stored at locations 216-223.

224-231 (E0-E7 hex) . . . . . .Absolute Address

Store-Status Clock-Comparator Save Area: During the execution of the store-status operation, the contents of the clock comparator are stored at locations 224-231.

224-231 (E0-E7 hex) . . . . . Real Address

Machine-Check Clock-Comparator Save Area: During a machine-check interruption, the contents of the clock comparator are stored at locations 224-231.

232-239 (E8-EF hex) . . . . . Real Address

Machine-Check-Interruption Code: During a machine-check interruption, the machine-check-interruption code is stored at locations 232-239.

244-247 (F4-F7 hex) . . . . . Real Address

External-Damage Code: During a machine-check interruption due to certain external-damage conditions, depending on the model, an external-damage code may be stored at locations 244-247.

248-251 (F8-FB hex) . . . . . Real Address

Failing-Storage Address: During a machine-check interruption, a 31-bit failing-storage address may be stored at locations 248-251.

256-263 (100-107 hex) . . . . . Absolute Address

Store-Status PSW Save Area: During the execution of the store-status operation, the contents of the current PSW are stored at locations 256-263. The PSW has the short-PSW format, as shown in Figure 4-3 on page 4-8.

256-271 (100-10F hex) . . . . . Real Address

Fixed-Logout Area: During a machine-check interruption, logout information may be stored at locations 256-271.

In ESA/XC virtual machines, a failing-storage address-space-identification token (ASIT) may be stored in locations 256-263 during a machine-check interruption. A failing-storage ASIT is stored whenever a failing-storage address is stored at locations 248-251. See Reference [12.] on page xxx for details.

264-267 (108-10B hex) . . . . . Absolute Address

Store-Status Prefix Save Area: During the execution of the store-status operation, the contents of

the prefix register are stored at locations 264-267.

288-351 (120-15F hex) . . . . . Absolute Address

Store-Status Access-Register Save Area: During the execution of the store-status operation, the contents of the access registers are stored at locations 288-351.

288-351 (120-15F hex) . . . . . Real Address

Machine-Check Access-Register Save Area: During a machine-check interruption, the contents of the access registers are stored at locations 288-351.

352-367 (160-17F hex) . . . . . Absolute Address

Store-Status Floating-Point-Register Save Area: During the execution of the store-status operation, the contents of floating-point registers 0, 2, 4, and 6 are stored at locations 352-367.

352-367 (160-17F hex) . . . . . Real Address

Machine-Check Floating-Point-Register Save Area: During a machine-check interruption, the contents of floating-point registers 0, 2, 4, and 6 are stored at locations 352-367.

384-447 (180-1BF hex) . . . . . Absolute Address

Store-Status General-Register Save Area: During the execution of the store-status operation, the contents of the general registers are stored at locations 384-447.

384-447 (180-1BF hex) . . . . . Real Address

Machine-Check General-Register Save Area: During a machine-check interruption, the contents of the general registers are stored at locations 384-447.

448-511 (1C0-1FF hex) . . . . . Absolute Address

Store-Status Control-Register Save Area: During the execution of the store-status operation, the contents of the control registers are stored at locations 448-511.

448-511 (1C0-1FF hex) . . . . . .Real Address

Machine-Check Control-Register Save Area: During a machine-check interruption, the contents of the control registers are stored at locations 448-511.

Programming Notes:

1. When the CPU is in the access-register mode, some instructions, such as MVCL, which address operands in more than one address space, may cause a storage-alteration PER event in one address space concurrently with a region-translation, segment-translation, or page-translation exception in another address space. The access registers used to cause these conditions in such a case are different. In order to identify both access registers, two access identifications, namely the exception access identification and the PER access identification, are provided.
2. The store-status and machine-check architectural-mode identifications at absolute and real locations 163, respectively, indicate that the CPU is in the z/Architecture architectural mode. When z/Architecture is installed on the CPU but the CPU is in the ESA/390 or ESA/390-compatibility mode, the store-status and machine-check-interruption operations store zero at location 163.
3. Figure 3-19 on page 3-87 illustrates assigned storage locations 0-511 in both the z/Architecture architectural mode and the ESA/390-compatibility mode. Figure 3-20 on page 3-90 illustrates assigned storage locations 512-8191 in the z/Architecture architectural mode. Locations marked [A] are absolute; all other locations are real.

Hex	Dec	z/Architecture Fields	ESA/390-Compatibility-Mode Fields
0 4	0 4	Initial-Program-Loading PSW (when CZAM installed) [A]	Initial-Program-Loading PSW [A]; Restart New PSW
8 C	8 12	CCW-Type IPL CCW1 (when CZAM installed) [A]	CCW-Type IPL CCW1 [A]; Restart Old PSW

Figure 3-19. Assigned Storage Locations 0–511 in the z/Architecture Architectural Mode and ESA/390-Compatibility Mode (Part 1 of 4)

Hex	Dec	z/Architecture Fields	ESA/390-Compatibility-Mode Fields
10	16	CCW-Type IPL CCW2, bytes 0-3 (when CZAM installed) [A]; Machine-loader execution-space size [A]; Available for use by programming after IPL	CCW-Type IPL CCW2, bytes 0-3 [A]; Machine-loader execution-space size [A]; Available for use by programming after IPL
14	20	CCW-Type IPL CCW2, bytes 4-7 (when CZAM installed) [A]; LD-IPL System-IPL Parameter-List Pointer [A]	CCW-Type IPL CCW2, bytes 4-7 [A]; LD-IPD System-IPL Parameter-List Pointer [A]
18	24		External Old PSW
1C	28		
20	32		Supervisor-Call Old PSW
24	36		
28	40		Program Old PSW
2C	44		
30	48		Machine-Check Old PSW
34	52		
38	56		Input / Output Old PSW
3C	60		
40	64		
44	68		
48	72	Available for Use by Programming	
4C	76		
50	80		
54	84		
58	88		External New PSW
5C	92		
60	96		Supervisor-Call New PSW
64	100		
68	104		Program New PSW
6C	108		
70	112		Machine-Check New PSW
74	116		
78	120		Input / Output New PSW
7C	124		
80	128	External-Interruption Parameter	External-Interruption Parameter
84	132	CPU Address | External-Interruption Code	CPU Address | External-Interruption Code
88	136	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | ILC | 0 | SVC-Interruption Code	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | ILC | 0 | SVC-Interruption Code
8C	140	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | ILC | 0 | Program-Interruption Code	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | ILC | 0 | Program-Interruption Code
90	144	Data-Exception Code or Vector-Exception Code	Data-Exception Code or Translation-Exception Identification
94	148	Monitor-Class Number | Per Code | ATMID | AI	Monitor-Class Number | Per Code | ATMID | SI
98	152	PER Address	PER Address
9C	156		Monitor Code
A0	160	Exception Access ID | PER Access ID | Operand Access ID | SS Arch. Mode ID [A]; MC Arch. Mode ID	Exception Access ID | PER Access ID | | SS Arch. Mode ID [A]; MC Arch. Mode ID
A4	164		
A8	168	Translation-Exception Identification	
AC	172		
B0	176	Monitor Code	
B4	180		

Figure 3-19. Assigned Storage Locations 0–511 in the z/Architecture Architectural Mode and ESA/390-Compatibility Mode (Part 2 of 4)

Hex	Dec	z/Architecture Fields	ESA/390-Compatibility-Mode Fields
B8	184	Subsystem Identification Word	Subsystem Identification Word
BC	188	I/O Interruption Parameter	I/O Interruption Parameter
C0	192	I/O Interruption-Identification Word	I/O Interruption-Identification Word
C4	196		
C8	200	STFL Facility List	STFL Facility List
CC	204	Available for Use by Programming	
D0	208		
D4	212		Store-Status Extended-Save-Area Address [A]; Machine-Check Extended-Save-Area Address;
D8	216		Store-Status CPU-Timer Save Area [A]; Machine-Check CPU-Timer Save Area
DC	220		
E0	224		Store-Status Clock-Comparator Save Area [A]; Machine-Check Clock-Comparator Save Area
E4	228		
E8	232	Machine-Check Interruption Code	Machine-Check Interruption Code
EC	236		
F0	240		
F4	244	External-Damage Code	External-Damage Code
F8	248	Failing-Storage Address	Failing-Storage Address
FC	252		
100	256	Enhanced-Monitor Counter-Array Origin	Store-Status PSW Save Area [A]; Machine-Check Fixed-Logout Area (Part 1)
104	260		
108	264	Enhanced-Monitor Counter-Array Size	Store-Status Prefix Save Area [A]; Machine-Check Fixed-Logout Area (Part 2)
10C	268	Enhanced-Monitor Exception Count	Machine-Check Fixed-Logout Area (Part 3)
110	272	Breaking-Event Address	
114	276		
118	280		
11C	284		
120	288	Restart Old PSW	Store Status Access-Register Save Area [A]; Machine-Check Access-Register Save Area (64 bytes)
124	292		
128	296		
12C	300		
130	304	External Old PSW	
134	308		
138	312		
13C	316		
140	320	Supervisor-Call Old PSW	
144	324		
148	328		
14C	332		
150	336	Program Old PSW	
154	340		
158	344		
15C	348		

Figure 3-19. Assigned Storage Locations 0–511 in the z/Architecture Architectural Mode and ESA/390-Compatibility Mode (Part 3 of 4)

Hex	Dec	z/Architecture Fields	ESA/390-Compatibility-Mode Fields
160	352	Machine-Check Old PSW	Store Status Floating-Point-Register 0, 2, 4, & 6 Save Area [A]; Machine-Check Floating-Point-Register 0, 2, 4, & 6 Save Area (32 bytes)
164	356		
168	360		
16C	364		
170	368	Input/Output Old PSW	
174	372		
178	376		
17C	380		
180	384		Store Status General-Register Save Area [A]; Machine-Check General-Register Save Area (64 bytes)
184	388		
188	392		
18C	396		
190	400		
194	404		
198	408		
19C	412		
1A0	416	Restart New PSW	
1A4	420		
1A8	424		
1AC	428		
1B0	432	External New PSW	
1B4	436		
1B8	440		
1BC	444		
1C0	448	Supervisor-Call New PSW	Store Status Control-Register Save Area [A]; Machine-Check Control-Register Save Area (64 bytes)
1C4	452		
1C8	456		
1CC	460		
1D0	464	Program New PSW	
1D4	468		
1D8	472		
1DC	476		
1E0	480	Machine-Check New PSW	
1E4	484		
1E8	488		
1EC	492		
1F0	496	Input/Output New PSW	
1F4	500		
1F8	504		
1FC	508		

Figure 3-19. Assigned Storage Locations 0–511 in the z/Architecture Architectural Mode and ESA/390-Compatibility Mode (Part 4 of 4)

Hex	Dec	z/Architecture Fields
200	512	
204	516	

Figure 3-20. Assigned Storage Locations 512–8191 in the z/Architecture Architectural Mode (Part 1 of 3)

Hex	Dec	z/Architecture Fields
⋮	⋮	
FF8	4088	
FFC	4092	
1000	4096	
⋮	⋮	
11AC	4524	
11B0	4528	Machine-Check Extended-Save-Area Address
11B4	4532	
11B8	4536	
11BC	4540	
11C0	4544	Available for Use by Programming (64 bytes)
11C4	4548	
⋮	⋮	
11F8	4600	
11FC	4604	
1200	4608	Store-Status Floating-Point-Register Save Area [A]; Machine-Check Floating-Point-Register Save Area (128 bytes)
1204	4612	
⋮	⋮	
1278	4728	
127C	4732	
1280	4736	Store-Status General-Register Save Area [A]; Machine-Check General-Register Save Area (128 bytes)
1284	4740	
⋮	⋮	
12F8	4856	
12FC	4860	
1300	4864	Store-Status PSW Save Area [A]; Machine-Check Fixed Logout Area
1304	4868	
1308	4872	
130C	4876	
1310	4880	Assigned to IBM internal use
1314	4884	
1318	4888	Store-Status Prefix Save Area [A]
131C	4892	Store-Status Floating-Point-Control-Register Save Area [A]; Machine-Check Floating-Point-Control-Register Save Area
1320	4896	
1324	4900	Store-Status TOD-Programmable-Register Save Area [A]; Machine-Check TOD-Programmable-Register Save Area
1328	4904	Store-Status CPU-Timer Save Area [A]; Machine-Check CPU-Timer Save Area
132C	4908	
1330	4912	Store-Status Clock-Comp. Bits 0-55 Save Area [A]; Machine-Check Clock-Comparator Bits 0-55 Save Area
1334	4916	
1338	4920	
133C	4924	

Figure 3-20. Assigned Storage Locations 512–8191 in the z/Architecture Architectural Mode (Part 2 of 3)

Hex	Dec	z/Architecture Fields
1340	4928	Store-Status Access-Register Save Area [A]; Machine-Check Access-Register Save Area (64 bytes)
1344	4932	
⋮	⋮	
1378	4984	
137C	4988	
1380	4992	Store-Status Control-Register Save Area [A]; Machine-Check Control-Register Save Area (128 bytes)
1384	4996	
⋮	⋮	
13F8	5112	
13FC	5116	
1400	5120	
1404	5124	
⋮	⋮	
17F8	6136	
17FC	6140	
1800	6144	Program-Interruption Transaction Diagnostic Block (256 bytes)
1804	6148	
⋮	⋮	
18F8	6392	
18FC	6396	
1900	6400	
1904	6404	
⋮	⋮	
1FF8	8184	
1FFC	8188	

Figure 3-20. Assigned Storage Locations 512–8191 in the z/Architecture Architectural Mode (Part 3 of 3)

Chapter 4. Control

CPU States 4-2
Stopped State 4-2
Operating State 4-3
Load State 4-3
Check-Stop State. 4-3
Program-Status Word 4-4
Program-Status-Word Format 4-5
Short PSW Format. 4-7
Control Registers. 4-8
Tracing. 4-12
Implicit Tracing. 4-12
Branch Tracing 4-12
ASN Tracing 4-13
Mode Tracing. 4-13
Explicit Tracing 4-14
Control-Register Allocation 4-15
Trace Entries 4-15
Trace Operation. 4-25
Program-Event Recording. 4-26
PER Instruction-Fetching Nullification 4-26
Control-Register Allocation and Address-Space-Control Element 4-26
PER Operation. 4-28
Identification of Cause 4-29
Priority of Indication 4-33
Storage-Area Designation 4-34
PER Events 4-35
Successful Branching 4-35
Instruction Fetching 4-36
Storage Alteration 4-36
Store Using Real Address 4-37
Zero-Address Detection. 4-37
Transaction End. 4-39
Indication of PER Events Concurrently with Other Interruption Conditions 4-39
Indication of PER Events and Guarded-Storage Events 4-42
Breaking-Event-Address Recording 4-43
Breaking-Event-Address Register 4-44
Execution-Break Instructions. 4-44
Timing 4-46
Time-of-Day Clock and Epoch Index. 4-46
Format 4-46
States 4-47
Changes in Clock State 4-48
Setting and Inspecting the Clock 4-49
TOD Programmable Register 4-51
TOD-Clock Synchronization 4-53
Timing Mode 4-53
Timing State. 4-53
STP Clock Source State 4-54
TOD-Clock Steering 4-54
Offset-Based TOD-Clock Steering Overview 4-55
Hardware-Based TOD-Clock-Steering Overview 4-55
TOD-Offset-Update Events. 4-56
Episodes 4-56
TOD-Clock-Steering Registers 4-56
UTC Information Block (UIB) 4-57
Clock Comparator. 4-59
CPU Timer 4-62
Guarded-Storage Facility 4-64
Guarded-Storage-Facility Registers 4-64
Control Register 2. 4-64
Guarded-Storage-Designation (GSD) Register 4-65
Guarded-Storage-Section-Mask (GSSM) Register. 4-65
Guarded-Storage-Event Parameter-List-Address (GSEPLA) Register. 4-66
Guarded-Storage Control Block (GSCB) 4-66
Guarded-Storage-Event Parameter List (GSEPL) 4-66
Guarded-Storage Facility Operation 4-69
Guarded-Storage-Event Detection 4-69
Guarded-Storage-Event Processing 4-70
Externally Initiated Functions 4-73
Resets. 4-73
. CPU Reset 4-77
Initial CPU Reset. 4-78
Subsystem Reset 4-79
Clear Reset. 4-79
Power-On Reset 4-80
Initial Program Loading 4-81
CCW-Type IPL 4-81
Store Status 4-81
Multiprocessing. 4-82
Shared Main Storage 4-83
CPU-Address Identification. 4-83
CPU-Address Expansion 4-83
CPU-Address Contraction. 4-84
CPU Signaling and Response 4-84
Signal-Processor Orders. 4-84
Sense 4-85
External Call 4-85
Emergency Signal. 4-85
Start. 4-85
Stop. 4-85
Restart. 4-85

© Copyright IBM Corp. 2000, 2017

Stop and Store Status 4-85
Initial CPU Reset 4-85
CPU Reset 4-85
Set Prefix 4-85
Store Status at Address 4-86
Store Extended Status at Address 4-87
Set Architecture 4-89
Conditional Emergency Signal 4-90
Sense Running Status 4-91
Set Multithreading 4-91
Store Additional Status at Address 4-92
Conditions Determining Response 4-94
Conditions Precluding Interpretation of the Order Code 4-94
Status Bits 4-94
Facility Indications 4-98

This chapter describes in detail the facilities for controlling, measuring, and recording the operation of one or more CPUs.

CPU States

The stopped, operating, load, and check-stop states are four mutually exclusive states of the CPU. When the CPU is in the stopped state, instructions and interruptions, other than the restart interruption, are not executed. In the operating state, the CPU executes instructions and takes interruptions, subject to the control of the program-status word (PSW) and control registers, and in the manner specified by the setting of the operator-facility rate control. The CPU is in the load state during the initial-program-loading operation. The CPU enters the check-stop state only as the result of machine malfunctions.

A change between these four CPU states can be effected by use of the operator facilities or by acceptance of certain SIGNAL PROCESSOR orders addressed to that CPU. The states are not controlled or identified by bits in the PSW. The stopped, load, and check-stop states are indicated to the operator by means of the manual indicator, load indicator, and check-stop indicator, respectively. These three indicators are off when the CPU is in the operating state.

The CPU timer is updated when the CPU is in the operating state or the load state. The TOD clock is not affected by the state of any CPU.

Stopped State

The CPU changes from the operating state to the stopped state by means of the stop function. The stop function is performed when:

- The stop key is activated while the CPU is in the operating state.
- The CPU accepts a stop or stop-and-store-status order specified by a SIGNAL PROCESSOR instruction addressed to this CPU while it is in the operating state.
- The CPU has finished the execution of a unit of operation initiated by performing the start function with the rate control set to the instruction-step position.

When the stop function is performed, the transition from the operating to the stopped state occurs at the end of the current unit of operation. When the wait-state bit of the PSW is one, the transition takes place immediately, provided no interruptions are pending for which the CPU is enabled. In the case of interruptible instructions, the amount of data processed in a unit of operation depends on the particular instruction and may depend on the model.

Before entering the stopped state by means of the stop function, all pending allowed interruptions occur while the CPU is still in the operating state. They cause the old PSW to be stored and the new PSW to be fetched before the stopped state is entered. While the CPU is in the stopped state, interruption conditions remain pending.

The CPU is also placed in the stopped state when:

- CPU reset is completed. However, when the reset operation is performed as part of initial program loading for this CPU, then the CPU is placed in the load state and does not necessarily enter the stopped state.
- An address comparison indicates equality and stopping on the match is specified.

The execution of resets is described in “Resets” on page 4-73, and address comparison is described in “Address-Compare Controls” on page 12-1.

If the CPU is in the stopped state when a COMPARE AND REPLACE DAT TABLE ENTRY, INVALIDATE DAT TABLE ENTRY or INVALIDATE PAGE TABLE ENTRY instruction is executed on another CPU in the configuration and the instruction specifies the clearing of all TLBs in the configuration, the clearing of TLB entries is completed before the CPU leaves the stopped state.

Operating State

The CPU changes from the stopped state to the operating state by means of the start function or when a restart interruption (see “Restart Interruption” on page 6-56) occurs.

The start function is performed if the CPU is in the stopped state and (1) the start key associated with that CPU is activated or (2) that CPU accepts the start order specified by a SIGNAL PROCESSOR instruction addressed to that CPU. The effect of performing the start function is unpredictable when the stopped state has been entered by means of a reset.

When the rate control is set to the process position and the start function is performed, the CPU starts operating at normal speed. When the rate control is set to the instruction-step position and the wait-state bit is zero, one instruction or, for interruptible instructions, one unit of operation is executed, and all pending allowed interruptions occur before the CPU returns to the stopped state. When the rate control is set to the instruction-step position and the wait-state bit is one, the start function does not cause an instruction to be executed, but all pending allowed interruptions occur before the CPU returns to the stopped state.

Load State

The CPU enters the load state when the load-normal, load-with-dump, load-clear, or load-clear-list-directed key is activated. (See “Initial Program Loading” on page 4-81 and “Initial Program Loading” on page 17-16.)

When neither the configuration-z/Architecture-architectural-mode (CZAM) facility nor the ESA/390-compatibility-mode facility are installed, this sets the architectural mode to the ESA/390 mode. When the ESA/390-compatibility-mode facility is installed, this sets the architecture mode to the ESA/390-compatibility mode. When the CZAM facility is installed, the architectural mode is unchanged (that is, it remains in the z/Architecture architectural mode).

Programming Note: If a control program that operates in the z/Architecture architectural mode is loaded in a configuration where the CZAM facility may or may not be installed, then the program must be able to tolerate any of the ESA/390, ESA/390-compatibility, or z/Architecture architectural modes when execution begins.

- If the load operation places the configuration into the ESA/390 architectural mode or ESA/390 compatibility mode, then the program must issue the SIGP set-architecture order to switch to the z/Architecture architectural mode.
- If the load operation places the configuration into the z/Architecture mode, then the program must either not issue the SIGP set-architecture order, or be able to tolerate an invalid-parameter response from the SIGP instruction indicating that the configuration is already in the z/Architecture architectural mode.

Check-Stop State

The check-stop state, which the CPU enters on certain types of machine malfunction, is described in “Check-Stop State” on page 11-9. The CPU leaves the check-stop state when CPU reset is performed.

Programming Notes:

1. Except for the relationship between execution time and real time, the execution of a program is not affected by stopping the CPU.
2. When, because of a machine malfunction, the CPU is unable to end the execution of an instruction, the stop function is ineffective, and a reset function has to be invoked instead. A similar situation occurs when an unending string of interruptions results from a PSW with a PSW-format error of the type that is recognized early, or from a persistent interruption condition, such as one due to the CPU timer.
3. Pending I/O operations may be initiated, and active I/O operations continue to suspension or completion, after the CPU enters the stopped state. The interruption conditions due to suspen-

sion or completion of I/O operations remain pending when the CPU is in the stopped state.

Program-Status Word

The current program-status word (PSW) in the CPU contains information required for the execution of the currently active program. The PSW is 128 bits in length and includes the instruction address, condition code, and other control fields. In general, the PSW is used to control instruction sequencing and to hold and indicate much of the status of the CPU in relation to the program currently being executed. Additional control and status information is contained in control registers and permanently assigned storage locations.

The status of the CPU can be changed by loading a new PSW or part of a PSW.

Control is switched during an interruption of the CPU by storing the current PSW, so as to preserve the status of the CPU, and then loading a new PSW.

Execution of LOAD PSW or LOAD PSW EXTENDED, or the successful conclusion of the initial-program-loading sequence, introduces a new PSW. The instruction address is updated by sequential instruction execution and replaced by successful branches. Other instructions are provided which operate on a portion of the PSW. Figure 4-1 on page 4-4 summarizes these instructions.

Instruction	System Mask (PSW Bits 0-7)		PSW Key (PSW Bits 8-11)		Problem State (PSW Bit 15)		Address-Space Control (PSW Bits 16-17)		Condition Code and Program Mask (PSW Bits 18-23)		(PSW Bit 24)		Basic Addressing Mode (PSW Bit 32)		Extended Addressing Mode (PSW Bit 31)	
	Saved	Set	Saved	Set	Saved	Set	Saved	Set	Saved	Set	Saved	Set	Saved	Set	Saved	Set
BRANCH AND LINK	–	–	–	–	–	–	–	–	24AM	–	–	–	31AM	–	–	–
BRANCH AND SAVE	–	–	–	–	–	–	–	–	–	–	–	–	BAM	–	–	–
BRANCH AND SAVE AND SET MODE[6]	–	–	–	–	–	–	–	–	–	–	–	–	BAM	Yes[1]	Yes	Yes[1]
BRANCH AND SET AUTHORITY[7]	–	–	Yes	Yes	Yes	Yes	–	–	–	–	–	–	BAM[2]	BAM	–	–
BRANCH AND SET MODE[6]	–	–	–	–	–	–	–	–	–	–	–	–	BAM[1]	Yes[1]	Yes[1]	Yes[1]
BRANCH AND STACK[7]	Yes	–	Yes	–	Yes	–	Yes	–	Yes	–	Yes	–	BAM[3]	–	Yes	–
BRANCH IN SUBSPACE GROUP[7]	–	–	–	–	–	–	–	–	–	–	–	–	BAM[1]	BAM	–	–
BRANCH RELATIVE AND SAVE	–	–	–	–	–	–	–	–	–	–	–	–	BAM	–	–	–
BRANCH RELATIVE AND SAVE LONG	–	–	–	–	–	–	–	–	–	–	–	–	BAM	–	–	–
EXTRACT PSW	Yes	–	Yes	–	Yes	–	Yes	–	Yes	–	Yes	–	Yes	–	Yes	–
INSERT PROGRAM MASK	–	–	–	–	–	–	–	–	Yes	–	–	–	–	–	–	–
INSERT PSW KEY	–	–	Yes	–	–	–	–	–	–	–	–	–	–	–	–	–
INSERT ADDRESS SPACE CONTROL	–	–	–	–	–	–	Yes	–	–	–	–	–	–	–	–	–
Basic PROGRAM CALL[7]	–	–	–	–	Yes	Yes	–	–	–	–	–	Yes	BAM	BAM	–	–
Stacking PROGRAM CALL[7]	Yes	–	Yes	PKC	Yes	Yes	Yes	Yes	Yes	–	Yes	Yes	Yes	Yes	Yes	Yes
PROGRAM RETURN[7]	–	Yes[4]	–	Yes	–	Yes	–	Yes	–	Yes	–	Yes	–	Yes	–	Yes
PROGRAM TRANSFER[7]	–	–	–	–	–	Yes[5]	–	–	–	–	–	–	–	BAM	–	–
RESUME PROGRAM	–	–	–	–	–	–	–	Yes	–	Yes	–	Yes	–	Yes	–	Yes
SET ADDRESS SPACE CONTROL	–	–	–	–	–	–	–	Yes	–	–	–	–	–	–	–	–
SET ADDRESSING MODE[6]	–	–	–	–	–	–	–	–	–	–	–	–	–	Yes	–	Yes
SET PROGRAM MASK	–	–	–	–	–	–	–	–	–	Yes	–	–	–	–	–	–
SET PSW KEY FROM ADDRESS	–	–	–	Yes	–	–	–	–	–	–	–	–	–	–	–	–
SET SYSTEM MASK[6]	–	Yes	–	–	–	–	–	–	–	–	–	–	–	–	–	–
STORE THEN AND SYSTEM MASK	Yes	ANDs	–	–	–	–	–	–	–	–	–	–	–	–	–	–
STORE THEN OR SYSTEM MASK[6]	Yes	ORs	–	–	–	–	–	–	–	–	–	–	–	–	–	–
TRAP[7]	–	–	–	–	Yes	–	Yes	Yes	Yes	–	–	–	Yes	Yes	Yes	–

Figure 4-1. Operations on PSW Fields.

Instruction	System Mask (PSW Bits 0-7)		PSW Key (PSW Bits 8-11)		Problem State (PSW Bit 15)		Address-Space Control (PSW Bits 16-17)		Condition Code and Program Mask (PSW Bits 18-23)		(PSW Bit 24)		Basic Addressing Mode (PSW Bit 32)		Extended Addressing Mode (PSW Bit 31)	
	Saved	Set	Saved	Set	Saved	Set	Saved	Set	Saved	Set	Saved	Set	Saved	Set	Saved	Set

Explanation:

–	No.
1	The action takes place only if the associated R field in the instruction is nonzero.
2	In the reduced-authority state, the action takes place only if the R_1 field in the instruction is nonzero.
3	The action also takes place in the 64-bit addressing mode if the R_1 field in the instruction is zero.
4	PROGRAM RETURN does not change the PER mask.
5	PROGRAM TRANSFER does not change the problem-state bit from one to zero.
6	Some restrictions apply in the ESA/390-compatibility mode; see instruction description.
7	Not supported in the ESA/390-compatibility mode.
BAM	The basic-addressing-mode bit is saved or set in the 24-bit or 31-bit addressing mode.
ANDs	The logical AND of the immediate field in the instruction and the current system mask replaces the current system mask.
ORs	The logical OR of the immediate field in the instruction and the current system mask replaces the current system mask.
PKC	When the PSW-key-control bit, bit 131 of the entry-table entry, is zero, the PSW key remains unchanged. When the PSW-key-control bit is one, the PSW key is set with the entry key, bits 136-139 of the entry-table entry.
24AM	The condition code and program mask are saved in the 24-bit addressing mode.
31AM	The basic-addressing-mode bit is saved in the 31-bit addressing mode.

Figure 4-1. Operations on PSW Fields. (Continued)

A new or modified PSW becomes active (that is, the information introduced into the current PSW assumes control over the CPU) when the interruption or the execution of an instruction that changes the PSW is completed. The interruption for PER associated with an instruction that changes the PSW occurs under control of the PER mask that is effective at the beginning of the operation.

Bits 0-7 of the PSW are collectively referred to as the system mask.

Programming Notes:

1. A summary of the operations which save or set the problem state, addressing mode, and instruction address is contained in “Subroutine Linkage without the Linkage Stack” on page 5-14.

2. In the ESA/390-compatibility mode, it is unpredictable whether LOAD PSW EXTENDED is supported. If not supported, attempted execution of LPSWE results in an operation exception being recognized.

Program-Status-Word Format

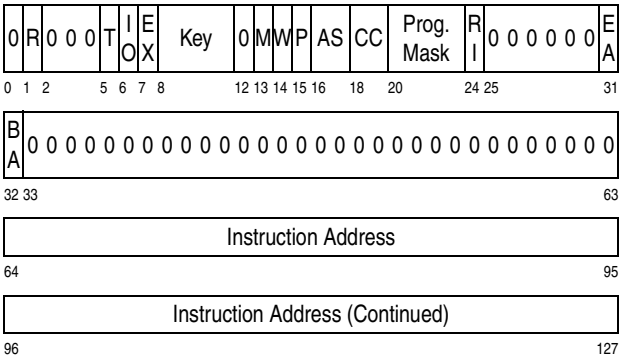


Figure 4-2. PSW Format

The following is a summary of the functions of the PSW fields. (See Figure 4-2.)

PER Mask (R): Bit 1 controls whether the CPU is enabled for interruptions associated with program-event recording (PER). When the bit is zero, no PER event can cause an interruption. When the bit is one, interruptions are permitted, subject to the PER-event-mask bits in control register 9.

Depending on the model, when bit position 1 of the PSW contains one, address-compare controls may be disabled and remain disabled, even if bit position 1 of the PSW transitions back to zero. See “Address-Compare Controls” on page 12-1 for details.

DAT Mode (T): Bit 5 controls whether implicit dynamic address translation of logical and instruction addresses used to access storage takes place. When the bit is zero, DAT is off, and logical and instruction addresses are treated as real addresses. When the bit is one, DAT is on, and the dynamic-address-translation mechanism is invoked.

In the ESA/390-compatibility mode, bit 5 of the PSW must be zero; otherwise, a specification exception is recognized.

I/O Mask (IO): Bit 6 controls whether the CPU is enabled for I/O interruptions. When the bit is zero, an I/O interruption cannot occur. When the bit is one, I/O interruptions are subject to the I/O-interruption subclass-mask bits in control register 6. When an I/O-interruption subclass-mask bit is zero, an I/O interruption for that I/O-interruption subclass cannot occur; when the I/O-interruption subclass-mask bit is one, an I/O interruption for that I/O-interruption subclass can occur.

External Mask (EX): Bit 7 controls whether the CPU is enabled for interruption by conditions included in the external class. When the bit is zero, an external interruption cannot occur. When the bit is one, an external interruption is subject to the corresponding external subclass-mask bits in control register 0; when the subclass-mask bit is zero, conditions associated with the subclass cannot cause an interruption; when the subclass-mask bit is one, an interruption in that subclass can occur.

PSW Key: Bits 8-11 form the access key for storage references by the CPU. If the reference is subject to key-controlled protection, the PSW key is matched with a storage key when information is stored or when information is fetched from a location that is protected against fetching. However, for one of the operands of each of MOVE TO PRIMARY, MOVE TO SECONDARY, MOVE WITH KEY, MOVE WITH SOURCE KEY, and MOVE WITH DESTINATION KEY, and for either or both operands of MOVE WITH OPTIONAL SPECIFICATIONS, an access key specified as an operand is used instead of the PSW key.

Bit 12: Bit 12 of the 16-byte z/Architecture PSW must be zero; otherwise, a specification exception is recognized. This specification exception may be recognized either when the instruction that attempted to load the PSW ends with suppression or when the newly-loaded PSW becomes active.

Programming Note: Bit 12 of an 8-byte short-format PSW in storage is inverted when the PSW is loaded. Therefore, bit 12 of the short-format PSW must be one. See “Short PSW Format” on page 4-7.

Machine-Check Mask (M): Bit 13 controls whether the CPU is enabled for interruption by machine-check conditions. When the bit is zero, a machine-check interruption cannot occur. When the bit is one, machine-check interruptions due to system damage and instruction-processing damage are permitted, but interruptions due to other machine-check-subclass conditions are subject to the subclass-mask bits in control register 14.

Wait State (W): When bit 14 is one, the CPU is waiting; that is, no instructions are processed by the CPU, but interruptions may take place. When bit 14 is zero, instruction fetching and execution occur in the normal manner. The wait indicator is on when the bit is one.

Problem State (P): When bit 15 is one, the CPU is in the problem state. When bit 15 is zero, the CPU is in the supervisor state. In the supervisor state, all instructions are valid. In the problem state, only those instructions are valid that provide meaningful information to the problem program and that cannot affect system integrity; such instructions are called unprivileged instructions. The instructions that are never valid in the problem state are called privileged instructions. When a CPU in the problem state attempts to execute a privileged instruction, a privileged-operation exception is recognized. Another group of instructions, called semiprivileged instructions, are executed by a CPU in the problem state only if specific authority tests are met; otherwise, a privileged-operation exception or some other program exception is recognized, depending on the particular requirement which is violated.

Address-Space Control (AS): In the z/Architecture architectural mode, bits 16 and 17, in conjunction with PSW bit 5, control the translation mode. See “Translation Modes” on page 3-40. In the ESA/390-compatibility mode, bits 16-17 may operate as defined in Reference [12.] on page xxx.

Condition Code (CC): Bits 18 and 19 are the two bits of the condition code. The condition code is set to 0, 1, 2, or 3, depending on the result obtained in executing certain instructions. Most arithmetic and

logical operations, as well as some other operations, set the condition code. The instruction BRANCH ON CONDITION can specify any selection of the condition-code values as a criterion for branching. A table in Appendix C summarizes the condition-code values that may be set for all instructions which set the condition code of the PSW.

Program Mask: Bits 20-23 are the four program-mask bits. Each bit is associated with a program exception, as follows:

Program-Mask Bit	Program Exception
20	Fixed-point overflow
21	Decimal overflow
22	HFP exponent underflow
23	HFP significance

When the mask bit is one, the exception results in an interruption. When the mask bit is zero, no interruption occurs. The setting of the HFP-exponent-underflow-mask bit or the HFP-significance-mask bit also determines the manner in which the operation is completed when the corresponding exception occurs.

Bit 24 (RI): Bit 24 is reserved for IBM use. An early specification exception may be recognized when bit 24 of the operand of LPSWE is one.

Extended Addressing Mode (EA): Bit 31 controls the size of effective addresses and effective-address generation in conjunction with bit 32, the basic-addressing-mode bit. When bit 31 is zero, the addressing mode is controlled by bit 32. When bits 31 and 32 are both one, 64-bit addressing is specified.

In the ESA/390-compatibility mode, it is unpredictable whether a specification exception is recognized when bit 31 of the PSW is one.

Basic Addressing Mode (BA): Bits 31 and 32 control the size of effective addresses and effective-address generation. When bits 31 and 32 are both zero, 24-bit addressing is specified. When bit 31 is zero and bit 32 is one, 31-bit addressing is specified. When bits 31 and 32 are both one, 64-bit addressing is specified. Bit 31 one and bit 32 zero is an invalid combination that causes a specification exception to be recognized. The addressing mode does not control the size of PER addresses or of addresses used to access DAT, ASN, dispatchable-unit-control, linkage, entry, and trace tables or access lists or the linkage stack. See “Address Generation” on page 5-10 and “Address Size and Wraparound” on page 3-6. The control of the addressing mode by bits 31 and 32 of the PSW is summarized as follows:

PSW.31	PSW.32	Addressing Mode
0	0	24-bit
0	1	31-bit
1	1	64-bit

Instruction Address: Bits 64-127 of the PSW are the instruction address. This address designates the location of the leftmost byte of the next instruction to be executed, unless the CPU is in the wait state (bit 14 of the PSW is one).

Bit positions 0, 2-4, 25-30, and 33-63 are unassigned and must contain zeros. A specification exception is recognized when these bit positions do not contain zeros.

When bits 31 and 32 of the PSW specify the 24-bit addressing mode, bits 64-103 of the instruction address must be zeros, or, when bits 31 and 32 specify the 31-bit mode, bits 64-96 must be zeros. Otherwise, a specification exception is recognized. A specification exception is also recognized when bit 31 is one and bit 32 is zero or when bit position 12 does not contain a zero.

LOAD PSW EXTENDED has a 16-byte second operand, as shown in Figure 4-2 on page 4-5. The instruction loads the operand unchanged and without examination as the current PSW.

Short PSW Format

Certain instructions are capable of loading the full 128-bit PSW, or portions thereof, from a shorter doubleword storage location, called a *short PSW.* The short PSW, shown in Figure 4-3, is similar to the ESA/390-format PSW in that bit 12 must be one. However, unlike the ESA/390-format PSW in which

bit 31 must be zero, bit 31 of the short PSW may be either zero or one in the short PSW.

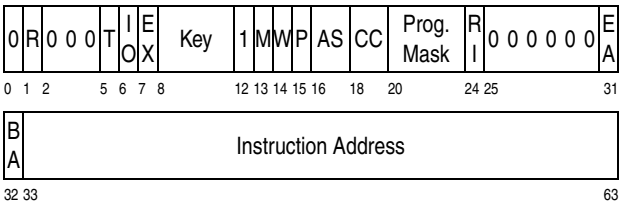


Figure 4-3. Short PSW Format

The bit positions of the short PSW and the corresponding bit positions in the full 128-bit PSW are shown in Figure 4-4.

PSW Bit Positions	Short-PSW Bit Positions
0-11	0-11
12 *	12 *
13-32	13-32
33-96	—
97-127	33-63

Explanation:

* Bit 12 must be zero in the full 128-bit PSW and one in the short PSW. Bit 12 of the short PSW is inverted when loaded into the full 128-bit PSW. While in the ESA/390 compatibility mode, bit 12 of the full 128-bit PSW is inverted (a) in the old PSW stored during an interruption, and (b) in the results placed in bits 32-63 of general register R_1 by EXTRACT PSW. In the results stored by EXTRACT STACKED STATE when the code is one, bit 12 of general register R_1 is one, regardless of the architectural mode.

— Zeros are placed in bit positions 33-96 of the full 128-bit PSW when a short PSW is loaded.

Figure 4-4. PSW and Short-PSW Bit Positions

LOAD PSW and the initial-program-loading (IPL) process translate a short PSW into the full 128-bit PSW, as shown in Figure 4-4. When the P control (bit 13) of the RESUME PROGRAM instruction's parameter list is zero, the instruction loads selected bits of a short PSW into the full 128-bit PSW.

In the ESA/390-compatibility mode, the PSW used in assigned storage locations and in the LOAD PSW and RESUME PROGRAM instructions has the same format as the short PSW except that when loading the PSW, (a) bit 5 must be zero, and (b) it is unpredictable whether bit 31 must be zero.

Depending on the model, either LOAD PSW (LPSW) recognizes a specification exception if bit 12 of its second operand is not one, or this error is indicated by an early specification exception after the completion of the execution of the LOAD PSW.

Note: The term short PSW refers only to the eight-byte entity in storage. The actual PSW used by the CPU is the 16-byte register illustrated in Figure 4-2.

Control Registers

The control registers provide for maintaining and manipulating control information outside the PSW. There are sixteen 64-bit control registers, however in the ESA/390-compatibility mode, bits 0-31 of each control register contains zeros.

The LOAD CONTROL (LCTLG) instruction causes all control-register bit positions within those registers designated by the instruction to be loaded from storage. In the ESA/390-compatibility mode, LCTLG is not supported; attempted execution of LCTLG results in an operation exception being recognized.

The LOAD CONTROL (LCTL) instruction loads only bit positions 32-63 of the control registers, and bits 0-31 of the registers remain unchanged. The instructions BRANCH AND SET AUTHORITY, BRANCH AND STACK, BRANCH IN SUBSPACE GROUP, EXTRACT AND SET EXTENDED AUTHORITY, LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, and SET SECONDARY ASN WITH INSTANCE, provide specialized functions to place information into certain control-register bit positions.

Information loaded into the control registers becomes active (that is, assumes control over the system) at the completion of the instruction that causes the information to be loaded.

At the time the registers are loaded, the information is not checked for exceptions, such as an address designating an unavailable or protected location. The validity of the information is checked and the exceptions, if any, are indicated at the time the information is used.

The STORE CONTROL (STCTG) instruction causes the contents of all control-register bit positions, within those registers designated by the instruction, to be

placed in storage. The STORE CONTROL (STCTL) instruction places the contents of bit positions 32-63 of the control registers in storage, and bits 0-31 of the registers are ignored. The instructions EXTRACT AND SET EXTENDED AUTHORITY, EXTRACT PRIMARY ASN, EXTRACT PRIMARY ASN AND INSTANCE, EXTRACT SECONDARY ASN, EXTRACT SECONDARY ASN AND INSTANCE, and PROGRAM CALL provide specialized functions to obtain information from certain control-register bit positions.

Only the general structure of the control registers is described here; the definition of a particular control-register bit position appears in the description of the facility with which the position is associated. Figure 4-5 on page 4-10 shows the control-register bit positions which are assigned and the initial values of the positions upon execution of initial CPU reset. All control-register bit positions not listed in the figure are initialized to zero.

The following applies when the configuration is operating in the ESA/390-compatibility mode:

- Bits 0-31 of all control registers contain zeros.
- It is unpredictable whether any of bits 32-63 of a control register that are unique to z/Architecture features have any effect.

Control Register	Bits	Name of Field	Associated with	Initial Value
0	8	Transactional-execution control[5]	Transactional execution	0
0	9	Transactional-execution program-interruption filtering override[5]	Transactional execution	0
0	10	Clock-comparator sign control	Multiple-epoch facility	0
0	15	Measurement-counter-extraction-authorization control[5]	CPU measurement[4]	0
0	30	Warning-track subclass mask[5]	Virtual machines	0
0	32	TRACE TOD-clock control[6]	Tracing	0
0	33	SSM-suppression control	SET SYSTEM MASK	0
0	34	TOD-clock-sync control	TOD clock	0
0	35	Low-address-protection control	Low-address-protection	0
0	36	Extraction-authority control	Instruction authorization	0
0	37	Secondary-space control[5]	Instruction authorization	0
0	38	Fetch-protection-override control	Key-controlled protection	0
0	39	Storage-protection-override control	Key-controlled protection	0
0	40	Enhanced-DAT-enablement control[5]	Dynamic address translation	0
0	43	Instruction-execution-protection-enablement control[5]	Instruction-execution protection	0
0	44	ASN-and-LX-reuse control[5]	Instruction authorization	0
0	45	AFP-register control	Floating point	0
0	46	Vector enablement control	Vector facility for z/Architecture	0
0	48	Malfunction-alert subclass mask	External interruptions	0
0	49	Emergency-signal subclass mask	External interruptions	0
0	50	External-call subclass mask	External interruptions	0
0	52	Clock-comparator subclass mask	External interruptions	0
0	53	CPU-timer subclass mask	External interruptions	0
0	54	Service-signal subclass mask	External interruptions	0
0	56	Unused[1]		1
0	57	Interrupt-key subclass mask	External interruptions	1
0	58	Measurement-alert subclass mask	CPU measurement[4]	1
0	59	Timing-alert subclass mask	External interruptions	0
0	61	Crypto control	Cryptography	0
1	0-51	Primary region-table origin[2,5]	Dynamic address translation	0
1	0-51	Primary segment-table origin[2,5]	Dynamic address translation	0
1	0-51	Primary real-space token origin[2,5]	Dynamic address translation	0
1	54	Primary subspace-group control[5]	Subspace groups	0
1	55	Primary private-space control[5]	Dynamic address translation	0
1	56	Primary storage-alteration-event[5]	Program-event recording control	0
1	57	Primary space-switch-event control[5]	Program interruptions	0
1	58	Primary real-space control[5]	Dynamic address translation	0
1	60-61	Primary designation-type control[3,5]	Dynamic address translation	0
1	62-63	Primary table length[3,5]	Dynamic address translation	0
2	33-57	Dispatchable-unit-control-table origin	Access-register translation	0
2	59	Guarded-storage-facility enablement control[6]	Guarded-storage facility	0
2	61	Transaction diagnostic scope[5]	Transactional execution	0
2	62-63	Transaction diagnostic control[5]	Transactional execution	0
3	0-31	Secondary ASN-second-table-entry instance number[5]	Instruction authorization	0
3	32-47	PSW-key mask	Instruction authorization	0
3	48-63	Secondary ASN[5]	Address spaces	0

Figure 4-5. Assignment of Control-Register Fields (Part 1 of 3)

Control Register	Bits	Name of Field	Associated with	Initial Value
4	0-31	Primary ASN-second-table-entry instance number[5]	Instruction authorization	0
4	32-47	Authorization index[5]	Instruction authorization	0
4	48-63	Primary ASN[5]	Address spaces	0
5	33-57	Primary-ASN-second-table-entry origin[5]	Access-register translation	0
6	32-39	I/O-interruption subclass mask	I/O interruptions	0
7	0-51	Secondary region-table origin[2,5]	Dynamic address translation	0
7	0-51	Secondary segment-table origin[2,5]	Dynamic address translation	0
7	0-51	Secondary real-space token origin[2,5]	Dynamic address translation	0
7	54	Secondary subspace-group control[5]	Subspace groups	0
7	55	Secondary private-space control[5]	Dynamic address translation	0
7	56	Secondary storage-alteration-event[5]	Program-event recording control	0
7	58	Secondary real-space control[5]	Dynamic address translation	0
7	60-61	Secondary designation-type control[3,5]	Dynamic address translation	0
7	62-63	Secondary table length[3,5]	Dynamic address translation	0
8	16-31	Enhanced-monitor masks[5]	MONITOR CALL	0
8	32-47	Extended authorization index	Access-register translation	0
8	48-63	Monitor masks	MONITOR CALL	0
9	32	Successful-branching-event mask	Program-event recording	0
9	33	Instruction-fetching-event mask	Program-event recording	0
9	34	Storage-alteration-event mask	Program-event recording	0
9	36	Store-using-real-address-event mask	Program-event recording	0
9	37	Zero-address-detection-event mask[6]	Program-event recording	0
9	38	Transaction-end event mask[5]	Program-event recording	0
9	39	Instruction-fetching-nullification-event mask[6]	Program-event recording	0
9	40	Branch-address control	Program-event recording	0
9	41	Per-event-suppression control[5]	Program-event recording	0
9	42	Storage-alteration-space control[5]	Program-event recording	0
10	0-63	PER starting address[7]	Program-event recording	0
11	0-63	PER ending address[7]	Program-event recording	0
12	0	Branch-trace control[5]	Tracing	0
12	1	Mode-trace control[5]	Tracing	0
12	2-61	Trace-entry address[7]	Tracing	0
12	62	ASN-trace control[5]	Tracing	0
12	63	Explicit-trace control	Tracing	0
13	0-51	Home region-table origin[2,5]	Dynamic address translation	0
13	0-51	Home segment-table origin[2,5]	Dynamic address translation	0
13	0-51	Home real-space token origin[2,5]	Dynamic address translation	0
13	55	Home private-space control[5]	Dynamic address translation	0
13	56	Home storage-alteration-event[5]	Program-event recording control	0
13	57	Home space-switch-event control[5]	Program interruptions	0
13	58	Home real-space control[5]	Dynamic address translation	0
13	60-61	Home designation-type control[3,5]	Dynamic address translation	0
13	62-63	Home table length[3,5]	Dynamic address translation	0

Figure 4-5. Assignment of Control-Register Fields (Part 2 of 3)

Control Register	Bits	Name of Field	Associated with	Initial Value
14	32	Unused[1]		1
14	33	Unused[1]		1
14	34	Extended save-area control (ESA/390-compatibility mode only)	Floating point	0
14	35	Channel-report-pending subclass mask	I/O machine-check handling	0
14	36	Recovery subclass mask	Machine-check handling	0
14	37	Degradation subclass mask	Machine-check handling	0
14	38	External-damage subclass mask	Machine-check handling	1
14	39	Warning subclass mask	Machine-check handling	0
14	42	TOD-clock-control-override control	TOD clock	0
14	44	ASN-translation control[5]	Instruction authorization	0
14	45-63	ASN-first-table origin[5]	ASN translation	0
15	0-60	Linkage-stack-entry address[5]	Linkage-stack operations	0

Explanation:

The fields not listed are unassigned. The initial value for all unlisted control-register bit positions is zero.

1 This bit is not used but is initialized to one for consistency with the System/370 definition.

2 The address-space-control element (ASCE) in the control register has one of three formats, depending on bit 58 of the register, the real-space control, and bits 60 and 61 of the register, the designation-type control. When bit 58 is zero, the ASCE is a region-table designation if bits 60 and 61 are 11, 10, or 01 binary, or it is a segment-table designation if bits 60 and 61 are 00 binary. When bit 58 is one, the ASCE is a real-space designation. Bits 0-51 are the region-table origin, the segment-table origin or the real-space token origin, depending on whether the ASCE is a region-table designation, a segment-table designation, or a real-space designation, respectively.

3 Bits 60-63 are assigned when the ASCE in the control register is a region-table designation or a segment-table designation.

4 See *The Set-Program-Parameter and CPU-Measurement Facilities* (SA23-2260) for details on these bits.

5 Not applicable in the ESA/390-compatibility mode.

6 It is unpredictable whether this bit has any effect in the ESA/390-compatibility mode.

7 In the ESA/390-compatibility mode. bits 0-31 of the register are all zeros. Results are unpredictable if bit 32 of the register is one.

Figure 4-5. Assignment of Control-Register Fields (Part 3 of 3)

Programming Notes:

1. The detailed definition of a particular control-register bit position can be located by referring to the entry “control-register assignment” in the Index.
2. To ensure that existing programs operate correctly if and when new facilities using additional control-register bit positions are installed, the program should load zeros in unassigned positions.

Tracing

Tracing assists in the determination of system problems by providing an ongoing record in storage of significant events. Tracing consists of four separately controllable functions which cause entries to be made in a trace table: branch tracing, ASN tracing, mode tracing, and explicit tracing.

Implicit Tracing

Branch tracing, ASN tracing, and mode tracing together are referred to as implicit tracing. In the ESA/390-compatibility mode, implicit tracing is not supported.

Branch Tracing

When branch tracing is on, a branch trace entry is made in the trace table for each execution of certain branch instructions when they cause branching. The branch address is placed in the trace entry. The trace entry also indicates the following about the addressing mode in effect after branching and the branch

address: (1) the CPU is in the 24-bit addressing mode, (2) the CPU either is in the 31-bit addressing mode or is in the 64-bit addressing mode and bits 0-32 of the branch address are all zeros, or (3) the CPU is in the 64-bit addressing mode and bits 0-32 of the branch address are not all zeros. The branch instructions that are traced are:

- BRANCH AND LINK (BALR only) when the R_2 field is not zero
- BRANCH AND SAVE (BASR only) when the R_2 field is not zero
- BRANCH AND SAVE AND SET MODE when the R_2 field is not zero
- BRANCH AND SET AUTHORITY
- BRANCH AND STACK when the R_2 field is not zero
- BRANCH IN SUBSPACE GROUP
- RESUME PROGRAM
- TRAP

However, a branch trace entry is made for BRANCH IN SUBSPACE GROUP only if ASN tracing is not on.

If both branch tracing and mode tracing are on and BRANCH AND SAVE AND SET MODE or RESUME PROGRAM changes the extended-addressing-mode bit, PSW bit 31, a mode-switching-branch trace entry is made instead of a branch trace entry.

ASN Tracing

When ASN tracing is on, an entry named the same as the instruction is made in the trace table for each execution of the following instructions:

- BRANCH IN SUBSPACE GROUP
- PROGRAM CALL
- PROGRAM RETURN
- PROGRAM TRANSFER
- SET SECONDARY ASN

However, the entry for PROGRAM RETURN is made only when PROGRAM RETURN unstacks a linkage-stack state entry that was formed by PROGRAM CALL, not when PROGRAM RETURN unstacks an entry formed by BRANCH AND STACK.

When ASN tracing is on, a PROGRAM TRANSFER trace entry is also made for each execution of PROGRAM TRANSFER WITH INSTANCE, and a SET SECONDARY ASN trace entry is also made for each execution of SET SECONDARY ASN WITH INSTANCE. In either case, a bit in the trace entry indicates whether the entry was made due to the without-instance or the with-instance instruction.

If both ASN tracing and mode tracing are on and PROGRAM CALL uses a 20-bit PC number and changes PSW bit 31, first a PROGRAM CALL trace entry is made, and then a mode-switch trace entry is made. In this case except when PROGRAM CALL uses a 32-bit PC number, only a PROGRAM CALL trace entry is made since it indicates the old and new values of the extended-addressing-mode bit, PSW bit 31. A 32-bit PC number may be used if the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control, bit 44 of control register 0.

Mode Tracing

Mode tracing records a switch from a basic (24-bit or 31-bit) addressing mode to the extended (64-bit) addressing mode or from the extended mode to a basic mode.

When mode tracing is on, a mode-switch trace entry is made in the trace table for each execution of the following instructions if the execution changes PSW bit 31:

- BRANCH AND SAVE AND SET MODE
- BRANCH AND SET MODE
- PROGRAM CALL
- PROGRAM RETURN
- RESUME PROGRAM
- SET ADDRESSING MODE

However, a mode-switch trace entry is not made for PROGRAM RETURN if ASN tracing is on and PROGRAM RETURN unstacks a state entry formed by PROGRAM CALL; a PROGRAM RETURN trace entry is made instead, and it contains information about PSW bit 31.

BRANCH AND SAVE AND SET MODE and RESUME PROGRAM cause trace entries to be made as follows: a branch trace entry if only branch tracing is on, a mode-switching-branch trace entry if both branch tracing and mode tracing are on, or a mode-switch trace entry if only mode tracing is on.

The trace entries produced by implicit tracing are summarized in Figure 4-6.

Instruction	Implicit Tracing Enabled						
	Branch	ASN	Mode	Branch and ASN	Branch and Mode	ASN and Mode	All
	Trace Entries Made						
BAKR	B	–	–	B	B	–	B
BALR	B	–	–	B	B	–	B
BASR	B	–	–	B	B	–	B
BASSM	B	–	MS	B	B | MSB	MS	B | MSB
BSA	B	–	–	B	B	–	B
BSG	B	BSG	–	BSG	B	BSG	BSG
BSM	–	–	MS	–	MS	–	MS
PC-20	–	PC	MS	PC	MS	PC & MS	PC & MS
PC-32	–	PC	MS	PC	MS	PC	PC
PR-b	–	–	MS	–	MS	MS	MS
PR-pc	–	PR	MS	PR	MS	PR	PR
PT or PTI	–	PT	–	PT	–	PT	PT
RP	B	–	MS	B	B | MSB	MS	B | MSB
SSAR or SSAIR	–	SSAR	–	SSAR	–	SSAR	SSAR
SAM24/31/64	–	–	MS	–	MS	MS	MS
TRAP2/4	B	–	–	B	B	–	B

Explanation:

–	None.
-20	The case when PROGRAM CALL uses a 20-bit PC number.
-32	The case when PROGRAM CALL uses a 32-bit PC number.
-b	The case when PROGRAM RETURN unstacks a branch state entry.
-pc	The case when PROGRAM RETURN unstacks a program-call state entry.
|	OR.
&	AND.
B	Branch trace entry. Made only if the branch is taken and a mode-switching-branch trace entry is not made.
MS	Mode-switch trace entry. Made only if PSW bit 31 is changed.
MSB	Mode-switching-branch trace entry. Made only if PSW bit 31 is changed (which can occur only if the branch is taken.

Figure 4-6. Summary of Implicit Tracing

Explicit Tracing

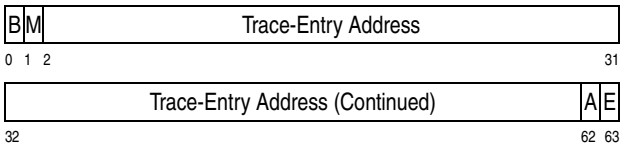
Note: In the following discussion, the term “TRACE” by itself refers to the generic instruction name which includes the specific instructions having the mnemonics TRACE and TRACG. To avoid ambiguity, when a specific instruction is described, the generic name TRACE will be followed by the specific instruction’s mnemonic in parentheses.

When explicit tracing is on, execution of TRACE (TRACE or TRACG) causes an entry to be made in the trace table. The entry for TRACE (TRACE) includes bits 16-63 from the TOD clock, the second operand of the TRACE instruction, and bits 32-63 of a range of general registers. The entry for TRACE (TRACG) is the same except that it includes bits 0-79 from the TOD clock and bits 0-63 of a range of general registers. When the multiple-epoch facility is installed, the entry for TRACE (TRACG) also includes bits 1-7 of the epoch index.

When the CPU is in the transactional-execution mode, an instruction that would otherwise cause tracing to occur is restricted. Attempted execution of such an instruction causes the transaction to be aborted with abort code 11, and the condition code is set to 3.

Control-Register Allocation

The information to control tracing is contained in control register 12 and has the following format:



Branch-Trace-Control Bit (B): Bit 0 of control register 12 controls whether branch tracing is turned on or off. If the bit is zero, branch tracing is off; if the bit is one, branch tracing is on.

Mode-Trace-Control Bit (M): Bit 1 of control register 12 controls whether mode tracing is turned on or off. If the bit is zero, mode tracing is off; if the bit is one, mode tracing is on.

Trace-Entry Address: Bits 2-61 of control register 12, with two zero bits appended on the left and two on the right, form the real address of the next trace entry to be made.

In the ESA/390-compatibility mode, it is unpredictable whether bit 32 of the trace-entry address is treated as being zero.

ASN-Trace-Control Bit (A): Bit 62 of control register 12 controls whether ASN tracing is turned on or off. If the bit is zero, ASN tracing is off; if the bit is one, ASN tracing is on.

Explicit-Trace-Control Bit (E): Bit 63 of control register 12 controls whether explicit tracing is turned on or off. If the bit is zero, explicit tracing is off, which causes the TRACE instruction to be executed as a no-operation; if the bit is one, the execution of the TRACE instruction creates an entry in the trace table, except that no entry is made when bit 0 of the second operand of the TRACE instruction is one.

Programming Note: The following considerations apply to tracing in the ESA/390-compatibility mode:

- Bits 0-31 of control register 12 are always zero (the LCTLG instruction is not supported, thus there is no means by which bits 0-31 can be set to a nonzero value). This means that neither branch- nor mode-tracing can be performed.
- If bit 32 of control register 12 is one, it is unpredictable whether the bit is ignored and treated as zero, or the bit is treated as bit 32 of the trace-entry address. If bit 32 of the trace-entry address is one, then a trace entry may be formed between 2- and 4 G-bytes if the real storage is available; otherwise, an addressing exception may be recognized if the real storage is not available.
- Instructions that perform ASN tracing require that dynamic-address translation be enabled. DAT is not supported in the ESA/390-compatibility mode, thus the ASN-tracing control is ignored.

Trace Entries

Trace entries are of nine types, with most types having more than one detailed format. The types and numbers of formats are as follows:

- Branch (three formats)
- BRANCH IN SUBSPACE GROUP (two formats)
- Mode switch (three formats)
- Mode-switching branch (three formats)
- PROGRAM CALL (seven formats)
- PROGRAM RETURN (nine formats)
- PROGRAM TRANSFER (three formats)
- SET SECONDARY ASN (one format)
- TRACE (two formats)

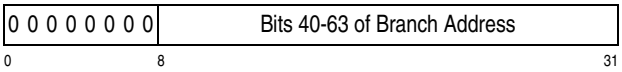
Format-1 and format-2 PROGRAM CALL trace entries are made if the ASN-and-LX reuse facility is not enabled. Entries of formats 3-7 are made if the facility is enabled.

The PROGRAM TRANSFER trace entry is also made for PROGRAM TRANSFER WITH INSTANCE, and the SET SECONDARY ASN trace entry is also made for SET SECONDARY ASN WITH INSTANCE. In either case, bit 15 (N) of the entry is one if the entry was made because of execution of the with-instance instruction.

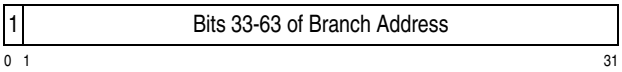
The entries are shown in Figure 4-7. In that figure, each entry is labeled with “Fn,” indicating a format number, to allow references to each format within a trace-entry type. Also, “Branch,” referring to the mnemonic of an instruction that causes a branch trace entry, refers to BAKR, BALR, BASR, BASSM, BSA, or BSG.

Figure 4-7 lists the trace entries in ascending order of values in bit fields that identify the entries.

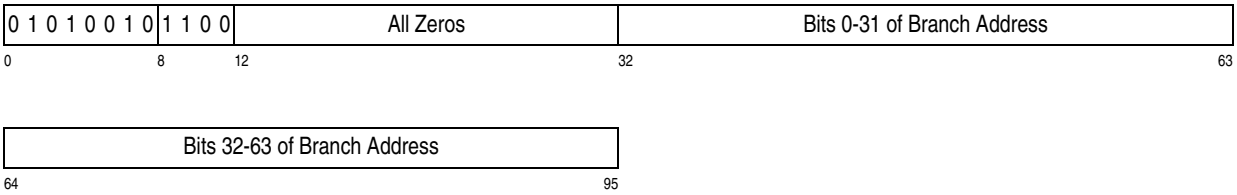
F1 Branch (Branch, RP, or TRAP2/4 when Resulting Mode Is 24-Bit)



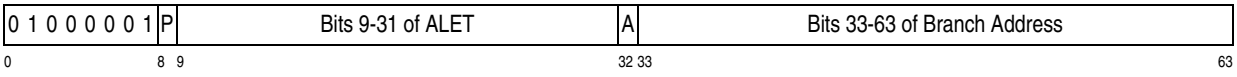
F2 Branch (Branch, RP, or TRAP2/4 when Resulting Mode Is 31-Bit, or when Resulting PSW Bit 31 is One (See Note) and Bits 0-32 of Branch Address Are All Zeros)



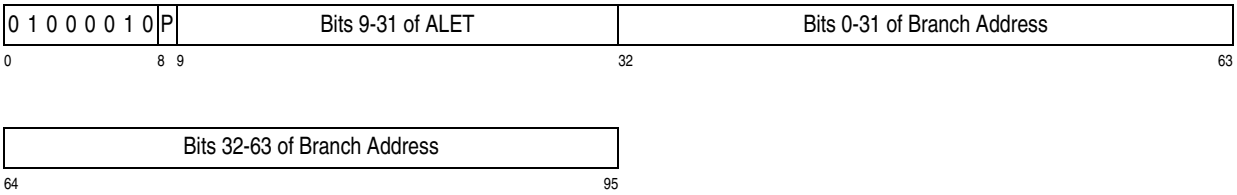
F3 Branch (Branch, RP, or TRAP2/4 when Resulting PSW Bit 31 Is One (See Note) and Bits 0-32 of Branch Address Are Not All Zeros)



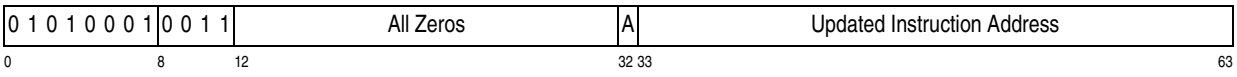
F1 BRANCH IN SUBSPACE GROUP (if ASN Is Tracing on, in 24-Bit or 31-Bit Mode)



F2 BRANCH IN SUBSPACE GROUP (if ASN Is Tracing on, in 64-Bit Mode)



F1 Mode Switch (BASSM, BSM, PC, PR, RP, or SAM64 from 24-Bit or 31-Bit Mode when Resulting PSW Bit 31 is One (See Note))



F2 Mode Switch (BASSM, BSM, PC, PR, RP, SAM24, or SAM31 from 64-Bit Mode to 24-Bit or 31-Bit Mode when Bits 0-31 of Updated Instruction Address Are All Zeros)

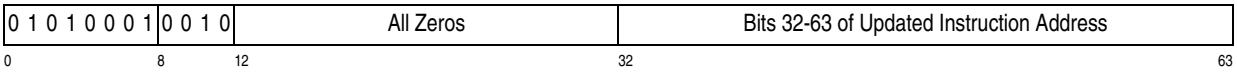
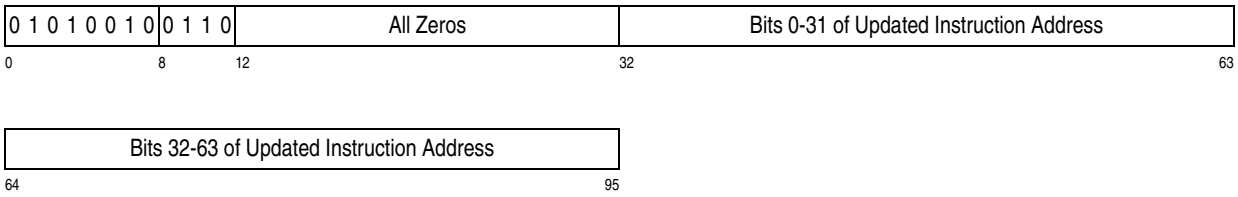
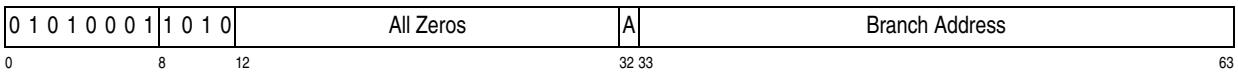


Figure 4-7. Trace Entries (Part 1 of 7)

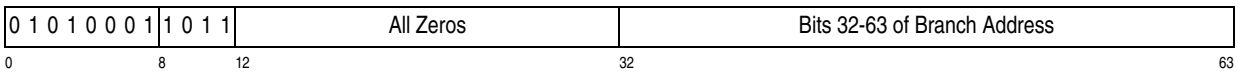
F3 Mode Switch (BASSM, BSM, PC, PR, or RP from 64-Bit Mode to 24-Bit or 31-Bit Mode when Bits 0-31 of Updated Instruction Address Are Not All Zeros)



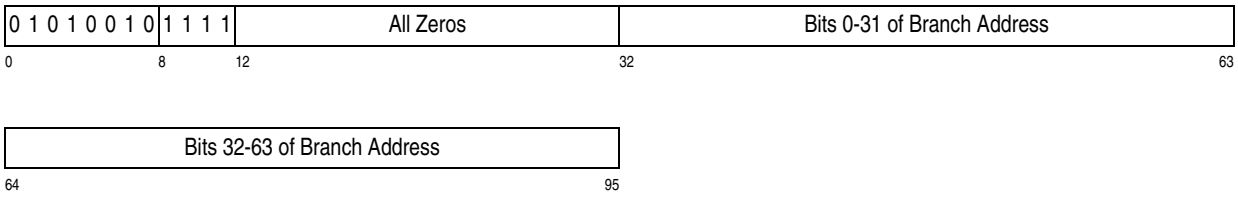
F1 Mode-Switching Branch (BASSM or RP from 64-Bit Mode to 24-Bit or 31-Bit Mode)



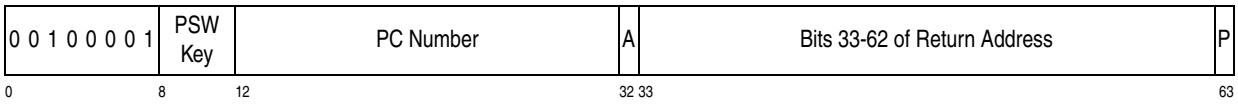
F2 Mode-Switching Branch (BASSM or RP from 24-Bit or 31-Bit Mode when Resulting PSW Bit 31 Is One (See Note) and Bits 0-31 of Branch Address Are All Zeros)



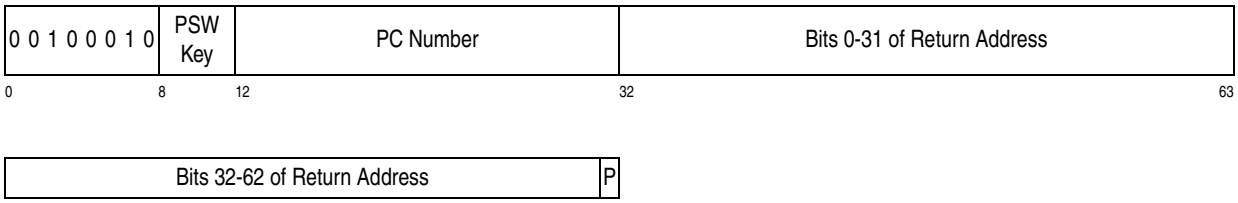
F3 Mode-Switching Branch (BASSM or RP from 24-Bit or 31-Bit Mode when Resulting PSW Bit 31 Is One (See Note) and Bits 0-31 of Branch Address Are Not All Zeros)



F1 PROGRAM CALL (in 24-Bit or 31-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Not Enabled)



F2 PROGRAM CALL (in 64-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Not Enabled)



F3 PROGRAM CALL (in 24-Bit or 31-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Enabled and 20-Bit PC Number is Used)

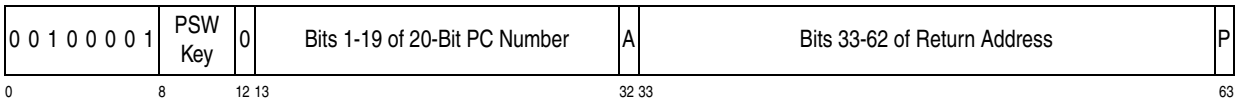
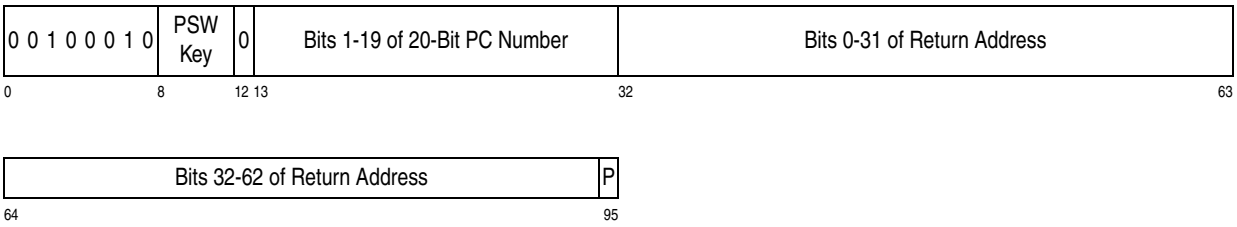
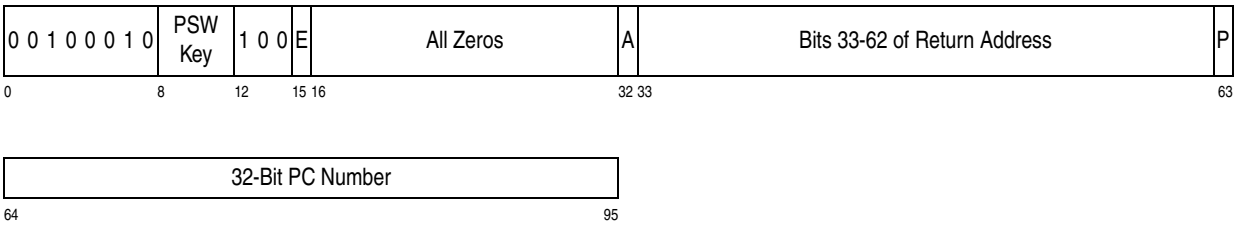


Figure 4-7. Trace Entries (Part 2 of 7)

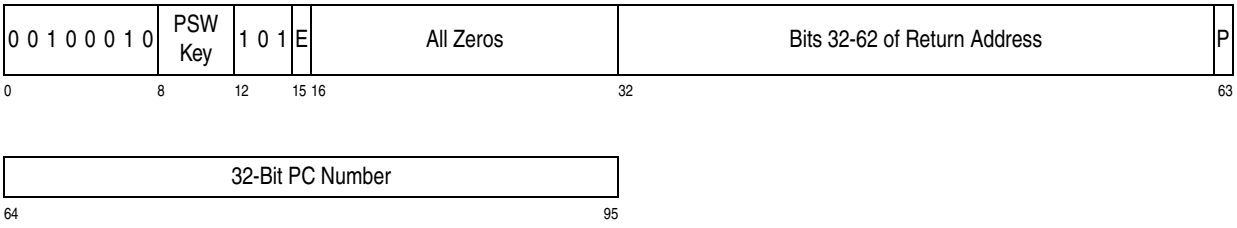
F4 PROGRAM CALL (in 64-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Enabled and 20-Bit PC Number Is Used)



F5 PROGRAM CALL (in 24-Bit or 31-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Enabled and 32-Bit PC Number is Used)



F6 PROGRAM CALL (in 64-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Enabled, 32-Bit PC Number Is Used, and Bits 0-31 of Return Address Are All Zeros)



F7 PROGRAM CALL (in 64-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Enabled, 32-Bit PC Number Is Used, and Bits 0-31 of Return Address Are Not All Zeros)

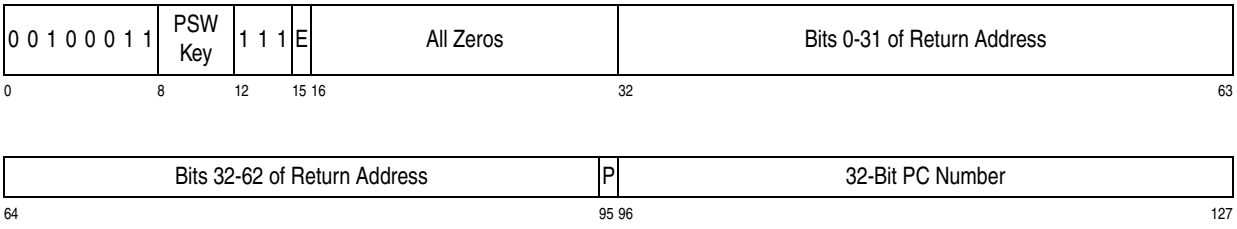
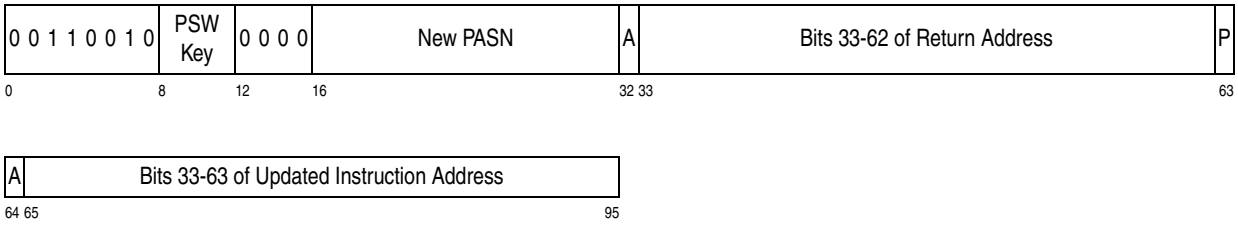
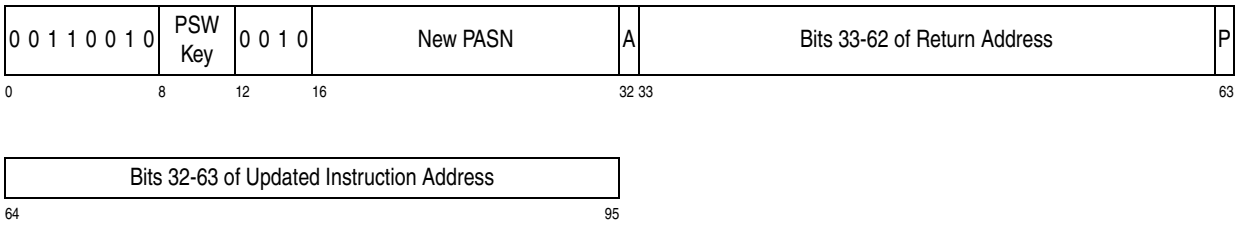


Figure 4-7. Trace Entries (Part 3 of 7)

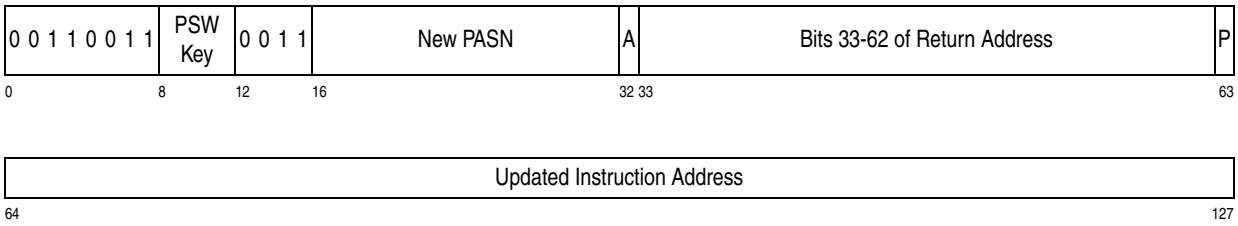
F1 PROGRAM RETURN (in 24-Bit or 31-Bit Mode when Resulting Mode Is 24-Bit or 31-Bit)



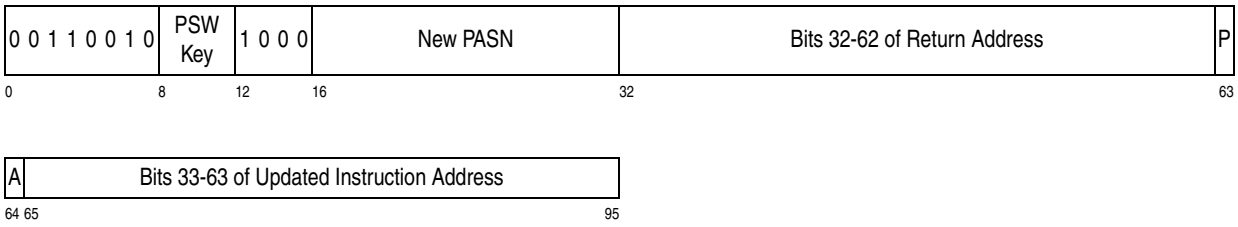
F2 PROGRAM RETURN (in 64-Bit Mode when Bits 0-31 of Updated Instruction Address Are All Zeros and Resulting Mode Is 24-Bit or 31-Bit)



F3 PROGRAM RETURN (in 64-Bit Mode when Bits 0-31 of Updated Instruction Address Are Not All Zeros and Resulting Mode Is 24-Bit or 31-Bit)



F4 PROGRAM RETURN (in 24-Bit or 31-Bit Mode when Resulting PSW Bit 31 Is One (See Note) and Bits 0-31 of Return Address Are All Zeros)



F5 PROGRAM RETURN (in 64-Bit Mode when Bits 0-31 of Updated Instruction Address Are All Zeros, Resulting PSW Bit 31 Is One (See Note), and Bits 0-31 of Return Address Are All Zeros)

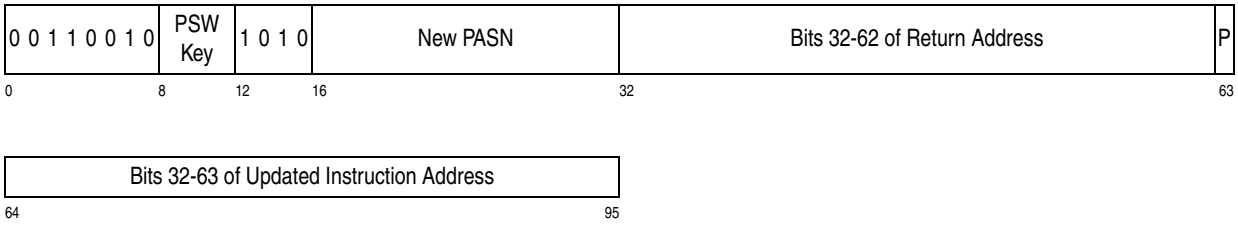
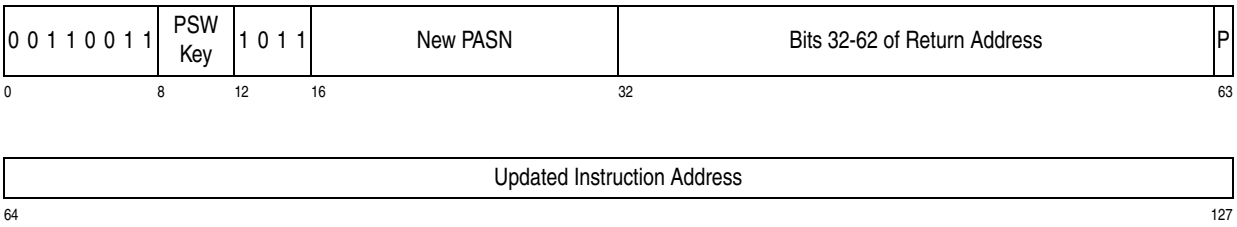
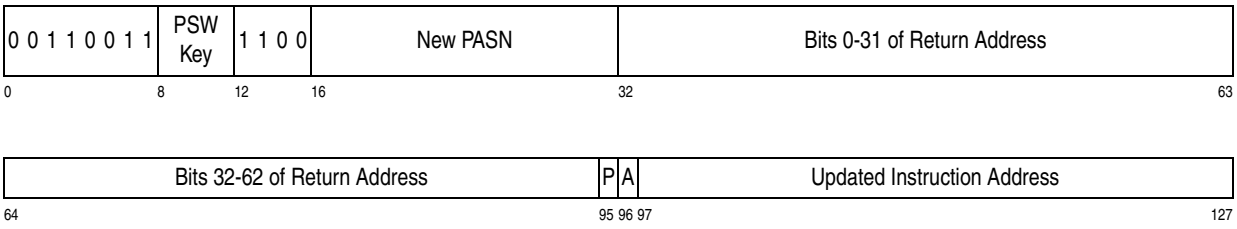


Figure 4-7. Trace Entries (Part 4 of 7)

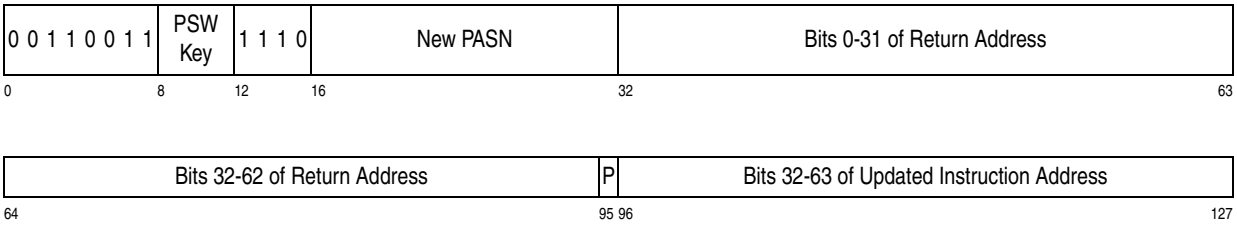
F6 PROGRAM RETURN (in 64-Bit Mode when Bits 0-31 of Updated Instruction Address Are Not All Zeros, Resulting PSW Bit 31 Is One (See Note), and Bits 0-31 of Return Address Are All Zeros)



F7 PROGRAM RETURN (in 24-Bit or 31-Bit Mode when Resulting PSW Bit 31 Is One (See Note) and Bits 0-31 of Return Address Are Not All Zeros)



F8 PROGRAM RETURN (in 64-Bit Mode when Bits 0-31 of Updated Instruction Address Are All Zeros, Resulting PSW Bit 31 Is One (See Note), and Bits 0-31 of Return Address Are Not All Zeros)



F9 PROGRAM RETURN (in 64-Bit Mode when Bits 0-31 of Updated Instruction Address Are Not All Zeros, Resulting PSW Bit 31 Is One (See Note), and Bits 0-31 of Return Address Are Not All Zeros)

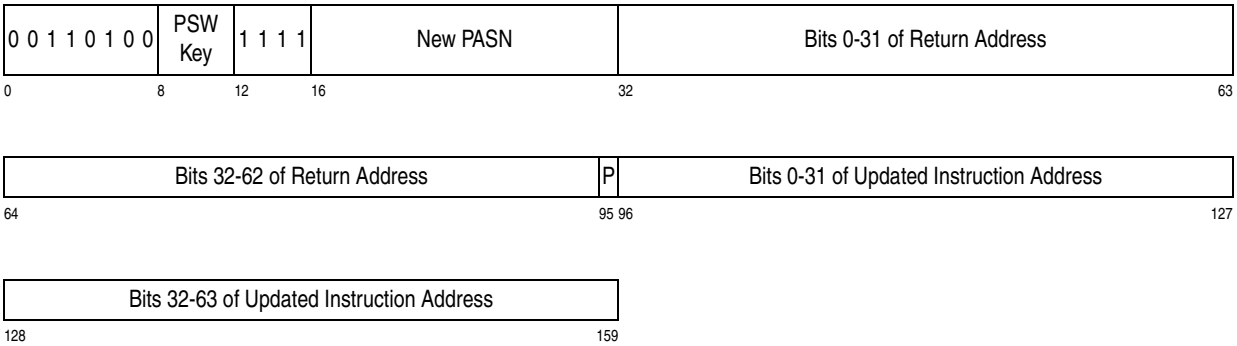
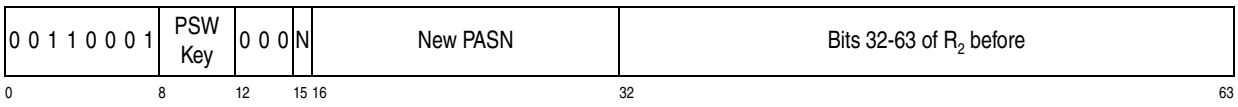
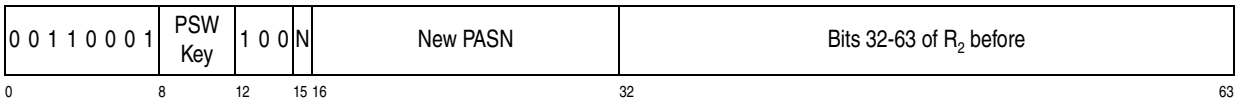


Figure 4-7. Trace Entries (Part 5 of 7)

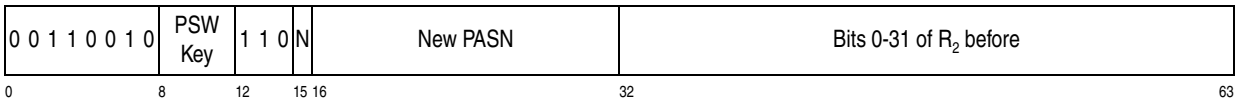
F1 PROGRAM TRANSFER (WITH INSTANCE if N Is One) (in 24-Bit or 31-Bit Mode)

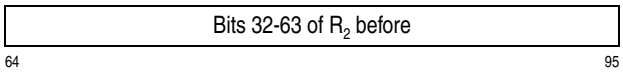


F2 PROGRAM TRANSFER (WITH INSTANCE if N Is One) (in 64-Bit Mode when Bits 0-31 of R_2 Are All Zeros)

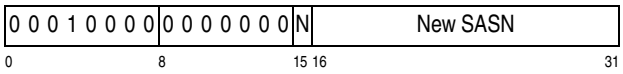


F3 PROGRAM TRANSFER (WITH INSTANCE if N Is One) (in 64-Bit Mode when Bits 0-31 of R_2 Are Not All Zeros)

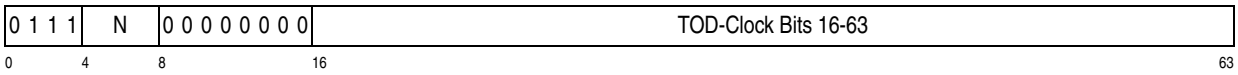


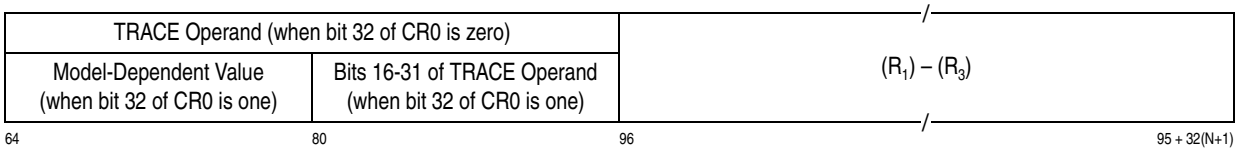


F1 SET SECONDARY ASN (WITH INSTANCE if N Is One)

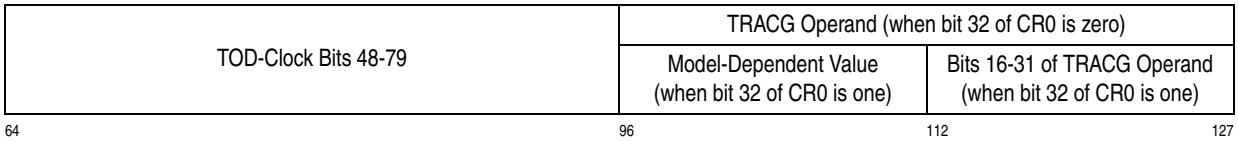


F1 TRACE (TRACE)





F2 TRACE (TRACG)



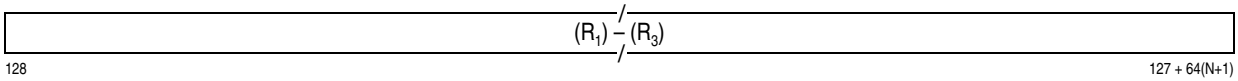


Figure 4-7. Trace Entries (Part 6 of 7)

Note: The terminology “when Resulting PSW Bit 31 Is One” is used instead of “when Resulting Mode Is 64-Bit” because, if the resulting PSW bit 32 is zero, an early specification exception will be recognized. PROGRAM RETURN can set PSW bit 31 to one and bit 32 to zero.

Figure 4-7. Trace Entries (Part 7 of 7)

The fields in the trace entries are defined as follows. The fields are described in the order in which they first appear in Figure 4-7 on page 4-16.

Branch Address: The branch address is the address of the next instruction to be executed when the branch is taken. In a branch trace entry made when the 24-bit addressing mode is in effect after branching (a format-1 entry), bit positions 8-31 contain bits 40-63 of the branch address. When the 31-bit addressing mode is in effect after branching or PSW bit 31 is one after branching and bits 0-32 of the branch address are all zeros, bit positions 1-31 of the trace entry (format 2) contain bits 33-63 of the branch address. When PSW bit 31 is one after branching and bits 0-32 of the branch address are not all zeros, bit positions 32-95 of the trace entry (format 3), contain bits 0-63 of the branch address.

In a BRANCH IN SUBSPACE GROUP trace entry made on execution in the 24-bit or 31-bit addressing mode, bit positions 33-63 of the trace entry (format 1) contain bits 33-63 of the branch address, or, in the 64-bit addressing mode, bit positions 32-95 of the trace entry (format 2) contain bits 0-63 of the branch address.

In a mode-switching-branch trace entry made on a switch from the 64-bit addressing mode to the 24-bit or 31-bit addressing mode, bit positions 33-63 of the entry (format 1) contain bits 33-63 of the branch address; or, on a switch from PSW bit 31 being off to the bit being on, bit positions 32-63 of the entry (format 2) contain bits 32-63 of the branch address if bits 0-31 of the branch address are zeros, or bits 32-95 of the entry (format 3) contain bits 0-63 of the branch address if bits 0-31 of the branch address are not all zeros.

Primary-List Bit (P) and Bits 9-31 of ALET: Bit position 8 of a BRANCH IN SUBSPACE GROUP trace entry contains bit 7 of the access-list-entry token (ALET) in the access register designated by the R_2 field of the instruction. Bit positions 9-31 of the trace entry contain bits 9-31 of the ALET.

Basic-Addressing-Mode Bit (A): Bit position 32 of a BRANCH IN SUBSPACE GROUP trace entry made on execution in the 24-bit or 31-bit addressing mode (a format-1 entry) contains the basic-addressing-mode bit that replaces bit 32 of the PSW.

Bit position 32 of a mode-switch trace entry that indicates a switch from PSW bit 31 being off to the bit being on (a format-1 entry) contains the value of PSW bit 32 that existed before the mode-switch operation.

Bit position 32 of a mode-switching-branch trace entry that indicates a switch from the 64-bit addressing mode to the 24-bit or 31-bit addressing mode (a format-1 entry) contains the value that replaces PSW bit 32.

Bit position 32 of a PROGRAM CALL trace entry made on execution in the 24-bit or 31-bit addressing mode (regardless of the resulting addressing mode) (a format-1 entry) contains the basic-addressing-mode bit, bit 32, from the current PSW.

Bit position 32 of a PROGRAM RETURN trace entry made when the resulting addressing mode is the 24-bit or 31-bit mode (a format-1, format-2, or format-3 entry) contains the basic-addressing-mode bit that replaces bit 32 of the PSW.

Bit position 64 of a PROGRAM RETURN trace entry made in the 24-bit or 31-bit addressing mode when the return address occupies only one word in the entry, (a format-1 or format-4 entry), contains the value of PSW bit 32 that existed before the PROGRAM RETURN operation. When the return address occupies two words (a format-7 entry), bit position 96 contains that value of PSW bit 32.

Updated Instruction Address: Bit positions 33-63 of a mode-switch trace entry that indicates a switch from PSW bit 31 being off to the bit being on (a format-1 entry) contains bits 33-63 of the updated instruction address in the PSW (bits 97-127 of the PSW) before that address is replaced, if it is replaced, by the mode-switch operation. Bit positions 32-63 of a mode-switch trace entry (format 2) that indicates a switch from the 64-bit addressing mode to the 24-bit or 31-bit addressing mode contains bits 32-63 of the updated instruction address in the PSW

(bits 96-127 of the PSW) before that address is replaced, if it is replaced, by the mode-switch operation, if bits 0-31 of the updated instruction address are zeros; or bit positions 32-95 of the trace entry (format 3) contain bits 0-63 of that updated instruction address (bits 64-127 of the PSW) if bits 0-31 of the address are not all zeros.

The following description of a PROGRAM RETURN trace entry applies when the return address in the entry occupies only one word in the entry. Bit positions 65-95 of the trace entry made on execution in the 24-bit or 31-bit addressing mode (a format-1 or format-4 entry) contain bits 33-63 of the updated instruction address in the PSW (bits 97-127 of the PSW) before that address is replaced from the linkage-stack state entry; or, when the execution is in the 64-bit addressing mode, bit positions 64-95 of the trace entry (format 2 or 5) contain bits 32-63 of that updated instruction address (bits 96-127 of the PSW) if bits 0-31 of the address are zeros, or bit positions 64-127 of the trace entry (format 3 or 6) contain bits 0-63 of that updated instruction address (bits 64-127 of the PSW) if bits 0-31 of the address are not all zeros. If the return address in the PROGRAM RETURN trace entry occupies two words, the updated instruction address in the entry is moved one word to the right in the entry (formats 7-9).

PSW Key: Bit positions 8-11 of a PROGRAM CALL, PROGRAM TRANSFER, or PROGRAM RETURN trace entry contain the PSW key from the current PSW.

PC Number: Bit positions 12-31 of a PROGRAM CALL trace entry of format 1-4 contain the value of the rightmost 20 bits of the second-operand address. Bit positions 64-95 of a format-5 or format-6 PROGRAM CALL trace entry, or bit positions 96-127 of a format-7 entry, contain the value of the rightmost 32 bits of the second-operand address.

Return Address: Bit positions 33-62 of a PROGRAM CALL trace entry made on execution in the 24-bit or 31-bit addressing mode (a format-1, format-3, or format-5 entry) contain bits 33-62 of the updated instruction address in the PSW (bits 97-126 of the PSW) before that address is replaced from the entry-table entry; or, when the execution is in the 64-bit addressing mode, bit positions 32-94 of the trace entry (format 2, 4, or 7) contain bits 0-62 of that updated instruction address (bits 64-126 of the PSW), or, when bits 0-31 of the address are all zeros, bit positions 32-62 of the trace entry (format 6) contain bits 32-62 of the address.

Extended-Addressing-Mode Bit (E): Bit position 15 of a PROGRAM CALL trace entry made using a 32-bit PC number (a format-5, format-6, or format-7 entry) contains the extended-addressing-mode bit that replaces bit 31 of the PSW.

Bit positions 33-62 of a PROGRAM RETURN trace entry made when the resulting addressing mode is the 24-bit or 31-bit mode (a format-1, format-2, or format-3 entry) contain bits 33-62 of the instruction address that replaces bits 64-127 of the PSW; or, when the resulting PSW bit 31 is one (which causes the addressing mode be the 64-bit mode unless the resulting PSW bit 32 is zero), bit positions 32-62 of the trace entry (formats 4-6) contain bits 32-62 of that instruction address if bits 0-31 of the address are zeros, or bit positions 32-94 of the trace entry (formats 7-9) contain bits 0-62 of that instruction address if bits 0-31 of the address are not all zeros.

Problem-State Bit (P): Bit position 63 of a PROGRAM CALL trace entry made on execution in the 24-bit or 31-bit addressing mode (regardless of the resulting mode) (a format-1, format-3, format-5, or format-6 entry), or bit position 95 of the entry (format 2, 4, or 7) made on execution in the 64-bit addressing mode, contains the problem-state bit from the current PSW.

Bit position 63 of a PROGRAM RETURN trace entry made when the resulting addressing mode is the 24-bit or 31-bit mode (a format-1, format-2, or format-3 entry) or when the resulting PSW bit 31 is one and bits 0-31 of the return address are zeros (formats 4-6) contains the problem-state bit that replaces bit 15 of the PSW. Bit position 95 of a PROGRAM RETURN trace entry made when the resulting PSW bit 31 is one and bits 0-31 of the return address are not all zeros (formats 7-9) contains that problem-state bit.

New PASN: Bit positions 16-31 a PROGRAM TRANSFER trace entry contain the new PASN (which may be zero) specified in bit positions 48-63 of general register R_1.

Bit positions 16-31 of a PROGRAM RETURN trace entry contain the new PASN that is restored from the linkage-stack state entry.

Bits 32-63 of R_2 before: Bit positions 32-63 of a PROGRAM TRANSFER trace entry made on execution in the 24-bit or 31-bit addressing mode (a format-1 entry) contain bits 32-63 of the general register designated by the R_2 field of the instruction. (Bits 32 and 33-62 of that register replace bits 32 and 97-126, respectively, of the PSW. Bit 63 of the register replaces the problem-state bit in the PSW.) When PROGRAM TRANSFER or PROGRAM TRANSFER WITH INSTANCE is executed in the 64-bit addressing mode, bit positions 32-63 of the trace entry (format 2) contain bits 32-63 of the R_2 general register if bits 0-31 of the register are zeros, or bit positions 32-95 of the trace entry (format 3) contain bits 0-63 of the register if bits 0-31 of the register are not all zeros.

New SASN: Bit positions 16-31 of a SET SECONDARY ASN trace entry contain the ASN value loaded into control register 3 by the instruction.

Number of Registers (N): Bits 4-7 of the trace entry for TRACE contain a value which is one less than the number of general registers which have been provided in the trace entry. The value of N ranges from zero, meaning the contents of one general register are provided in the trace entry, to 15, meaning the contents of all 16 general registers are provided.

Epoch Index Bits 1-7: When the multiple-epoch facility is installed in the configuration, bits 9-15 of the trace entry for TRACE (TRACG) are obtained from bit positions 1-7 of the epoch index, as would be provided by a single STORE CLOCK EXTENDED instruction executed at the time the TRACE instruction was executed (the single execution of which also provided bits 0-79 of the TOD clock)

When the multiple-epoch facility is not installed in the configuration, the following applies:

- When the TOD clock has not wrapped around to zeros, bits 9-15 of the trace entry for TRACE (TRACG) contain zeros.
- When the TOD clock has wrapped around to zeros, it is unpredictable whether bits 9-15 of the trace entry for TRACE (TRACG) contain zeros or bits 1-7 of the epoch index (as described above).

TOD-Clock Bits 16-63 or 0-79: When the store-clock-fast facility is not installed, or when the TRACE TOD-clock control in bit 32 of control register 0 is zero, bits 16-63 of the trace entry for TRACE (TRACE) are obtained from bit positions 16-63 of the TOD clock, as would be provided by a STORE CLOCK instruction executed at the time the TRACE instruction was executed. When the store-clock-fast facility is installed and the TRACE TOD-clock control in bit 32 of control register zero is one, bits 16-63 of the trace entry for TRACE (TRACE) are obtained from bit positions 16-63 of the TOD clock, as would be provided by a STORE CLOCK FAST instruction executed at the time the TRACE instruction was executed.

Bits 16-95 of the trace entry for TRACE (TRACG) are obtained from bit positions 0-79 of the TOD clock, as would be provided by a STORE CLOCK EXTENDED instruction executed at the time the TRACE instruction was executed.

TRACE Operand: When the store-clock-fast facility is not installed, or when the TRACE TOD-clock control in bit 32 of control register 0 is zero, bit positions 64-95 of the trace entry for TRACE (TRACE) and bit positions 96-127 of the trace entry for TRACE (TRACG) contain a copy of the 32 bits of the second operand of the TRACE instruction for which the entry is made.

When the store-clock-fast facility is installed and the TRACE TOD-clock control in bit 32 of control register 0 is one, the trace-operand field in the trace entry is formed as follows:

- Bit positions and 80-95 of the trace entry for TRACE (TRACE) and bit positions 112-127 of the trace entry for TRACE (TRACG) contain a copy of bits 16-31 of the second operand of the TRACE instruction for which the entry is made.
- The contents of bit positions 64-79 of the trace entry for TRACE (TRACE) and the contents of bit positions 96-111 for TRACE (TRACG) are set to a model-dependent value.

(R_1)-(R_3): The four-byte fields starting with bit 96 of the trace entry for TRACE (TRACE) contain the contents of bit positions 32-63 of the general registers whose range is specified by the R_1 and R_3 fields of the TRACE instruction. The general registers are stored in ascending order of register numbers, starting with general register R_1 and continuing up to and including general register R_3, with general register 0 following general register 15. The eight-byte fields starting with bit 128 of the trace entry for TRACE

(TRACG) similarly contain the contents of bit positions 0-63 of those registers.

Programming Note: The size of the trace entry for TRACE (TRACE) in units of words is 3 + (N + 1). The maximum size of an entry is 19 words, or 76 bytes. For TRACE (TRACG), the size in units of words is 4 + 2(N + 1), and the maximum size is 36 words, or 144 bytes.

Trace Operation

When an instruction which is subject to tracing is executed and the corresponding tracing function is turned on, a trace entry of the appropriate type and format is made, except as noted below. The real address of the trace entry is formed by appending two zero bits on the left and two on the right to the value in bit positions 2-61 of control register 12. The address in control register 12 is subsequently increased by the size of the entry created.

In the ESA/390-compatibility mode, implicit tracing (that is, branch tracing, mode tracing, and ASN tracing) is not supported.

No trace entry is stored if the incrementing of the address in control register 12 would cause a carry to be propagated into bit position 51 (that is, if the trace-entry address would be in the next 4 K-byte block). If this would be the case for the entry to be made, a trace-table exception is recognized and the operation is nullified. When PROGRAM CALL is to form both a PROGRAM CALL trace entry and a mode-switch trace entry, neither entry is stored, and a trace-table exception is recognized, if either entry would cause a carry into bit position 51. For the purpose of recognizing the trace-table exception in the case of a TRACE instruction, the maximum length of 76 (TRACE) or 144 (TRACG) bytes is used instead of the actual length.

When the CPU is in the transactional-execution mode, an instruction that would otherwise cause tracing to occur is restricted. Attempted execution of such an instruction causes the transaction to be aborted with abort code 11, and the condition code is set to 3; no trace entry is made, and no trace-table exception condition (if any) is recognized.

The storing of a trace entry is not subject to key-controlled protection (nor, since the trace-entry address is real, is it subject to DAT protection), but it is subject to low-address protection; that is, if the address of the trace entry due to be created is in the range 0-511 or 4096-4607 and bit 35 of control register 0 is one, a protection exception is recognized, and instruction execution is suppressed. If the address of a trace entry is invalid, an addressing exception is recognized, and instruction execution is suppressed.

The three exceptions associated with storing a trace entry (addressing, protection, and trace table) are collectively referred to as trace exceptions.

If a program interruption takes place for a condition which is not a trace-exception condition and for which execution of an instruction is not completed, it is unpredictable whether part or all of any trace entry due to be made for such an interrupted instruction is stored in the trace table. Thus, for a condition which would ordinarily cause nullification or suppression of instruction execution, storage locations may have been altered beginning at the location designated by control register 12 and extending up to the length of the entry that would have been created.

When PROGRAM RETURN unstacks a linkage-stack state entry that was formed by BRANCH AND STACK and ASN tracing is on, trace exceptions may be recognized, even though a trace entry is not made and no part of a trace entry is stored.

The order in which information is placed in a trace entry is unpredictable. Furthermore, as observed by other CPUs and by channel programs, the contents of a byte of a trace entry may appear to change more than once before completion of the instruction for which the entry is made.

The trace-entry address in control register 12 is updated only on completion of execution of an instruction for which a trace entry is made.

A serialization and checkpoint-synchronization function is performed before the operation begins and again after the operation is completed. However, it is unpredictable whether or not a store into a trace-table entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store. Additionally, when the store-clock-fast facility is installed and the TRACE TOD-clock control in bit 32 of control register 0 is one, it is unpredictable whether explicit tracing causes serialization to be performed.

Program-Event Recording

The purpose of PER is to assist in debugging programs. It permits the program to be alerted to the following types of events:

- Execution of a successful branch instruction. The option is provided of having an event occur only when the branch-target location is within the designated storage area.
- Fetching of an instruction from the designated storage area.
- Alteration of the contents of the designated storage area. The option is provided of having an event occur only when the storage area is within designated address spaces.
- Execution of the STORE USING REAL ADDRESS instruction.
- Execution of the TRANSACTION END instruction.

The PER zero-address-detection facility may be available on a model implementing z/Architecture and is only available in the z/Architecture architectural mode. When this facility is installed, the program may be alerted to the following additional event:

- Execution of an instruction that accesses storage using an operand address formed from a general register containing zero.

The program can selectively specify that one or more of the above types of events be recognized, except that the event for STORE USING REAL ADDRESS can be specified only along with the storage-alteration event. The information concerning a PER event is provided to the program by means of a program interruption, with the cause of the interruption being identified in the interruption code.

PER Instruction-Fetching Nullification

The PER-3 facility may be available on a model implementing z/Architecture. When the PER-3 facility is installed in the machine, it is unpredictable whether the facility is installed in the ESA/390-compatibility mode.

When this facility is installed, bit 39 of control register 9, when one, specifies that PER instruction-fetching events force nullification. Bit 39 is effective for this purpose only when bit 33 of control register 9, the instruction-fetching PER-event mask bit, is also one. When bit 33 is zero, PER instruction-fetching events are not recognized, and bit 39 has no effect. When the PER-3 facility is not installed or bit 39 is zero, PER instruction-fetching events do not force nullification. A PER instruction-fetching event that forces nullification is referred to as a PER instruction-fetching nullification event. A PER event that does not force nullification is referred to as a PER basic event.

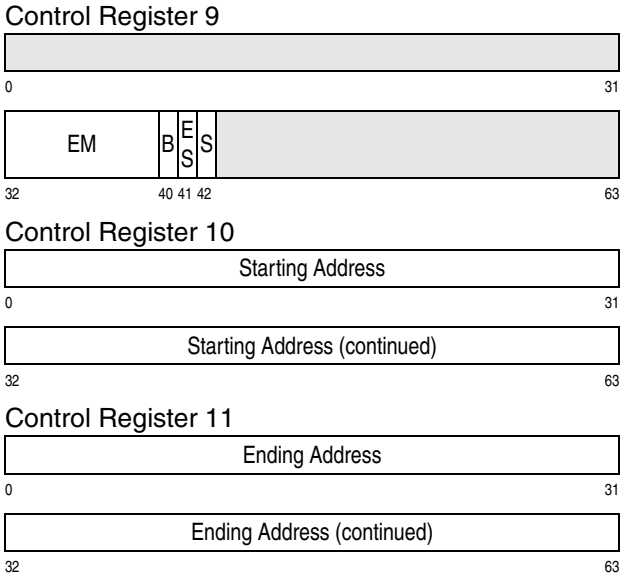
When the PER-3 facility is installed, and bit 39 is one, the interruption caused by a PER instruction-fetching event occurs before the fetched instruction is executed, the PER instruction-fetching nullification event is indicated, no other PER events and no other program interruption conditions are reported, and execution of the instruction is nullified. When the PER-3 facility is not installed, or bit 39 is zero, nullification is not forced, the PER instruction-fetching basic event is indicated, other PER events and other program interruption conditions may be concurrently reported, and the execution of the instruction may be completed, terminated, suppressed, or nullified. In the absence of other conditions, the interruption caused by the PER instruction-fetching basic event occurs after execution of the fetched instruction, or units of operation thereof, are completed.

Control-Register Allocation and Address-Space-Control Element

The information for controlling PER resides in control registers 9, 10, and 11 and the address-space-control element.

Depending on the model, when any or all of control registers 9, 10, or 11 contain nonzero values, address-compare controls may be disabled and remain disabled even if all of control registers 9-11 transition back to zeros. See “Address-Compare Controls” on page 12-1 for details.

The information in the control registers has the following format:



PER-Event Masks (EM): Bits 32-34 and 36 of control register 9 specify which types of events are recognized. When the PER zero-address-detection facility is installed, bit 37 of the PER-event masks is also used.When the transactional-execution facility is installed, bit 38 of the PER-event masks is also used. When the PER-3 facility is installed, bit 39 of the PER-event masks is also used. In the ESA/390-compatibility mode it is unpredictable whether the zero-address-detection and instruction-fetching-nullification masks. bits 37 and 39, respectively, are supported.

The bits are assigned as follows:

Bit 32: Successful-branching event
Bit 33: Instruction-fetching event
Bit 34: Storage-alteration event
Bit 36: Store-using-real-address event (bit 34 must also be one)
Bit 37: Zero-address-detection event
Bit 38: Transaction-end event
Bit 39: Instruction-fetching nullification event (bit 33 must also be one)

Bits 32-34 and bit 36, when ones, specify that the corresponding types of events be recognized. However, bit 36 is effective for this purpose only when bit 34 is also one. When bit 34 is one, the storage-alteration event is recognized. When bits 34 and 36 are ones, both the storage-alteration event and the store-using-real-address event are recognized. When a bit is zero, the corresponding type of event is not recognized. When bit 34 is zero, both the storage-alteration event and the store-using-real-address event are not recognized.

When the PER-3 facility is not installed, bit 39 is ignored. Bit 39 is effective only when bit 33 is also one. When bit 33 is one, and the PER-3 facility is installed, and bit 39 is one, the PER instruction-fetching nullification event is recognized. When bit 33 is one and bit 39 is zero (or the PER-3 facility is not installed) the PER instruction-fetching basic event is recognized. When bit 33 is zero, neither the PER instruction-fetching basic event nor the PER instruction-fetching nullification event is recognized.

When the transactional-execution facility is not installed, or when the facility is installed and bit 38 is zero, transaction-end events are not recognized. When the transactional-execution facility is installed and bit 38 is one, a transaction-end event is recognized as a result of the completion of an outermost TRANSACTION END instruction. In the ESA/390-compatibility mode, bit 38 of control register 9 is ignored.

Branch-Address Control (B): Bit 40 of control register 9 specifies, when one, that successful-branching events occur only for branches that are to a location within the designated storage area. When bit 40 is zero, successful branching events occur regardless of the branch-target address.

Event-Suppression Control (ES): When the CPU is in the transactional-execution mode at the beginning of an instruction, bit 41 of control register 9 specifies, when one, that the PER event masks in bits 32-34, 36, 37, and 39 of the register are to be ignored and assumed to contain zeros. Except as noted below, when the CPU is not in the transactional-execution mode at the beginning of the instruction, or when bit 41 of the register is zero, all PER event masks operate as defined.

When (a) an outermost TRANSACTION BEGIN instruction is executed, (b) there are no concurrent program-exception conditions, and (c) the ES control is one, any PER storage-alteration or zero-address-detection event for the TBEGIN-specified TDB and any instruction-fetching basic event are suppressed; instruction-fetching nullification events are not suppressed in this case.

In the ESA/390-compatibility mode, the event-suppression control is ignored.

Architecture Notes:

Storage-Alteration-Space Control (S): Bit 42 of control register 9 specifies, when one, that storage-alteration events occur as a result of references to the designated storage area only within designated address spaces. An address space is designated as one for which storage-alteration events occur by means of the storage-alteration-event bit in the address-space-control element that is used to translate references to the address space. Bit 42 is ignored when DAT is not in effect. When DAT is not in effect or bit 42 is zero, storage-alteration events are not restricted to occurring for only particular address spaces.

PER Starting Address: Bits 0-63 of control register 10 are the address of the beginning of the designated storage area. In the ESA/390-compatibility mode, it is unpredictable whether bit 32 of the PER starting address is treated as being zero.

PER Ending Address: Bits 0-63 of control register 11 are the address of the end of the designated storage area. In the ESA/390-compatibility mode, it is unpredictable whether bit 32 of the PER ending address is treated as being zero.

Storage-Alteration-Event Bit (S): When the storage-alteration-space control in control register 9 is one, bit 56 of the address-space control element (ASCE) specifies, when one, that the address space defined by the ASCE is one for which storage-alteration events can occur. See the sections “Control Register 1” on page 3-42, “Control Register 7” on page 3-44, and “Control Register 13” on page 3-45 for illustrations of the ASCE.

Bit 56 of the ASCE is examined when the ASCE is used to perform dynamic-address translation for a storage-operand store reference. The address-space-control element may be the PASCE, SASCE, or HASCE in control register 1, 7, or 13, respectively, or it may be obtained from an ASN-second-table entry during access-register translation. Instead of being obtained from an ASN-second-table entry in main storage, bit 56 of the ASCE may be obtained from an ASN-second-table entry in the ART-lookaside buffer (ALB). Bit 56 of the ASCE is ignored when the storage-alteration-space control is zero.

Programming Notes:

1. Models may operate at reduced performance while the CPU is enabled for PER events. In order to ensure that CPU performance is not degraded because of the operation of the PER facility, programs that do not use it should disable the CPU for PER events by setting either the PER mask in the PSW to zero or the PER-event masks in control register 9 to zero, or both. No degradation due to PER occurs when either of these fields is zero.

2. Some degradation may be experienced on some models every time control registers 9, 10, and 11 are loaded, even when the CPU is disabled for PER events (see the programming note under “Storage-Area Designation”).

3. Enabling the CPU for PER instruction-fetching nullification may be used to determine the state of the CPU before execution of any instruction within the storage area designated by control registers 10 and 11. The instruction nullified could be the first instruction after a successful branch, after LOAD PSW, or after LOAD PSW EXTENDED; or it could be the target of an execute-type instruction or the leftmost instruction in the storage area and accessed in the process of sequential execution. After recording the desired information, in order to allow the CPU to execute this instruction, either the CPU must be disabled for instruction-fetching nullification or control registers 10 and 11 must be changed to designate a different storage area. This can be contrasted to enabling for PER successful branching within the same storage area, which causes a PER event to be reported only in the first case mentioned above, but does not require special action to continue.

PER Operation

PER is under control of bit 1 of the PSW, the PER mask. When the PER mask and a particular PER-event mask bit are all ones, the CPU is enabled for the corresponding type of event; otherwise, it is disabled. However, the CPU is enabled for the store-using-real-address event only when the storage-alteration mask bit and the store-using-real-address mask bit are both one.

The CPU is enabled for the PER instruction-fetching nullification event only when the PER-3 facility is

installed, and then only when the instruction-fetching-event mask bit, the instruction-fetching-nullification-event mask bit, and the PER mask are all ones. An interruption due to a PER instruction-fetching nullification event causes the execution of the instruction causing the event to be nullified.

An interruption due to a PER basic event normally occurs after the execution of the instruction responsible for the event. The occurrence of the event does not affect the execution of the instruction, which may be completed, partially completed, terminated, suppressed, or nullified. However, recognition of a storage-alteration event causes no more than 4K bytes to be stored beginning with the byte that caused the event, and recognition of a zero-address-detection event may occur on completion of a unit of operation; recognition of these PER events may result in partial completion of an interruptible instruction.

An interruption for an instruction-fetching nullification event occurs before the instruction responsible for the event is executed, and the operation is nullified.

When a PER event is recognized while the CPU is in the transactional-execution mode, the transaction is aborted as described in "Transaction Abort Processing" on page 5-102, and bit 6 is set in the program-interruption code, indicating that the interruption occurred while the CPU was in the transactional-execution mode. For storage-alteration events that occur while the CPU is in the transactional-execution mode (except for those caused by NONTRANSACTIONAL STORE), the store is discarded even though the event results in a PER interruption.

A PER storage-alteration or zero-address-detection event is detected for the TBEGIN-specified transaction diagnostic block during the execution of the outermost TRANSACTION BEGIN instruction rather than during transaction abort processing. It is unpredictable whether a PER storage-alteration or zero-address-detection event is detected for the TBEGIN-specified TDB during the execution of inner TBEGIN instructions.

When the CPU is disabled for a particular PER event at the time it occurs, either by the PER mask in the PSW or by the masks in control register 9, the event is not recognized.

A change to the PER mask in the PSW or to the PER control fields in control registers 9, 10, and 11 affects PER starting with the execution of the immediately following instruction. Thus, if, as a result of the change, an instruction-fetching nullification event applies to the immediately following instruction, execution of that instruction will be nullified and the instruction-fetching nullification event reported.

A change to the storage-alteration-event bit in an address-space-control element in control register 1, 7, or 13 also affects PER starting with the execution of the immediately following instruction. A change to the storage-alteration-event bit in an address-space-control element that may be obtained, during access-register translation, from an ASN-second-table entry in either main storage or the ALB does not necessarily have an immediate, if any, effect on PER. However, PER is affected immediately after either PURGE ALB or COMPARE AND SWAP AND PURGE that purges the ALB is executed.

If a PER basic event occurs during the execution of an instruction which changes the CPU from being enabled to being disabled for that type of event, that PER event is recognized.

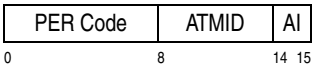
PER basic events may be recognized in a trial execution of an instruction, and subsequently the instruction, DAT-table entries, and operands may be refetched for the actual execution. If any refetched field was modified by another CPU or by a channel program between the trial execution and the actual execution, it is unpredictable whether the PER events indicated are for the trial or the actual execution.

For special-purpose instructions that are not described in this publication, the operation of PER may not be exactly as described in this section.

Identification of Cause

A program interruption for PER sets bit 8 of the interruption code to one and places identifying information in real storage locations 150-159. When the PER event is a storage-alteration event or a zero-address-detection event, information is also stored in location 161. Additional information is provided by means of the instruction address in the program old PSW and the ILC.

Locations 150-151:



PER Code: The occurrence of PER events is indicated by ones in bit positions 0-2 and 4-7. The bit position in the PER code for a particular type of event is as follows:

Bit	PER Event
0	Successful-branching
1	Instruction-fetching
2	Storage-alteration
4	Store-using-real-address
5	Zero-address-detection
6	Transaction-end
7	Instruction-fetching nullification (PER-3)

A one in bit position 2 and a zero in bit position 4 of location 150 indicate a storage-alteration event, while ones in bit positions 2 and 4 indicate a store-using-real-address event. When a program interruption occurs, more than one type of PER basic event can be concurrently indicated. However, when a storage-alteration event and a zero-address-detection event are concurrently recognized, only the storage-alteration event is indicated. Additionally, if another program-interruption condition exists, the interruption code for the program interruption may indicate both the PER basic events and the other condition.

When a program interruption occurs for a PER instruction-fetching nullification event, bits 1 and 7 are set to one in the PER code. No other PER events are concurrently indicated.

When the transactional-execution facility is installed, and a program interruption occurs for a transaction-end event, bit 6 is set to one in the PER code. If an instruction-fetching basic event coincides with the transaction-end event, bit 1 is also set to one in the PER code. No other PER events are concurrently indicated with a transaction-end event.

A zero is stored in bit position 3 of locations 150-151. When the PER zero-address-detection facility is not installed, zero is stored in bit position 5. When the transactional-execution facility is not installed, zero is stored in bit position 6. When PER-3 is not installed, zero is stored in bit position 7.

Addressing-and-Translation-Mode Identification (ATMID): During a program interruption when a PER event is indicated, bits 31, 32, 5, 16, and 17 of the PSW at the beginning of the execution of the instruction that caused the event may be stored in bit positions 8 and 10-13, respectively, of real locations 150-151. If bits 31, 32, 5, 16, and 17 are stored, then a one bit is stored in bit position 9 of locations 150-151. If bits 31, 32, 5, 16, and 17 are not stored, then zero bits are stored in bit positions 8-13 of locations 150-151.

Bits 8-13 of real locations 150-151 are named the addressing-and-translation-mode identification (ATMID). Bit 9 is named the ATMID-validity bit. When bit 9 is zero, it indicates that an invalid ATMID (all zeros) was stored.

The meanings of the bits of a valid ATMID are as follows:

Bit	Meaning
8	PSW bit 31
9	ATMID-validity bit
10	PSW bit 32
11	PSW bit 5
12	PSW bit 16
13	PSW bit 17

A valid ATMID is necessarily stored only if the PER event was caused by one of the following instructions:

- BRANCH AND SAVE AND SET MODE (BASSM)
- BRANCH AND SET AUTHORITY (BSA)
- BRANCH AND SET MODE (BSM)
- BRANCH IN SUBSPACE GROUP (BSG)
- LOAD PSW (LPSW)
- LOAD PSW EXTENDED (LPSWE)
- PROGRAM CALL (PC)
- PROGRAM RETURN (PR)
- PROGRAM TRANSFER (PT)
- PROGRAM TRANSFER WITH INSTANCE (PTI)
- RESUME PROGRAM (RP)
- SET ADDRESS SPACE CONTROL (SAC)
- SET ADDRESS SPACE CONTROL FAST (SACF)
- SET ADDRESSING MODE (SAM24, SAM31, SAM64)
- SET SYSTEM MASK (SSM)
- STORE THEN AND SYSTEM MASK (STNSM)
- STORE THEN OR SYSTEM MASK (STOSM)
- SUPERVISOR CALL (SVC)
- TRAP (TRAP2, TRAP4)

It is unpredictable whether a valid ATMID is stored if the PER event was caused by any other instruction. The value of the PER instruction-fetching-nullification-event mask bit does not affect the contents of the ATMID field.

PER ASCE Identification (AI): If the PER code contains an indication of a storage-alteration event (bit 2 is one and bit 4 is zero), or a zero-address detection event (bit 5 is one), and the event occurred when both PSW bit 5 was one and an ASCE was used to translate the reference that caused the event, bits 14 and 15 of locations 150-151 are set to identify the address-space-control element (ASCE) that was used to translate the reference that caused the event, as follows:

Bits 14-15	Meaning
00	Primary ASCE was used.
01	An AR-specified ASCE was used. The PER access ID, real location 161, can be examined to determine the ASCE used. Even when an AR-specified ASCE is used, if the contents of the AR designate the primary, secondary, or home ASCE, bits 14 and 15 may be set to 00, 10, or 11, respectively, instead of to 01.
10	Secondary ASCE was used.
11	Home ASCE was used.

The CPU may avoid setting bits 14 and 15 to 01 by recognizing that access-list-entry token (ALET) 00000000 or 00000001 hex was used or that the ALET designated, through an access-list entry, an ASN-second-table entry containing an ASCE equal to the primary ASCE, secondary ASCE, or home ASCE. When an ALE-designated ASCE was used, and the designated ASCE matches more than one of the primary ASCE, secondary ASCE, and home ASCE, it is unpredictable which of the matching ASCEs is indicated.

If the PER storage-alteration event occurred as a result of STORE HALFWORD RELATIVE LONG or STORE RELATIVE LONG, bits 14 and 15 correspond to the ASCE used to fetch the instruction: 00 when the CPU was in the primary-space mode, the secondary-space mode, or the access-register mode, and 11 when the CPU was in the home-space mode.

If either of the following is true, zeros are stored in bit positions 14 and 15 of locations 150-151.

- PSW bit 5 was zero, and an ASCE was not used to translate the reference that caused the event.
- The PER code indicates that neither a storage-alteration event nor a zero-address-detection event occurred.

The contents of the PER ASCE identification are unpredictable when any of the following is true:

- PSW bit 5 was zero, and an ASCE was used to translate the reference that caused the event.
- PSW bit 5 was one, and an ASCE was not used to translate the reference that caused the event.

Programming Note: Except for references to the enhanced-monitor counting array, an ASCE is not used to translate the reference when DAT is off or when the operand is defined to contain a real address (as is the case for LOAD USING REAL ADDRESS and STORE USING REAL ADDRESS). An ASCE is also not used to translate the reference when the operand is defined to contain a real or absolute address (as is the case for COMPARE AND REPLACE DAT TABLE ENTRY, INVALIDATE DAT TABLE ENTRY, INVALIDATE PAGE TABLE ENTRY, and PERFORM FRAME MANAGEMENT FUNCTION).

PER Address: In the z/Architecture architectural mode, the PER-address field at locations 152-159 contains the instruction address used to fetch the instruction responsible for the recognized PER event or events, as shown in Figure 4-8, below.

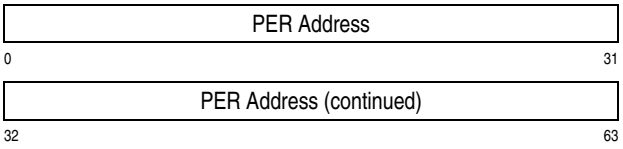


Figure 4-8. Locations 152-159 in the z/Architecture Architectural Mode

In the ESA/390-compatibility mode, the PER-address field at locations 152-155 contains bits 33-63 of the instruction address used to fetch the instruction responsible for the recognized PER event or events, as shown in Figure 4-9, below. Bit 0 of location 152 is stored as zero.

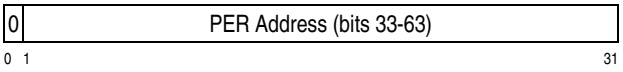


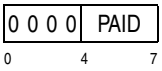
Figure 4-9. Locations 152-155 in the ESA/390-Compatibility Mode

When the instruction is the target of an execute-type instruction (EXECUTE or EXECUTE RELATIVE

LONG), the instruction address used to fetch the execute-type instruction is placed in the PER-address field.

PER Access Identification (PAID): If a storage-alteration event or zero-address-detection event is indicated in the PER code, and the PER ASCE identification (AI, bits 14-15 of locations 150-151) contains 01 binary, an indication of the address space to which the event applies is stored at location 161, as shown below.

Location 161:



The number of the access register used is stored in bit positions 4-7 of location 161, and zeros are stored in bit positions 0-3. The contents of location 161 are unpredictable if when the PER ASCE identification does not contain 01 binary.

Instruction Address: The instruction address in the program old PSW is the address of the instruction which would have been executed next, unless another program condition is also indicated, in which case the instruction address is that determined by the instruction ending due to that condition. When a PER instruction-fetching nullification event is recognized, the instruction address in the program old PSW is the address of the instruction responsible for the event. This is the same address stored in the PER address field in real storage locations 152-159.

ILC: For PER instruction nullification events, the ILC is 0. For PER basic events, the ILC indicates the length of the instruction designated by the PER address, except when a concurrent specification exception for the PSW introduced by LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, or a supervisor-call interruption sets an ILC of 0.

Programming Notes:

1. PSW bit 31 is the extended-addressing-mode bit, and PSW bit 32 is the basic-addressing-mode bit. When PSW bit 31 and 32 are both one, they specify the 64-bit addressing mode. When PSW bit 31 is zero, PSW 32 specifies the 24-bit addressing mode if the bit is zero or the 31-bit addressing mode if the bit is one. PSW bit 5 is the DAT-mode bit, and PSW bits 16 and 17 are the address-space-control bits. For the handling of instruction and logical addresses in the different translation modes, see “Translation Modes” on page 3-40.

2. A valid ATMID allows the program handling the PER event to determine the address space from which the instruction that caused the event was fetched and also to determine which translation mode applied to the storage-operand references of the instruction, if any. Each of the instructions for which a valid ATMID is necessarily stored can change one or more of PSW bits 5, 16, and 17, with the result that the values of those bits in the program old PSW that is stored because of the PER event are not necessarily the values that existed at the beginning of the execution of the instruction that caused the event. The instructions for which a valid ATMID is necessarily stored are the only instructions that can change any of PSW bits 5, 16, and 17.

3. If a storage-alteration PER event or zero-address-detection event is indicated and DAT was on when the event occurred, an indication of the address-space-control element that was used to translate the reference that caused the event is given by the PER ASCE identification, bits 14 and 15 of real locations 150-151. If bits 14 and 15 indicate that an AR-specified address-space-control element was used, the PER access identification in real location 161 can be used to determine the address space that was referenced. To determine if DAT was on, the program handling the PER event should first examine the ATMID-validity bit to determine whether a valid ATMID was stored and, if it was stored, then examine the DAT-mode bit in the ATMID. If a valid ATMID was not stored, the program should examine the DAT-mode bit in the program old PSW.

4. If a valid ATMID is stored, it also allows the program handling the PER event to determine the addressing mode (24-bit, 31-bit, or 64-bit) that existed for the instruction that caused the PER event. This knowledge of the addressing mode allows the program to determine, without any chance of error, the meaning of one bits in bit positions 0-39 of the addresses of the instruction and of the storage operands, if any, of the instruction and, thus, to determine accurately the locations of the instruction and operands. Note that the address of the instruction is not necessarily provided without error by the PER address in real locations 152-159 because that address

may be the address of an execute-type instruction, with the address of the target instruction still to be determined from the fields that specify the second-operand address of the execute-type instruction. Also note that another possible source of error is that, in the 24-bit or 31-bit addressing mode, an instruction or operand may wrap around in storage by beginning just below the 16 M-byte or 2 G-byte boundary, respectively.

5. A valid ATMID is necessarily stored for all instructions that can change the addressing-mode bits. However, the ATMID mechanism does not provide complete assurance that the instruction causing a PER event and the instruction's operands can be located accurately because LOAD CONTROL and LOAD ADDRESS SPACE PARAMETERS can change the address-space-control element that was used to fetch the instruction.

Priority of Indication

When a PER instruction-fetching nullification event is recognized and other program interruption conditions exist, only the program interruption condition with the highest priority is indicated. See "Multiple Program-Interruption Conditions" on page 6-50 for a description of the priority of program interruption conditions.

When a PER instruction-fetching nullification event is indicated, no other PER events are indicated. When a PER instruction-fetching nullification event is not indicated, then more than one PER basic event may be recognized and reported. The remainder of this section applies to these cases.

When a program interruption for PER occurs and more than one PER basic event has been recognized, all recognized PER events are concurrently indicated in the PER code. However, when either a storage-alteration or store-using-real-address event is recognized concurrently with a zero-address-detection event, only the storage alteration or store-using-real-address event is indicated. Certain other program-interruption conditions may be concurrently indicated with a PER basic event as described in "Indication of PER Events Concurrently with Other Interruption Conditions" on page 4-39.

When a zero-address-detection event is recognized for more than one storage operand, it is unpredictable which operand's ASCE identification and AR number, if applicable, are stored in locations 150-151 and 161.

In the case of an instruction-fetching basic event for SUPERVISOR CALL, the program interruption occurs immediately after the supervisor-call interruption.

If a PER basic event is recognized during the execution of an instruction which also introduces a new PSW with the type of PSW-format error which is recognized early (see "Exceptions Associated with the PSW" on page 6-9), both the specification exception and PER are indicated concurrently in the interruption code of the program interruption. If the PSW-format error is of the type which is recognized late, only PER is indicated in the interruption code. In both cases, the invalid PSW is stored as the program old PSW.

Recognition of a PER basic event does not normally affect the ending of instruction execution. However, in the following cases, execution of an interruptible instruction is not completed normally:

1. When the instruction is due to be interrupted for an asynchronous condition (I/O, external, restart, or repressible machine-check condition), a program interruption for the PER event occurs first, and the other interruptions occur subsequently (subject to the mask bits in the new PSW) in the normal priority order.
2. When the stop function is performed, a program interruption indicating the PER event occurs before the CPU enters the stopped state.
3. When any program exception is recognized, PER events recognized for that instruction execution are indicated concurrently.
4. Depending on the model, in certain situations, recognition of a PER event may appear to cause the instruction to be interrupted prematurely without concurrent indication of a program exception, without an interruption for any asynchronous condition, and without the CPU entering the stopped state. In particular, recognition of a storage-alteration event causes no more than 4K bytes to be stored beginning with the byte that caused the event, and recognition of a zero-address-detection event may occur on completion of a unit of operation.

In cases 1 and 2 above, if the only PER event that has been recognized is an instruction-fetching basic event and another unit of operation of the instruction remains to be executed, the event may be discarded,

with the result that a program interruption does not occur. Whether the event is discarded is unpredictable.

Recognition of a PER instruction-fetching nullification event causes execution of the instruction responsible for the event to be nullified.

Programming Notes:

1. In the following cases, an instruction can both cause a program interruption for a PER basic event and change the value of fields controlling an interruption for PER events. The original field values determine whether a program interruption takes place for the PER event.

   a. The instructions LOAD PSW, LOAD PSW EXTENDED, SET SYSTEM MASK, STORE THEN AND SYSTEM MASK, and SUPERVISOR CALL can cause an instruction-fetching event and disable the CPU for PER interruptions. Additionally, STORE THEN AND SYSTEM MASK can cause a storage-alteration event to be indicated. In all these cases, the program old PSW associated with the program interruption for the PER event may indicate that the CPU was disabled for PER events.

   b. An instruction-fetching event or a zero-address-detection event may be recognized during execution of a LOAD CONTROL instruction that changes the value of the PER-event masks in control register 9 or the addresses in control registers 10 and 11 controlling indication of instruction-fetching events.

   c. In the access-register mode, a storage-alteration event that is permitted by a one value of the storage-alteration-event bit in an address-space-control element in an ASN-second-table entry (designated by an access-list entry) may be caused by any store-type instruction that changes the value of the bit from one to zero.

2. When a PER interruption for a PER basic event occurs during the execution of an interruptible instruction, the ILC indicates the length of that instruction or execute-type instruction, as appropriate. When a PER interruption for a PER basic event occurs as a result of LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, or SUPERVISOR CALL, the ILC indicates the length of the instruction or of the execute-type instruction which designates the interrupted type of instruction as its target, as appropriate, unless a concurrent specification exception on LOAD PSW, LOAD PSW EXTENDED, or PROGRAM RETURN calls for an ILC of 0.

3. When a PER interruption is caused by branching, the PER address identifies the branch instruction (or execute-type instruction, as appropriate), whereas the old PSW points to the next instruction to be executed. When the interruption occurs during the execution of an interruptible instruction, the PER address and the instruction address in the old PSW are the same.

Storage-Area Designation

Two types of PER events — instruction fetching and storage alteration — always involve the designation of an area in storage. Successful-branching events may involve this designation. The storage area starts at the location designated by the starting address in control register 10 and extends up to and including the location designated by the ending address in control register 11. The area extends to the right of the starting address.

An instruction-fetching event occurs whenever the first byte of an instruction or the first byte of the target of an execute-type instruction is fetched from the designated area. A storage-alteration event occurs when a store access is made to the designated area by using an operand address that is defined to be a logical or a virtual address. However, when DAT is on and the storage-alteration-space control in control register 9 is one, a storage-alteration event occurs only when the storage area is within an address space for which the storage-alteration-event bit in the address-space-control element is one. A storage-alteration event does not occur for a store access made with an operand address defined to be a real address. When the branch-address control in control register 9 is one, a successful-branching event occurs when the first byte of the branch-target instruction is within the designated area.

The set of addresses designated for successful-branching, instruction-fetching, and storage-alteration events wraps around at address $2^{64} - 1$; that is, address 0 is considered to follow address $2^{64} - 1$. When the starting address is less than the ending

address, the area is contiguous. When the starting address is greater than the ending address, the set of locations designated includes the area from the starting address to address $2^{64} - 1$ and the area from address 0 to, and including, the ending address. When the starting address is equal to the ending address, only that one location is designated.

Address comparison for successful-branching, instruction-fetching, and storage-alteration events is always performed using 64-bit addresses. This is accomplished in the 24-bit or 31-bit addressing mode by extending the virtual, logical, or instruction address on the left with 40 or 33 zeros, respectively, before comparing it with the starting and ending addresses.

Programming Note: In some models, performance of address-range checking is assisted by means of an extension to each page-table entry in the TLB. In such an implementation, changing the contents of control registers 10 and 11 when the successful-branching, instruction-fetching, or storage-alteration-event mask is one, or setting any of these PER-event masks to one, may cause the TLB to be cleared of entries. This degradation may be experienced even when the CPU is disabled for PER events. Thus, when possible, the program should avoid loading control registers 9, 10, or 11.

PER Events

Successful Branching

When the branch-address control in control register 9 is zero, a successful-branching event occurs independent of the branch-target address. When the branch-address control is one, a successful-branching event occurs only when the first byte of the branch-target instruction is in the storage area designated by control registers 10 and 11.

Subject to the effect of the branch-address control, a successful-branching event occurs whenever one of the following instructions causes branching:

- BRANCH AND LINK (BAL, BALR)
- BRANCH AND SAVE (BAS, BASR)
- BRANCH AND SAVE AND SET MODE (BASSM)
- BRANCH AND SET AUTHORITY (BSA)
- BRANCH AND SET MODE (BSM)
- BRANCH AND STACK (BAKR)
- BRANCH IN SUBSPACE GROUP (BSG)
- BRANCH INDIRECT ON CONDITION
- BRANCH ON CONDITION (BC, BCR)
- BRANCH ON COUNT (BCT, BCTR, BCTG, BCTGR)
- BRANCH ON INDEX HIGH (BXH, BXHG)
- BRANCH ON INDEX LOW OR EQUAL (BXLE, BXLEG)
- BRANCH RELATIVE AND SAVE (BRAS)
- BRANCH RELATIVE AND SAVE LONG (BRASL)
- BRANCH RELATIVE ON CONDITION (BRC)
- BRANCH RELATIVE ON CONDITION LONG (BRCL)
- BRANCH RELATIVE ON COUNT (BRCT, BRCTG)
- BRANCH RELATIVE ON COUNT HIGH (BRCTH)
- BRANCH RELATIVE ON INDEX HIGH (BRXH, BRXHG)
- BRANCH RELATIVE ON INDEX LOW OR EQUAL (BRXLE, BRXLG)
- COMPARE AND BRANCH (CRB, CGRB)
- COMPARE AND BRANCH RELATIVE (CRJ, CGRJ)
- COMPARE IMMEDIATE AND BRANCH (CIB, CGIB)
- COMPARE IMMEDIATE AND BRANCH RELATIVE (CIJ, CGIJ)
- COMPARE LOGICAL AND BRANCH (CLRB, CLGRB)
- COMPARE LOGICAL AND BRANCH RELATIVE (CLRJ, CLGRJ)
- COMPARE LOGICAL IMMEDIATE AND BRANCH (CLIB, CLGIB)
- COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (CLIJ, CLGIJ)
- RESUME PROGRAM (RP)
- TRAP (TRAP2, TRAP4)

Subject to the effect of the branch-address control, a successful-branching event also occurs whenever one of the following instructions causes branching:

- PROGRAM CALL (PC)
- PROGRAM RETURN (PR)
- PROGRAM TRANSFER (PT)
- PROGRAM TRANSFER WITH INSTANCE (PTI)

For PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, and PROGRAM TRANSFER WITH INSTANCE, the branch-target address is considered to be the new instruction address that is placed in the PSW by the instruction.

When the guarded-storage facility is enabled, a successful-branching event is recognized as a result of a guarded-storage event caused by either of the following instructions:

- LOAD GUARDED (LGG)
- LOAD LOGICAL AND SHIFT GUARDED (LLGFSG)

When the branch-address control is one, the branch address is considered to be the contents of the guarded-storage-event handler address (GSEHA) field in the guarded-storage-event parameter list (GSEPL).

A successful-branching event causes a PER successful-branching event to be recognized if bit 32 of the PER-event masks is one and the PER mask in the PSW is one.

A PER successful-branching event is indicated by setting bit 0 of the PER code to one.

Instruction Fetching

An instruction-fetching event occurs if the first byte of the instruction is within the storage area designated by control registers 10 and 11. An instruction-fetching event also occurs if the first byte of the target of an execute-type instruction is within the designated storage area.

Instruction-Fetching Basic Event: An instruction-fetching event causes a PER instruction-fetching basic event to be recognized if the PER mask in the PSW is one and bit 33 of the PER-event masks is one and either the PER-3 facility is not installed, or bit 39 of the PER-event masks is zero.

If an instruction-fetching basic event is the only PER event recognized for an interruptible instruction that is to be interrupted because of an asynchronous condition (I/O, external, restart, or repressible machine-check condition) or the performance of the stop function, and if a unit of operation of the instruction remains to be executed, the instruction-fetching event may be discarded, and whether it is discarded is unpredictable.

The PER instruction-fetching basic event is indicated by setting bit 1 of the PER code to one and bit 7 of the PER code to zero.

Instruction-Fetching Nullification Event: An instruction-fetching event causes a PER instruction-fetching nullification event to be recognized if the PER mask in the PSW is one and bit 33 of the PER-event masks is one and the PER-3 facility is installed and bit 39 of the PER-event masks is one.

It is unpredictable whether the PER-3 facility is installed in the ESA/390-compatibility mode.

The PER instruction-fetching nullification event is indicated by setting bits 1 and 7 of the PER code to one.

Storage Alteration

A storage-alteration event occurs whenever a CPU, by using a logical or virtual address, makes a store access without an access exception to the storage area designated by control registers 10 and 11. However, when DAT is on and the storage-alteration-space control in control register 9 is one, the event occurs only if the storage-alteration-event bit is one in the address-space-control element that is used by DAT to translate the reference to the storage location.

The contents of storage are considered to have been altered whenever the CPU executes an instruction that causes all or part of an operand to be stored within the designated storage area. Alteration is considered to take place whenever storing is considered to take place for purposes of indicating protection exceptions, except that recognition does not occur for the storing of data by a channel program. (See “Recognition of Access Exceptions” on page 6-47.) Storing constitutes alteration for PER purposes even if the value stored is the same as the original value. Additionally, the contents of a TBEGIN-specified TDB are considered to have been altered by the execution of an outermost TBEGIN instruction, regardless of whether the TDB is actually stored by a transaction being aborted; it is unpredictable whether a PER storage-alteration event is detected for the first-operand location of an inner TBEGIN instruction.

Implied locations that are referred to by the CPU are not monitored. Such locations include PSW and interruption-code locations, the program-interruption transaction diagnostic block, the enhanced-monitor exception counter, and the trace entry designated by control register 12. These locations, however, are monitored when information is stored there explicitly by an instruction. Similarly, monitoring does not apply to the storing of data by a channel program. Implied

locations in the linkage stack, which are stored in by instructions that operate on the linkage stack, and enhanced-monitor-counting-array entries which are stored in by the MONITOR CALL instruction, are monitored.

The I/O instructions are considered to alter the second-operand location only when storing actually occurs.

Storage alteration does not apply to instructions whose operands are specified to have real or absolute addresses. Thus, storage alteration does not apply to COMPARE AND REPLACE DAT TABLE ENTRY,INVALIDATE DAT TABLE ENTRY (invalidation-and-clearing operation), INVALIDATE PAGE TABLE ENTRY, PAGE IN, PERFORM FRAME MANAGEMENT FUNCTION, RESET REFERENCE BIT EXTENDED, RESET REFERENCE BITS MULTIPLE, SET STORAGE KEY EXTENDED, STORE USING REAL ADDRESS, TEST BLOCK, and TEST PENDING INTERRUPTION (when the effective address is zero). Storage alteration does not apply to the store to real location 200 by STORE FACILITY LIST, nor does it apply to stores to the trace table by instructions that cause tracing to occur.

A storage-alteration event causes a PER storage-alteration event to be recognized if bit 34 of the PER-event masks is one and the PER mask in the PSW is one. Bit 36 of the PER-event masks is ignored when determining whether a PER storage-alteration event is to be recognized.

A PER storage-alteration event is indicated by setting bit 2 of the PER code to one and bit 4 of the PER code to zero.

Store Using Real Address

A store-using-real-address event occurs whenever the STORE USING REAL ADDRESS instruction is executed.

There is no relationship between the store-using-real-address event and the designated storage area.

A store-using-real-address event causes a PER store-using-real-address event to be recognized if bits 34 and 36 of the PER-event mask are ones and the PER mask in the PSW is one.

A PER store-using-real-address event is indicated by setting bits 2 and 4 of the PER code to one.

Zero-Address Detection

When the PER zero-address-detection facility is installed, a zero-address-detection event occurs whenever a CPU makes a storage access using an effective operand address formed from a general register, or subfield of a general register, containing zero. However, during execution of an RX-, RXE-, RXF-, RXY-, or VRX-format instruction, the event occurs only if the CPU makes a storage access using an effective operand address formed under one of the following conditions:

1. The base register number is zero, the index register number is nonzero, and the index register contains zero.
2. The index register number is zero, the base register number is nonzero, and the base register contains zero.
3. When both the base register number and the index register number are nonzero, it is unpredictable which one of the following conditions causes the event:

   (a) The base register contains zero.
   (b) The sum of the contents of the base register and the index register is zero.

During the execution of a VRV-format instruction, it is unpredictable which one or more of the following conditions will cause the event to occur if the CPU makes a storage access:

1. The base register number is zero, and the value of the indexed element of the second operand is zero.
2. The base register number is non-zero and the base register contains zero
3. The sum of the contents of the base register and the value of the indexed element is zero.

Except as noted below, zero-address detection for an operand address is performed whenever a fetch, store, or update reference is made to storage using the address, and zero-address detection is performed only when an operand address is used to access storage.

Except for BRANCH INDIRECT ON CONDITION, zero-address detection is not performed on the branch address of branch-type instructions. For BRANCH INDIRECT ON CONDITION, zero-address

detection is performed on the second-operand address but not on the branch address fetched from the second operand. For LOAD GUARDED and LOAD LOGICAL AND SHIFT GUARDED, zero-address detection is performed on the second-operand address of the instruction, even though these are considered to be branch-type instructions when a guarded-storage event is recognized.

Zero-address detection is also not performed on the target-instruction address of the EXECUTE instruction, and the contents of general register R_2 for the TEST BLOCK instruction.

For COMPRESSION CALL, CONVERT UTF-16 TO UTF-32, CONVERT UTF-16 TO UTF-8, CONVERT UTF-32 TO UTF-16, CONVERT UTF-32 TO UTF-8, CONVERT UTF-8 TO UTF-16, and CONVERT UTF-8 TO UTF-32, it is unpredictable whether a PER zero-address-detection event is recognized for any storage operand when the end of the first operand is reached, but the end of the second operand has not been reached.

Conditions for causing a zero-address-detection event are evaluated at the start of instruction execution. It is unpredictable if those conditions are reevaluated during instruction execution.

Except as noted below, the determination of whether a register's contents are zero is dependent upon the current addressing mode, as follows:

- In the 24- and 31-bit addressing modes, a register is considered to contain zero if bits 32-63 are all zeros.
- In the 64-bit addressing mode, a register is considered to contain zero if bits 0-63 are all zeros.

For COMPARE AND REPLACE DAT TABLE ENTRY, when bit position 59 of general register R_2 contains zero, general register R_2 is considered to contain zero if bits 0-52 are all zeros; when bit position 59 of general register R_2 contains one, general register R_2 is considered to contain zero if bits 0-51 are all zeros. For INVALIDATE DAT TABLE ENTRY, general register R_1 is considered to contain zero if bits 0-51 are all zeros, and for INVALIDATE PAGE TABLE ENTRY, general register R_1 is considered to contain zero if bits 0-52 are all zeros, regardless of the addressing mode.

Zero-Address-Detection Event: A zero-address-detection event causes a PER zero-address-detection event to be recognized if the PER mask in the PSW is one and bit 37 of the PER-event mask is one.

A PER zero-address-detection event is indicated by setting bit 5 of the PER code to one.

Programming Notes:

1. Following are examples of instructions and corresponding general register values that cause a zero-address-detection event:

   ```
   LAM     3,4,8(7)      GR7=0
   LG      1,8(0,3)      GR3=0
   L       5,87(4,0)     GR4=0
   LG      6,0(1,2)      GR2=0
   MVCL    2,4           GR2=0 or GR4=0
   MVCL    0,8           GR0=0 or GR8=0
   ```

   The examples using the MVCL instructions will cause the event only if a nonzero second-operand length is specified in general register $R_2 + 1$; see Programming Note 3.

2. A zero-address-detection event may occur for normal programming situations. In particular, an instruction that intentionally accesses the prefix storage area or a data space with an origin of zero may use a base register containing zero in the formation of a storage operand address, and thus cause a zero-address-detection event to occur. If normal execution of a program frequently uses registers containing zero in the formation of storage addresses, then the PER zero-address-detection event will be of limited use in locating programming bugs involving the unintentional use of a general register containing zero in the formation of a storage address.

3. A PER zero-address-detection event is not recognized for an operand address that is not used to access storage. For example, when the MOVE LONG instruction is executed with a second-operand length of zero specified in general register $R_2 + 1$, a PER zero-address-detection event is not recognized when the second-operand address is zero.

4. Recognition of a PER zero-address-detection event is unpredictable for instructions where it is unpredictable whether access exceptions are recognized when the operand values allow the operation to complete without storage being accessed.

Architecture Note:

Transaction End

When the CPU is in the transactional-execution mode at the beginning of an outermost TRANSACTION END instruction, a transaction-end event occurs at the completion of the instruction.

There is no relationship between the transaction-end event and the designated storage area.

A transaction-end event causes a PER transaction-end event to be recognized if bit 38 of the PER-event mask is one and the PER mask in the PSW is one.

A PER transaction-end event is indicated by setting bit 6 of the PER code to one.

A transaction-end event is not recognized when a TRANSACTION END instruction is executed and the CPU is not in the transactional-execution mode.

Indication of PER Events Concurrently with Other Interruption Conditions

When a PER instruction-fetching nullification event is reported, no other PER events and no other program interruption conditions other than transaction-abort exception (if applicable) are reported.

The following rules govern the indication of PER basic events caused by an instruction that also causes a program exception, a monitor event, a space-switch event, or a supervisor-call interruption.

1. The indication of an instruction-fetching basic event does not depend on whether the execution of the instruction was completed, terminated, suppressed, or nullified. However, special cases of suppression and nullification are as follows:

   a. When the instruction is designated by an odd instruction address in the PSW, the instruction-fetching event is not indicated.

   b. When an access exception applies to the first, second, or third halfword of the instruction designated by the PSW instruction address, and the PER-3 facility is installed, the instruction-fetching event is not indicated. However, if the PER-3 facility is not installed, it is unpredictable whether the instruction-fetching event is indicated.

   c. When either (a) an access exception applies to the first, second, or third halfword of the target location of an execute-type instruction, or (b) the target address of an EXECUTE instruction is odd, the following applies:

      1) If the PER-3 facility is installed, then the following applies:

         a) An instruction-fetching event is not indicated for the target location.

         b) It is unpredictable whether an instruction-fetching event is indicated for the execute-type instruction, including the case where the PER address range includes both the execute-type instruction and its target.

      2) If the PER-3 facility is not installed, it is unpredictable whether the instruction-fetching event is indicated for either the execute-type instruction or the target location.

2. When the operation is completed or partially completed, the event is indicated, regardless of whether any program exception, space-switch event, or monitor event is also recognized.

3. Successful branching, zero-address detection, storage alteration, and store using real address are not indicated for an operation or, in case the instruction is interruptible, for a unit of operation that is suppressed or nullified.

4. When the execution of the instruction is terminated, storage alteration or zero-address detection is indicated whenever the event has occurred. A model may indicate the event if the event would have occurred had the execution of the instruction been completed, even if altering the contents of the result field is contingent on operand values. For purposes of this definition, the occurrence of those exceptions which permit termination (addressing, protection, and data) is considered to cause termination, even if no result area is changed.

5. When LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, SET SYSTEM MASK, STORE THEN OR SYSTEM MASK, or SUPERVISOR CALL causes a PER basic condition and

at the same time introduces a new PSW with the type of PSW-format error that is recognized immediately after the PSW becomes active, the interruption code identifies both the PER basic condition and the specification exception.

6. When LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, or SUPERVISOR CALL causes a PER condition and at the same time introduces a new PSW with the type of PSW-format error that is recognized as part of the execution of the following instruction, the introduced PSW is stored as the old PSW without the following instruction being fetched or executed and without the specification exception being recognized.

7. When an outermost TRANSACTION BEGIN instruction causes a PER basic event and a concurrent program exception, the event-suppression control does not apply.

When a PER event occurs while the CPU is in the transactional-execution mode, the transaction is aborted as described in "Transaction Abort Processing" on page 5-102, and bit 6 is set in the program-interruption code.

The indication of PER basic events concurrently with other program-interruption conditions for the same instruction, as described in cases 1-4 above, is summarized in Figure 4-10 on page 4-41. Cases 5 and 6 are shown in Figure 4-11 on page 4-42.

Programming Notes:

1. The execution of the interruptible instructions COMPARE AND FORM CODEWORD, COMPARE LOGICAL LONG, COMPARE UNTIL SUBSTRING EQUAL, COMPRESSION CALL, MOVE LONG, TEST BLOCK, and UPDATE TREE can cause events for instruction fetching and zero-address detection. The execution of the interruptible instructions PERFORM FRAME MANAGEMENT FUNCTION (when the enhanced-DAT facility is installed, and the frame-size code designates a 1 M-byte frame), SET STORAGE KEY EXTENDED (when the enhanced-DAT facility is installed, and the multiple-block control is one) and TEST BLOCK can cause events for instruction-fetching. Execution of COMPRESSION CALL, MOVE LONG, and UPDATE TREE can cause events for instruction-fetching, zero-address detection and storage-alteration.

   Interruption of such an instruction may cause a PER basic event to be indicated more than once. It may be necessary, therefore, for a program to remove the redundant event indications from the PER data. The following rules govern the indication of the applicable events during execution of these instructions:

   a. The instruction-fetching basic event is indicated whenever the instruction is fetched for execution, regardless of whether it is the initial execution or a resumption, except that the event may be discarded (not indicated) if it is the only PER event to be indicated, the interruption is due to an asynchronous interruption condition or the performance of the stop function, and a unit of operation of the instruction remains to be executed.

   b. The storage-alteration event is indicated only when data has been stored in the designated storage area by the portion of the operation starting with the last initiation and ending with the last byte transferred before the interruption. No special indication is provided on premature interruptions as to whether the event will occur again upon the resumption of the operation. When the designated storage area is a single byte location, a storage-alteration event can be recognized only once in the execution of MOVE LONG or COMPRESSION CALL, but could be recognized more than once for UPDATE TREE.

2. The following is an outline of the general action a program must take to delete multiple entries for PER basic events in the PER data for an interruptible instruction so that only one entry for each complete execution of the instruction is obtained:

   a. Check to see if the PER address is equal to the instruction address in the old PSW and if the last instruction executed was interruptible.

   b. If both conditions are met, delete instruction-fetching events.

   c. If both conditions are met and the event is storage alteration, delete the event if some part of the remaining destination operand is within the designated storage area.

3. An example of the indication of a PER instruction-fetching basic event caused by either a

LOAD PSW (or LOAD PSW EXTENDED) instruction or the following instruction, in connection with an early PSW-format error or odd instruction address introduced by the LOAD PSW instruction, is shown in Figure 4-11 on page 4-42.

Notes on the Definition:

Concurrent Condition	Type of Ending	PER Basic Event					
		Branch	Instr. Fetch	Storage Alter.	STURA	TEND	Z-Addr Detect
Specification exception due to odd instruction address in the PSW:	S	–	No	–	–	–	–
Access exception fetching the instruction designated by the PSW instruction address							
Without PER-3	N or S	–	U	–	–	–	–
With PER-3	N or S	–	No	–	–	–	–
Specification exception due to the EXECUTE target address being odd:							
Without PER-3	S	–	U	–	–	–	–
With PER-3 (PER instruction-fetching-basic event detected only on the target instruction)	S	–	No	–	–	–	–
With PER-3 (PER instruction-fetching-basic event detected on the EXECUTE instruction)	S	–	U	–	–	–	–
Access exception fetching the target of an execute-type instruction:							
Without PER-3	N or S	–	U	–	–	–	–
With PER-3 (PER instruction-fetching-basic event detected only on the target instruction)	N or S	–	No	–	–	–	–
With PER-3 (PER instruction-fetching-basic event detected on the execute-type instruction)	N or S	–	U	–	–	–	–
Other nullifying exceptions	N	–	Yes	No[1]	–	–	No[1]
Other suppressing exceptions	S	–	Yes	No[1]	–	–	No[1]
All terminating exceptions	T	No	Yes	Yes[2]	–	–	U
All completing exceptions or events	C	Yes	Yes	Yes	–	–	Yes

Explanation:

– The PER event does not apply, or PER event applies but is prevented by the concurrent condition.

U It is unpredictable whether the PER event is indicated.

1 Although PER events of this type are not indicated for the current unit of operation of an interruptible instruction, PER events of this type that were recognized on completed units of operation of the interruptible instruction are indicated.

2 This event may be indicated, depending on the model, if the event has not occurred but would have been indicated if execution had been completed.

C The operation or, in the case of the interruptible instructions, the unit of operation is completed.

N The operation or, in the case of the interruptible instructions, the unit of operation is nullified.

No The PER event is not indicated.

Figure 4-10. Indication of PER Basic Events with Other Concurrent Conditions

Concurrent Condition	Type of Ending	PER Basic Event					
		Branch	Instr. Fetch	Storage Alter.	STURA	TEND	Z-Addr Detect

S The operation or, in the case of the interruptible instructions, the unit of operation is suppressed.

T The execution of the instruction is terminated.

U It is unpredictable whether the PER event is indicated.

Yes The PER event is indicated with the other program interruption condition if the event has occurred; that is, the instruction address in the PSW was replaced and the branch-address control and PER designated storage area allow the event occurrence, an attempt was made to execute an instruction whose first byte is located in the designated storage area, the contents of the designated storage area was altered, or a general register containing zero was used in the formation of an operand address used to reference storage.

Figure 4-10. Indication of PER Basic Events with Other Concurrent Conditions

LPSW at 4000 Loads a PSW		Designated Storage Area Includes		Two-Byte Instruction Is at 6000			
PSW Has Early PSW-Format Error	Instruction Address in PSW	4000	6000-6001	Interruption Code	Address in Program Old PSW	ILC	PER Address
N	6000	N	N	None	-	-	-
N	6000	N	Y	P	6002	1	6000
N	6000	Y	-	P	6000	2	4000
N	6001	N	N	S	6001+J[1]	K[1]	None
N	6001	N	Y	S[2]	6001+J[1]	K[1]	None
N	6001	Y	-	P[2 3]	6001[2 3]	2	4000
Y	6000	N	-	S	6000	0[4]	None
Y	6000	Y	-	P,S[2 3]	6000[2 3]	0[4]	4000
Y	6001	N	-	S	6001	0[4]	None
Y	6001	Y	-	P,S[2 3]	6001[2 3]	0[4]	4000

Explanation:

- Immaterial or not applicable.
1 See "ILC on Instruction-Fetching Exceptions" on page 6-8.
2 See "Indication of PER Events Concurrently with Other Interruption Conditions" on page 4-39.
3 See "Priority of Indication" on page 4-33.
4 See "Zero ILC" on page 6-7.
J Unpredictably 2, 4, or 6.
K 1, 2, or 3 depending on whether J is 2, 4, or 6, respectively.
N No.
P PER instruction-fetching basic event.
S Specification exception.
Y Yes.

Figure 4-11. Example of Instruction-Fetching PER Basic Event and Early PSW-Format Error or Odd Instruction Address

Indication of PER Events and Guarded-Storage Events

For LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED in which a guarded-storage event is recognized, any or all the following PER events are

recognized coincident with the guarded-storage event (GSE):

- An instruction-fetch basic event
- A storage-alteration event (for the GSE parameter list)
- A successful-branching event (branching to the GSE handler)
- In the absence of a PER storage-alteration event, a PER zero-address-detection event

When a GSE coincides with a PER event, the following applies:

- The PER address contains the address of LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED instruction, or, if the instruction was the target of an execute-type instruction, the PER address contains the address of the execute-type instruction.
- The instruction address of the program-old PSW is set as follows:
  - When the GSE parameter list (GSEPL) is accessible, the instruction address is set from the contents of the GSE handler-address (GSEHA) field. The GSEHA field is considered to be a branch address, thus the instruction address in the program-old PSW is subject to the action described in "Formation of the Branch Address" on page 5-13.
  - When the GSEPL is not accessible, the instruction address is set as follows:
    - If the CPU was not in the transactional-execution mode when the GSE was recognized, and the access exception results in nullification, then the instruction address points to the instruction causing the GSE (that is, the same as the PER address).
    - If the CPU was not in the transactional-execution mode when the GSE was recognized, and the access exception results in suppression or termination, then the instruction address points to the next-sequential instruction following the instruction that caused the GSE.
    - If the CPU was in the transactional-execution mode when the GSE was recognized, then the instruction address is set from the corresponding field of the transaction-abort PSW.
- The ILC indicates the length of the instruction designated by the PER address.

Programming Note: A GSE handler may adjust the contents of the second operand of the LGG or LLGFSG instruction such that a repeated execution of the instruction no longer results in a GSE being recognized. In this case, PER storage-alteration and successful-branching events are no longer applicable for the re-executed instruction. However, unless the program adjusts the PER controls, PER instruction-fetch-basic and zero-address-detection events that are recognized on the first execution of the instruction (that caused the GSE to be detected) will continue to be recognized when the instruction is re-executed, even though a GSE is not detected on the second execution.

Breaking-Event-Address Recording

When the PER-3 facility is installed, it provides the program with the address of the last instruction to cause a break in the sequential execution of the CPU. Breaking-event-address recording can be used as a debugging assist for wild-branch detection. This facility provides a 64-bit register in the CPU, called the breaking-event-address register. Each time an instruction other than TRANSACTION ABORT causes a break in the sequential instruction execution (that is, the instruction address in the PSW is replaced, rather than incremented by the length of the instruction) the address of that instruction is placed in the breaking-event-address register. Whenever a program interruption occurs, whether or not PER is indicated, the current contents of the breaking-event-address register are placed in real storage locations 272-279.

Normally, operation of the CPU is controlled by instructions in storage that are executed sequentially, one at a time, left to right in an ascending sequence of storage addresses. This is called sequential instruction execution, and, as part of this operation, the instruction address in the PSW is incremented by the length of each instruction executed. The execution of some instructions cause the contents of the

instruction address to be replaced, rather than incremented. The action to replace the instruction address is called a breaking event, and the instruction causing the action is called an execution-break instruction.

Programming Notes:

1. When the breaking-event-address register is physically installed in the machine, the full 64-bit value of the register is updated whenever a breaking event occurs. This occurs regardless of the architectural mode of the configuration

   Although the breaking-event-address register is not stored in the ESA/390 architectural mode or ESA/390-compatibility mode, a breaking event that occurs in one of these modes may be visible if the configuration switches to the z/Architecture architectural mode. If (a) the CPU does not prevent a configuration operating in the ESA/390-compatibility mode from entering the 64-bit addressing mode, (b) the program enters the 64-bit addressing mode and executes above the 2 G-byte boundary, and (c) the program switches to the z/Architecture architectural mode, then a breaking-event address above the 2 G-byte boundary may be visible following the next program interruption.

2. The addressing mode and address space used to translate the instruction address are not preserved; therefore, breaking-event-address recording is most effective for debugging code that does not contain address-space switches or addressing mode changes.

Breaking-Event-Address Register

When the PER-3 facility is installed, each CPU has a 64-bit register called the breaking-event-address register.

Each time execution of an instruction other than TRANSACTION ABORT replaces the instruction address in the PSW by any means other than sequential instruction execution, the instruction address used to fetch that instruction is placed in the breaking-event-address register. If the instruction causing the breaking event is the target of an execute-type instruction (EXECUTE or EXECUTE RELATIVE LONG), then the instruction address used to fetch the execute-type instruction is placed in the breaking-event-address register.

Each time a program interruption occurs, the contents of the breaking-event-address register are placed in real storage locations 272-279.

The contents of the breaking-event-address register are unpredictable after the SIGNAL PROCESSOR set-architecture order.

The contents of the breaking-event-address register are reset to a value of 0000000000000001 hex by initial CPU reset.

Execution-Break Instructions

A breaking event is considered to occur whenever one of the following instructions causes branching:

- BRANCH AND LINK (BAL, BALR)
- BRANCH AND SAVE (BAS, BASR)
- BRANCH AND SAVE AND SET MODE (BASSM)
- BRANCH AND SET MODE (BSM)
- BRANCH AND STACK (BAKR)
- BRANCH INDIRECT ON CONDITION
- BRANCH ON CONDITION (BC, BCR)
- BRANCH ON COUNT (BCT, BCTR, BCTG, BCTGR)
- BRANCH ON INDEX HIGH (BXH, BXHG)
- BRANCH ON INDEX LOW OR EQUAL (BXLE, BXLEG)
- BRANCH RELATIVE ON CONDITION (BRC)
- BRANCH RELATIVE ON CONDITION LONG (BRCL)
- BRANCH RELATIVE ON COUNT (BRCT, BRCTG)
- BRANCH RELATIVE ON COUNT HIGH (BRCTH)
- BRANCH RELATIVE ON INDEX HIGH (BRXH, BRXHG)
- BRANCH RELATIVE ON INDEX LOW OR EQUAL (BRXLE, BRXLG)
- COMPARE AND BRANCH (CRB, CGRB)
- COMPARE AND BRANCH RELATIVE (CRJ, CGRJ)
- COMPARE IMMEDIATE AND BRANCH (CIB, CGIB)
- COMPARE IMMEDIATE AND BRANCH RELATIVE (CIJ, CGIJ)
- COMPARE LOGICAL AND BRANCH (CLRB, CLGRB)
- COMPARE LOGICAL AND BRANCH RELATIVE (CLRJ, CLGRJ)
- COMPARE LOGICAL IMMEDIATE AND BRANCH (CLIB, CLGIB)

- COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (CLIJ, CLGIJ)

A breaking event is also considered to occur whenever one of the following instructions completes:

- BRANCH AND SET AUTHORITY (BSA)
- BRANCH IN SUBSPACE GROUP (BSG)
- BRANCH RELATIVE AND SAVE (BRAS)
- BRANCH RELATIVE AND SAVE LONG (BRASL)
- LOAD PSW (LPSW)
- LOAD PSW EXTENDED (LPSWE)
- PROGRAM CALL (PC)
- PROGRAM RETURN (PR)
- PROGRAM TRANSFER (PT)
- PROGRAM TRANSFER WITH INSTANCE (PTI)
- RESUME PROGRAM (RP)
- TRAP (TRAP2, TRAP4)

A breaking event is also considered to occur whenever a guarded-storage event is detected at the completion of the following instructions:

- LOAD GUARDED (LGG)
- LOAD LOGICAL AND SHIFT GUARDED (LLGFSG)

A breaking event is not considered to occur as a result of a transaction being aborted (either implicitly or as a result of the TRANSACTION ABORT instruction).

Timing

The timing facilities include three facilities for measuring time: the TOD clock, the clock comparator, and the CPU timer. A TOD programmable register is associated with the TOD clock. When the multiple-epoch facility is installed, the TOD-clock epoch index is also provided.

In a multiprocessing configuration, a single TOD clock is shared by all CPUs. Each CPU has its own clock comparator, CPU timer, and TOD programmable register.

Time-of-Day Clock and Epoch Index

The time-of-day (TOD) clock provides a high-resolution measure of real time suitable for the indication of date and time of day. The duration of the TOD clock, beginning with a value of zero and continuing until it wraps around to zero, is approximately 143 years; this duration is referred to as an *epoch*. A single TOD clock is shared by all CPUs in the configuration.

Format

The TOD clock is a 104-bit register.

For certain instructions, the TOD clock is considered to be extended to the left by an 8-bit *epoch index*. In a multiprocessing configuration, a single epoch index is shared by all CPUs. When the multiple-epoch facility is not installed in the configuration, the epoch index contains zeros when the TOD clock has not wrapped around to zero; when the TOD clock has wrapped around to zeros, it is unpredictable whether or not the epoch index contains zeros. When the multiple-epoch facility is installed in the configuration, the epoch index extends the capacity of the monotonic sequence of clock values to approximately 36,534 years.

The TOD clock nominally is incremented by adding a one in bit position 51 every microsecond. In models having a higher or lower resolution, a different bit position is incremented at such a frequency that the rate of advancing the clock is the same as if a one were added in bit position 51 every microsecond. The resolution of the TOD clock is such that the incrementing rate is comparable to the instruction-execution rate of the model.

The stepping value of TOD-clock bit position 63, if implemented, is 2^{-12} microseconds, or approximately 244 picoseconds. This value is called a clock unit.

Figure 4-12 illustrates the format of the TOD clock and TOD-programmable field, and their relationship to the operands of the instructions that inspect or set them. Additionally, the PTFF instruction may manipulate the TOD clock and epoch index, by means of fields in the instruction's parameter block that have the same resolution as the TOD clock (see "PER-

FORM TIMING FACILITY FUNCTION" on page 10-81).

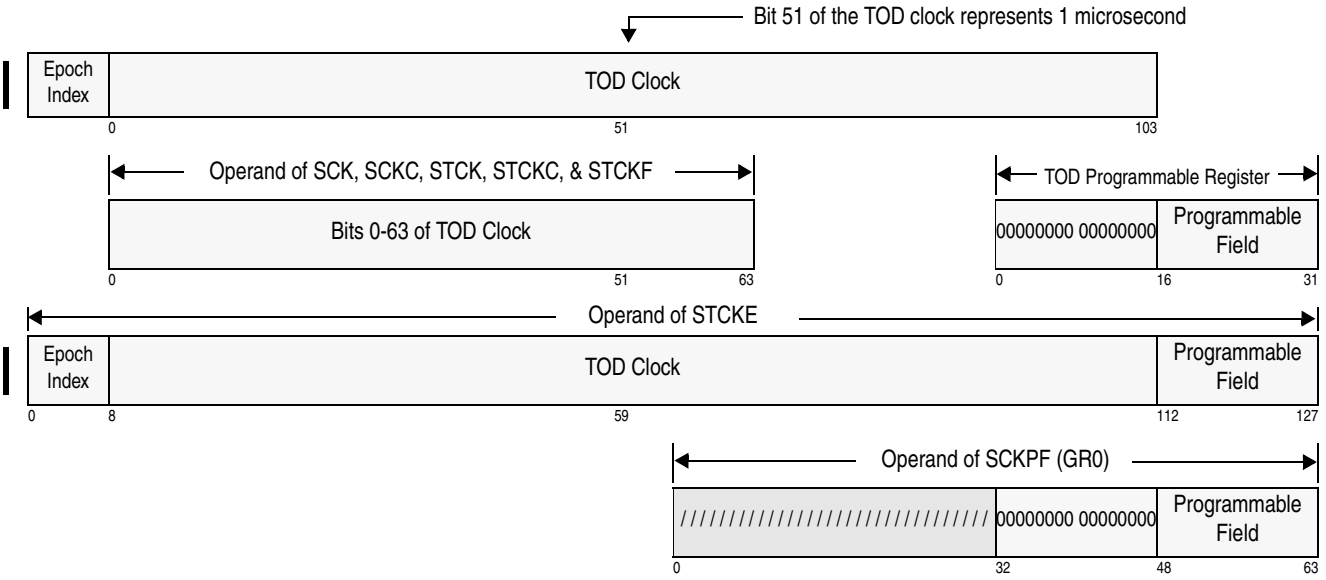


Figure 4-12. Format of the TOD Clock, Epoch Index, and TOD-Programmable Field as Set and Inspected by Various Instructions

When incrementing of the TOD clock causes a carry to be propagated out of bit position 0, the program is not alerted, and no interruption condition is generated as a result of the overflow. The handling of such a carry out of bit position 0 is as follows:

- The value of the TOD clock wraps around to zero.
- When the multiple-epoch facility is not installed in the configuration, it is unpredictable whether (a) the carry is ignored, and counting continues from zero, or (b) the carry causes the epoch index to be incremented by 1. Additionally, in case (b), above, the epoch index may revert to a value of zeros in future observations.
- When the multiple-epoch facility is installed in the configuration, the carry causes the epoch index to be incremented by 1. Any carry propagated out of the leftmost bit position of the epoch index is ignored.

The operation of the clock is not affected by any normal activity or event in the system. Incrementing of the clock does not depend on whether the wait-state bit of the PSW is one or whether the CPU is in the operating, load, stopped, or check-stop state. Its operation is not affected by CPU, initial-CPU, or clear resets or by initial program loading. Operation of the clock is also not affected by the setting of the rate control or by an initial-machine-loading operation. Depending on the model and the configuration, the TOD clock may or may not be powered independent of the CPU.

States

The following states are distinguished for the TOD clock: set, not set, stopped, error, and not operational. The state determines the condition code set by execution of STORE CLOCK, STORE CLOCK EXTENDED, and STORE CLOCK FAST. The clock is incremented, and is said to be running, when it is in either the set state or the not-set state.

Not-Set State: When the power for the clock is turned on, the clock is set to zero, and the clock enters the not-set state. The clock is incremented when in the not-set state. When the TOD-clock-steering facility is installed, the TOD clock is never reported to be in the not-set state, as the TOD clock is placed in the set state as part of the initial-machine-loading (IML) process.

When the clock is in the not-set state, execution of STORE CLOCK, STORE CLOCK EXTENDED, or STORE CLOCK FAST causes condition code 1 to be set and the current value of the running clock to be stored.

Stopped State: The clock enters the stopped state when SET CLOCK is executed and the execution results in the clock being set. This occurs when SET CLOCK is executed without encountering any exceptions and either any manual TOD-clock control in the configuration is set to the enable-set position or the TOD-clock-control-override control, bit 42 of control register 14, is one. The clock can be placed in the stopped state from the set, not-set, and error states. The clock is not incremented while in the stopped state.

When the clock is in the stopped state, execution of STORE CLOCK, STORE CLOCK EXTENDED, or STORE CLOCK FAST causes condition code 3 to be set and the value of the stopped clock to be stored.

Set State: The clock enters the set state only from the stopped state. The change of state is under control of the TOD-clock-sync-control bit, bit 34 of control register 0, of the CPU which most recently caused the clock to enter the stopped state. If the bit is zero, the clock enters the set state at the completion of execution of SET CLOCK. If the bit is one, the clock remains in the stopped state until the bit is set to zero on that CPU or until another CPU executes a SET CLOCK instruction affecting the clock. If an external time reference (ETR) is installed, a signal from the ETR may be used to set the set state from the stopped state. When the system is not in the interpretive-execution mode, and an external time reference (ETR) is not attached to the configuration, the TOD-clock-sync control is treated as being zero, regardless of its actual value; in this case, the clock enters the set state and resumes incrementing upon completion of the instruction.

Programming Note: The STP facility does not provide a signal that will cause the clock to transition from the stopped state to the set state. If SET CLOCK is executed when the configuration is in STP-timing mode and bit 34 of control register 0 is one, the clock remains in the stopped state until the bit is set to zero on that CPU or until another CPU executes a SET CLOCK instruction affecting the clock.

Incrementing of the clock begins with the first stepping pulse after the clock enters the set state.

In the absence of an ETR, depending on the model, the clock may enter the set state from the stopped state without any programming action.

When the clock is in the set state, execution of STORE CLOCK, STORE CLOCK EXTENDED, or STORE CLOCK FAST causes condition code 0 to be set and the current value of the running clock to be stored.

Error State: The clock enters the error state when a malfunction is detected that is likely to have affected the validity of the clock value. It depends on the model whether the clock can be placed in this state. A timing-facility-damage machine-check-interruption condition is generated on each CPU in the configuration whenever the clock enters the error state. When the TOD-clock-steering facility is installed, the TOD clock is never reported to be in the error state. Errors in the TOD clock cause a system check stop.

When STORE CLOCK, STORE CLOCK EXTENDED, or STORE CLOCK FAST is executed and the clock is in the error state, condition code 2 is set, and the value stored is zero.

Not-Operational State: The clock is in the not-operational state when its power is off or when it is disabled for maintenance. It depends on the model whether the clock can be placed in this state. Whenever the clock enters the not-operational state, a timing-facility-damage machine-check-interruption condition is generated on each CPU in the configuration. When the TOD-clock-steering facility is installed, the TOD clock is never reported to be in the not-operational state.

When the clock is in the not-operational state, execution of STORE CLOCK, STORE CLOCK EXTENDED, or STORE CLOCK FAST causes condition code 3 to be set, and zero is stored.

Programming Note: When the TOD-clock-steering facility is installed, the TOD clock has only two states, the set state and the stopped state. Assuming proper operation by the operating system, problem programs are never dispatched while the TOD clock is in the stopped state. Thus, as observed by the problem program, the TOD clock is always in the set state and there is no need to test the condition code after issuing STORE CLOCK, STORE CLOCK EXTENDED, or STORE CLOCK FAST.

Changes in Clock State

When the TOD-clock-steering facility is not installed, and the TOD clock changes value because of the

execution of SET CLOCK or changes state, interruption conditions pending for the clock comparator and CPU timer may or may not be recognized for up to 1.048576 seconds (2^{20} microseconds) after the change.

When the TOD-clock-steering facility is installed and SET CLOCK is issued, interruption conditions for the clock comparator and CPU timer may or may not be recognized while the physical clock is in the stopped state. After the physical clock enters the set state, interruption conditions for the clock comparator are not necessarily recognized until one of the following instructions is issued: SET CLOCK COMPARATOR (SCKC), STORE CLOCK (STCK), or STORE CLOCK EXTENDED (STCKE). After the physical clock enters the set state, interruption conditions for the CPU timer are not necessarily recognized until SET CPU TIMER (SPT) is issued. If a CPU is in the wait state when the physical clock is set or changes state, the CPU does not necessarily recognize interruption conditions for the clock comparator or CPU timer until after it leaves the wait state and the appropriate aforementioned instruction is executed.

When the TOD-clock-steering facility is installed and the logical TOD clock is changed by PTFF-ATO, PTFF-STO, or PTFF-STOE, interruption conditions for the clock comparator are not necessarily recognized until one of the following instructions is issued: SET CLOCK COMPARATOR (SCKC), STORE CLOCK (STCK), or STORE CLOCK EXTENDED (STCKE). If a CPU is in the wait state when the logical TOD clock is changed, the CPU may or may not recognize interruption conditions for the clock comparator until after it leaves the wait state and one of the aforementioned instructions is executed. The CPU timer and CPU-timer interruptions are not affected by PTFF-ATO, PTFF-STO and PTFF-STOE.

The results of channel-subsystem-monitoring-facility operations may be unpredictable as a result of changes to the TOD clock.

Setting and Inspecting the Clock

When neither the TOD-clock-steering facility nor the multiple-epoch facility is installed, the clock can be set to a specified value by execution of SET CLOCK if the manual TOD-clock control of any CPU in the configuration is in the enable-set position or the TOD-clock-control-override control, bit 42 of control register 14, is one. SET CLOCK sets bits of the clock with the contents of corresponding bit positions of a doubleword operand in storage. When either the TOD-clock-steering facility or the multiple-epoch facility is installed, the use of the SET CLOCK instruction is discouraged; rather, the PERFORM TIMING FACILITY FUNCTION instruction should be used to set the clock (see “TOD-Clock Steering” on page 4-54 for additional details).

Setting the clock replaces the values in all bit positions from bit position 0 through the rightmost position that is incremented when the clock is running. However, on some models, the rightmost bits starting at or to the right of bit 52 of the specified value are ignored, and zeros are placed in the corresponding positions of the clock. Zeros are also placed in positions to the right of bit position 63 of the clock.

The TOD clock can be inspected by executing STORE CLOCK or STORE CLOCK FAST, which causes bits 0-63 of the clock to be stored in an eight-byte operand in storage, or by executing STORE CLOCK EXTENDED, which causes bits 0-103 of the clock to be stored in bytes 1-13 of a 16-byte operand in storage. STORE CLOCK EXTENDED also stores the concurrently-obtained epoch index in byte 0 of its storage operand, and it obtains the TOD programmable field from bit positions 16-31 of the TOD programmable register and stores it in byte positions 14 and 15 of the storage operand. Figure 4-12 on page 4-47 shows the format of the operands stored by STORE CLOCK, STORE CLOCK EXTENDED, and STORE CLOCK FAST.

The following discussion of the results of STORE CLOCK, STORE CLOCK EXTENDED, and STORE CLOCK FAST assumes that the TOD clock is running and in the set state; TOD-clock values or portions thereof are considered to be unsigned binary integers.

- The values stored by any of the following sequence of instructions always correctly imply the sequence of execution of these instructions by one or more CPUs for all cases where the sequence can be discovered by the program:
  - Two executions of STORE CLOCK (assuming no wrap around of the TOD clock occurs between executions)
  - Two executions of STORE CLOCK EXTENDED (assuming no wrap around of the TOD clock occurs between executions when the multiple-epoch facility is not

installed in the configuration, and assuming no wrap around of the epoch index occurs when the multiple-epoch facility is installed in the configuration)

– An execution of STORE CLOCK followed by an execution of STORE CLOCK EXTENDED (assuming no wrap around of the TOD clock occurs between executions)

– An execution of STORE CLOCK EXTENDED followed by an execution of STORE CLOCK (assuming no wrap around of the TOD clock occurs between executions)

To ensure that unique values are obtained, nonzero values may be stored in positions to the right of the rightmost incremented bit position. When stored by STORE CLOCK EXTENDED, the value in bit positions 64-103 of the clock (bit positions 72-111 of the storage operand) is always nonzero; this ensures that values stored by STORE CLOCK EXTENDED are always unique when compared with values stored by STORE CLOCK, extended on the right with zeros.

For the purpose of establishing uniqueness and sequence of occurrence of the results of STORE CLOCK and STORE CLOCK EXTENDED, the 64-bit value provided by STORE CLOCK may be considered to be extended to a full 104-bit TOD value by appending 40 zeros on the right. As neither STORE CLOCK nor STORE CLOCK FAST store an epoch index, meaningful comparison between the results of these instruction with those of STORE CLOCK EXTENDED is possible only when the instructions are executed in the same epoch.

- Regardless of which instruction is executed first, STORE CLOCK and STORE CLOCK FAST executed on either the same or different CPUs do not necessarily return different values of the clock. The result of the second execution may be less than, equal to, or greater than that of the first execution.

- Two executions of STORE CLOCK FAST do not necessarily return different values of the clock. When executed on the same CPU, and in the absence of a carry out of bit position 0 of the TOD clock, the result of the second execution may be equal to or greater than that of the first execution. When executed on different CPUs in the same configuration, the result of the second execution may be less than, equal to, or greater than that of the first execution.

- Regardless of which instruction is executed first, STORE CLOCK EXTENDED and STORE CLOCK FAST executed on either the same or different CPUs do not necessarily return different values for the common TOD-clock bit positions stored by the instructions (that is, the entire 64-bit result of STORE CLOCK FAST as compared with bit positions 8-71 of the result of STORE CLOCK EXTENDED). When executed on the same CPU, and in the absence of a carry out of bit position 0 of the TOD clock, the result of the second execution may be either equal to or greater than that of the first execution. When executed on different CPUs in the same configuration, the result of the second execution may be less than, equal to, or greater than that of the first execution.

Figure 4-13 summarizes the interdependencies of the results of STCK, STCKE, and STCKF described above, assuming no wrap-around of the TOD clock occurs between executions.

First Execution by:	Result of Second Execution by:		
	STCK	**STCKE**	**STCKF**
STCK	A: >[3]	A: >[1,3]	A: <=>
STCKE	A: >[1,3]	A: >	S: =>[2,3] D: <=>[2]
STCKF	A: <=>	S: =>[2,3] D: <=>[2]	S: =>[3] D: <=>

Explanation:

[1]	STCK results extended on the right with 40 binary zeros.
[2]	STCKF results compared with bits 8-71 of STCKE results.
<=>	Result of the second execution may be less than, equal to, or greater than that of the first execution.
>	Result the second execution is greater than that of the first execution.
=>	Result of the second execution is equal to or greater than that of the first execution.
A:	First and second execution are on any CPU.
D:	First and second execution on different CPUs.
S:	First and second execution on the same CPU.

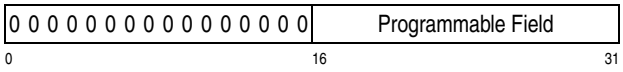
Figure 4-13. Interdependencies of the Results of STCK, STCKE, and STCKF

In a configuration where more than one CPU accesses the clock, SET CLOCK is interlocked such that the entire contents appear to be updated concurrently; that is, if SET CLOCK instructions are executed simultaneously by two CPUs, the final result is

either one or the other value. If SET CLOCK is executed by one CPU and STORE CLOCK, STORE CLOCK EXTENDED, or STORE CLOCK FAST by the other, the result obtained by STORE CLOCK, STORE CLOCK EXTENDED, or STORE CLOCK FAST is either the entire old value or the entire new value. When SET CLOCK is executed by one CPU, a STORE CLOCK, STORE CLOCK EXTENDED, or STORE CLOCK FAST instruction executed by another CPU may find the clock in the stopped state even when the TOD-clock-sync-control bit, bit 34 of control register 0, of each CPU is zero. Since the clock enters the set state before incrementing, the first STORE CLOCK, STORE CLOCK EXTENDED, or STORE CLOCK FAST instruction executed after the clock enters the set state may still find the original value introduced by SET CLOCK.

TOD Programmable Register

Each CPU has a TOD programmable register. Bits 16-31 of the register contain the programmable field that is appended on the right to the TOD-clock value by STORE CLOCK EXTENDED. The register has the following format:



The register is loaded by SET CLOCK PROGRAMMABLE FIELD. The contents of the register are reset to a value of all zeros by initial CPU reset.

Programming Notes:

1. Bit position 31 of the clock is incremented every 1.048576 seconds; for some applications, reference to the leftmost 32 bits of the clock may provide sufficient resolution.

2. Communication between systems is facilitated by establishing a standard time origin that is the calendar date and time to which a clock value of zero corresponds. January 1, 1900, 0 a.m. Coordinated Universal Time (UTC) is recommended as this origin, and it is said to begin the standard epoch for the clock. This is also the epoch used when the TOD clock is synchronized to the external time reference (ETR), and, for this reason, the epoch is sometimes referred to as ETR time. The former term, Greenwich Mean Time (GMT), is now obsolete and has been replaced with the more precise UTC.

3. Historically, one of the most important uses of standard time has been for navigation. Prior to 1972, standard time, then called GMT, was defined to have a variable-length second and was synchronized to within 100 milliseconds of the rotational position of the earth. Synchronization was accomplished by occasional changes in the length of the second, typically in parts per billion, and also by occasional insertion and deletion of small increments of time, typically 50 or 100 milliseconds. Beginning in 1972, a new standard time scale, called UTC, was defined to have a fixed-length second and be kept synchronized to within 900 milliseconds of the rotational position of the earth by means of occasional adjustments of exactly one second called a leap second. The change from GMT to UTC occurred between the last second of the day on December 31, 1971 and the first second of the day on January 1, 1972 and included insertion of 107.758 milliseconds in the standard time scale to make UTC exactly 10 seconds behind International Atomic Time (TAI). For reasons of simplicity in this document, the term UTC is sometimes extrapolated backward before 1972 by assuming no time adjustments in that time scale before 1972. For the same reasons, conversion between ETR time and UTC does not take into consideration the time adjustments prior to 1972, and, thus, ETR time differs from TAI by a fixed amount of 10 seconds. Coincident with the occurrence of the 26th leap second, UTC will be behind TAI by 37 seconds.

4. A program using the clock value as a time-of-day and calendar indication must be consistent with the programming support under which the program is to be executed. If the programming support uses the standard epoch, bit 0 of the clock remains one through the years 1972-2041. Bit 0 turned on at 11:56:53.685248 (UTC) May 11, 1971.

5. In converting to or from the current date or time, the programming support must take into account that leap seconds have been inserted or deleted because of time-correction standards. When the TOD clock has been set correctly to a time within the standard epoch, the sum of the accumulated leap seconds must be subtracted from the clock time to determine UTC time.

6. Because of the limited accuracy of manually setting the clock value, the rightmost bit positions of the clock, expressing fractions of a second, are

normally not valid as indications of the time of day. However, they permit elapsed-time measurements of high resolution.

7. The following chart shows the time interval between instants at which various bit positions of the TOD clock are stepped. This time value may also be considered as the weighted time value that the bit, when one, represents.

TOD-Clock Bit	Stepping Interval			
	Days	Hours	Min.	Seconds
51				0.000 001
47				0.000 016
43				0.000 256
39				0.004 096
35				0.065 536
31				1.048 576
27				16.777 216
23			4	28.435 456
19		1	11	34.967 296
15		19	5	19.476 736
11	12	17	25	11.627 776
7	203	14	43	6.044 416
3	3257	19	29	36.710 656

8. The following chart shows the TOD clock setting for 00:00:00 (0 am), UTC time, for several dates: January 1, 1900, January 1, 1972, and for that instant in time just after each of the 27 leap seconds that will have occurred through January, 2017. Each of these leap seconds is inserted in the UTC time scale beginning at 23:59:60 UTC of the day previous to the one listed and ending at 00:00:00 UTC of the day listed.

Year	Month	Day	Leap Sec.	Clock Setting (Hex)
1900	1	1		0000 0000 0000 0000
1972	1	1		8126 D60E 4600 0000
1972	7	1	1	820B A981 1E24 0000
1973	1	1	2	82F3 00AE E248 0000
1974	1	1	3	84BD E971 146C 0000
1975	1	1	4	8688 D233 4690 0000
1976	1	1	5	8853 BAF5 78B4 0000
1977	1	1	6	8A1F E595 20D8 0000
1978	1	1	7	8BEA CE57 52FC 0000
1979	1	1	8	8DB5 B719 8520 0000
1980	1	1	9	8F80 9FDB B744 0000
1981	7	1	10	9230 5C0F CD68 0000
1982	7	1	11	93FB 44D1 FF8C 0000
1983	7	1	12	95C6 2D94 31B0 0000
1985	7	1	13	995D 40F5 17D4 0000
1988	1	1	14	9DDA 69A5 57F8 0000
1990	1	1	15	A171 7D06 3E1C 0000
1991	1	1	16	A33C 65C8 7040 0000
1992	7	1	17	A5EC 21FC 8664 0000
1993	7	1	18	A7B7 0ABE B888 0000
1994	7	1	19	A981 F380 EAAC 0000
1996	1	1	20	AC34 336F ECD0 0000
1997	7	1	21	AEE3 EFA4 02F4 0000
1999	1	1	22	B196 2F93 0518 0000
2006	1	1	23	BE25 1097 973C 0000
2009	1	1	24	C387 0CB9 BB60 0000
2012	7	1	25	C9CC 9A70 4D84 0000
2015	7	1	26	CF2D 54B4 FBA8 0000
2017	1	1	27	D1E0 D681 73CC 0000

9. The following chart shows various time intervals in clock units expressed in hexadecimal notation.

Interval	Clock Units (Hex)
1 microsecond	1000
1 millisecond	3E 8000
1 second	F424 0000
1 minute	39 3870 0000
1 hour	D69 3A40 0000
1 day	1 41DD 7600 0000
365 days	1CA E8C1 3E00 0000
366 days	1CC 2A9E B400 0000
1,461 days*	72C E4E2 6E00 0000

Explanation:

* Number of days in four years, including a leap year. Note that the year 1900 was not a leap year. Thus, the four-year span starting in 1900 has only 1,460 days.

10. The charts in notes 7-9 are useful when examining the value stored by STORE CLOCK. Similar charts for use when examining the value stored by STORE CLOCK EXTENDED are in programming notes at the end of the definition of that instruction.

11. In a multiprocessing configuration, after the TOD clock is set and begins running, the program should delay activity for 2^{20} microseconds (1.048576 seconds) to ensure that the CPU-timer and clock-comparator interruption conditions are recognized by the CPU.

12. Due to the sequencing rules for the results of STORE CLOCK and STORE CLOCK EXTENDED, the execution of STORE CLOCK may be considerably slower than that of STORE CLOCK EXTENDED and STORE CLOCK FAST. Depending on the model, the relative slowness of STORE CLOCK (as compared with STORE CLOCK EXTENDED and STORE CLOCK FAST) is particularly noticeable when multiple STORE CLOCK instructions are executed within a short period of time.

13. Uniqueness of TOD-clock values can be extended to apply to processors in separate configurations by including a configuration identification in the TOD programmable field.

14. Using the standard epoch, programs that issue STORE CLOCK or STORE CLOCK FAST will observe the TOD clock wrap around to zero on September 17, 2042, at 23:53:57.370496 TAI. If the program cannot tolerate such a wrap around, it should be adapted to use STORE CLOCK EXTENDED on a processor supporting the multiple-epoch facility.

15. The results of STORE CLOCK FAST are intended to be approximately equivalent to either (a) the results of STORE CLOCK or (b) bits 8-71 of the results of STORE CLOCK EXTENDED. However, when compared with the results of STORE CLOCK or STORE CLOCK EXTENDED, the values returned by STORE CLOCK FAST do not necessarily indicate the correct sequence of execution of the instruction by one or more CPUs. STORE CLOCK FAST should not be used in programs where a monotonically increasing result is required.

TOD-Clock Synchronization

The following functions are provided if the configuration is part of an ETR network:

- A clock in the stopped state, with the TOD-clock-sync-control bit (bit 34 of control register 0) set to one, is placed in the set state and starts incrementing when an ETR signal occurs.
- The stepping rates for the TOD clock and the ETR are synchronized.
- Bits 32 through the rightmost incremented bit of a clock in the set state are compared with the same bits of the ETR. An unequal condition is signaled by an external-damage machine-check-interruption condition. The machine-check-interruption condition may not be recognized for up to 1.048576 seconds (2^{20} microseconds) after the unequal condition occurs.

When the server timer protocol (STP) facility is installed, the timing mode, timing state and STP-clock-source state are defined as described below.

Timing Mode

The timing mode specifies the method by which the TOD clock is maintained for purposes of synchronization within a timing network. A TOD clock operates in one of the following timing modes:

Local Timing Mode: When the configuration is in local timing mode, the TOD clock has been initialized to a local time and is being stepped at the rate of the local hardware oscillator. The configuration is not part of a synchronized timing network.

ETR Timing Mode: When the configuration is in ETR-timing mode, the TOD clock has been initialized to the ETR and is being stepped by stepping signals from ETR. To be in ETR-timing mode, the configuration must be part of an ETR network.

If the machine implements hardware-based TOD-clock steering, the ETR timing mode is not available

STP Timing Mode: When the configuration is in STP-timing mode, the TOD clock has been initialized to coordinated server time (CST) and is being stepped at the rate of the local hardware oscillator. In STP timing mode, the TOD clock is steered so as to maintain, or attain, synchronization with CST. To be in STP-timing mode, the configuration must be part of an STP network.

Timing State

The timing state indicates the synchronization state of the TOD clock with respect to the timing network reference time.

Synchronized State: When a configuration is in the synchronized timing state, the TOD clock is in synchronization with the timing-network reference time as defined below:

- If the configuration is in ETR-timing mode, the configuration is synchronized with the ETR.

- If the configuration is in STP timing mode, the configuration is synchronized with coordinated server time (CST).

A configuration that is in the local-timing or uninitialized-timing mode is never in the synchronized state.

Unsynchronized State: When a configuration is in the unsynchronized timing state, the TOD clock is not in synchronization with the timing network reference time as defined below:

- If the configuration is in ETR-timing mode, the configuration has lost synchronization with the ETR.
- If the configuration is in STP timing mode, the configuration has lost or has not been able to attain synchronization with coordinated server time (CST). The configuration is out of synchronization with CST when the TOD clock differs from CST by an amount that exceeds a model dependent STP-sync-check-threshold value.

Stopped State: When a configuration is in the stopped timing state, either the TOD clock is in the stopped state, or TOD-clock recovery is in progress. After TOD-clock recovery completes, the TOD clock enters either the synchronized or unsynchronized state.

STP Clock Source State

The STP-clock-source state indicates whether a usable STP-clock source is available. The STP-clock source is used to determine the coordinated server time (CST) required to be able to synchronize the TOD clock.

Not Usable: The not-usable-STP-clock-source state indicates that a usable STP-clock source is not available to the STP facility. When a usable STP-clock source is not available, CST can not be determined.

Usable: The usable-STP-clock-source state indicates that a usable STP-clock source is available to the STP facility. When a usable STP-clock source is available, CST has been determined and can be used to synchronize the TOD clock to the STP network.

Programming Notes:

1. ETR TOD-clock synchronization provides for synchronizing and checking only bits 32 through the rightmost incremented bit of the TOD clock. Bits 0-31 of the TOD clock may be different from those of the ETR.
2. If the configuration is part of an ETR network, SET CLOCK must place all zeros in bit positions 32 through the rightmost incremented bit position of the TOD clock; otherwise, an external-damage machine-check-interruption condition will be recognized.

TOD-Clock Steering

TOD clock steering provides the means by which the TOD clock can be steered to maintain synchronization to a selected clock source. TOD clock steering is performed using one of the following steering mechanisms:

Offset-Based TOD-Clock Steering: The apparent stepping rate of the TOD clock is changed without changing the physical hardware oscillator which steps the physical clock. This is accomplished by means of a TOD offset register which is added to the physical clock to produce the TOD clock.

Hardware-Based TOD-Clock Steering: The stepping rate of the physical hardware oscillator is directly modifiable and provides the TOD clock frequency. A physical clock is made available via PTFF that is derived from the TOD clock in a manner that provides the appearance of a fixed-frequency physical clock. This is accomplished by means of a TOD-clock offset that is subtracted from the TOD clock to produce the physical clock.

When the STP-hardware-based-TOD-clock-steering facility is not installed, TOD-clock steering is performed using offset-based TOD-clock steering. When the STP-hardware-based-TOD-clock-steering facility is installed, steering is performed using hardware-based TOD-clock steering.

The steering method used is transparent to all programs.

TOD-clock steering includes the semiprivileged instruction PERFORM TIMING FACILITY FUNCTION (PTFF), which is in the “E” format, provides a

7-bit function code in general register 0 and a parameter block address in general register 1.

The presence of the TOD-clock steering facility is indicated by bit 28 of the facility list, as stored by the STORE FACILITY LIST or STORE FACILITY LIST EXTENDED instructions. Facility-indication bit 28, when one indicates that the TOD-clock steering facility is installed. The instruction PERFORM TIMING FACILITY FUNCTION (PTFF) is installed only in the z/Architecture architectural mode. However, when the TOD-clock steering facility is installed, it is not disabled when the configuration switches architectural modes.

The total steering rate is made up of two components, a fine-steering rate and a gross-steering rate.

The fine-steering rate is used to correct that inaccuracy in the local oscillator which is stable over a relatively long period of time. The value will normally be less than the specified tolerance of the local oscillator (typically $\pm$ 2.0 ppm), changes will occur infrequently (on the order of once per day to once per week), and changes will be small (typically less than $\pm$ 0.2 ppm).

The gross-steering rate is used as a dynamic correction for all other effects, the most predominate being to synchronize time with an external time source and with other clocks in the timing network. The value will normally change frequently (on the order of once per second to once per minute); and the value may range to more than $\pm$ 40 ppm.

TOD-clock steering includes the PTFF set-TOD-offset and set-TOD-offset-user control functions which allow a program to apply a 64-bit TOD epoch difference to the TOD clock.The multiple-epoch facility includes the PTFF set-TOD-offset-extended and set-TOD-offset-user-extended functions which allow a program to apply a 72-bit TOD epoch difference to the concatenation of the epoch index and TOD clock.

When the configuration is in STP timing mode, and the timing-control program detects an excessive offset between the system TOD and some reference clock, a new episode may be started with a step change to the TOD offset (see “Episodes” on page 4-56). This is however not visible to programs, as the epoch difference is adjusted in a complementary way. The epoch difference therefore consists of two components: a sync-check offset, and a user-specified epoch difference. A control program may request sync check notification, and subsequently request that its uncorrected sync check offset be corrected, resulting in a program-visible step in the TOD in a controlled manner.

TOD-clock steering also includes several query functions which may be used by the problem-state program to determine the quality of the TOD clock.

Offset-Based TOD-Clock Steering Overview

When the STP-hardware-based-TOD-clock-steering facility is not installed, offset-based TOD-clock steering is performed by computing a TOD-clock offset that is added to the physical clock to form the system TOD clock. Offset-based TOD-clock steering is accomplished by means of three values that are used to compute the TOD-clock offset: a start time (s), a base offset (b), and a steering rate (r).

The steering rate is a 32-bit signed binary fixed-point value and considered to be scaled by a factor of 2^{-44}.

When the multiple-epoch facility is not installed, the start time is a 64-bit unsigned binary integer with a resolution equal to bit 63 of the TOD clock that is set to the physical clock start time for the current steering episode (see "Episodes" on page 4-56). When the multiple-epoch facility is installed, the start time may be extended on the left with up to eight additional bits.

When the multiple-epoch facility is not installed, the base offset is a 64-bit signed binary integer with a resolution equal to bit 63 of the TOD clock. When the multiple-epoch facility is installed, the base offset may be extended on the left with up to eight additional bits.

Hardware-Based TOD-Clock-Steering Overview

When the STP-hardware-based-TOD-clock-steering facility is installed, hardware-based steering of the TOD clock is accomplished by changing the stepping rate of the hardware oscillator. At the start of each steering episode, the hardware steering rate is set to a steering rate that reflects the current total steering rate for the configuration. At an episode start time, the TOD-clock base offset is updated to include the accumulated hardware steering for the previous (old) episode. The TOD-clock base offset is used to compute a current TOD-clock offset that is used to determine the physical clock at any given time. The conceptual steering parameters are the same as for

offset-based TOD-clock steering: a start time (s), a base offset (b), and a steering rate (r) (see "Episodes" on page 4-56).

When hardware-based TOD-clock steering is installed, ETR steering does not apply.

TOD-Offset-Update Events

When the STP-hardware-based-TOD-clock-steering facility is not installed, the TOD offset is updated periodically rather than being computed continuously. This update is referred to as a TOD-offset-update event. A TOD-offset-update event is triggered by the carry out of a bit position of the physical clock. The bit position depends on the model, but is chosen such that for normal steering rates, the difference between the values computed for the TOD offset (d) by consecutive TOD-offset-update events is less than the resolution of the TOD clock.

When the STP-hardware-based-TOD-clock-steering facility is installed, the steering rate can be viewed as being applied continuously. The TOD offset is calculated immediately, whenever it is needed; thus, there is no TOD-offset-update event.

Episodes

The three values, start time (s), base offset (b), and steering rate (r), define a linear steering adjustment which can be applied indefinitely. The duration that these values are applied without being changed is called an episode.

When the STP-hardware-based-TOD-clock-steering facility is not installed, a request by the timing-facility-control program to change the steering rate causes the machine to schedule a new episode to take effect at a future time. To provide a smooth transition, the machine schedules the start time for the new episode to be at the next TOD-offset-update event and computes a new base offset such that there will be no discontinuity in the value of the TOD offset at the instant the new values take effect.

When the STP-hardware-based-TOD-clock-steering facility is installed, a request by timing-facility-control program to change the steering rate causes a new episode to begin immediately. A new base offset is computed such that there is no discontinuity in the value of the TOD offset at the instant the new values take effect.

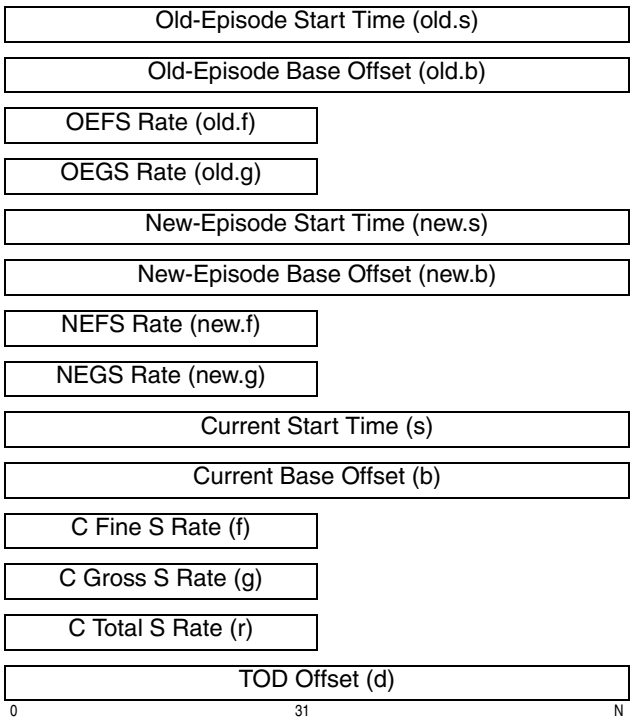
The machine places the new values into special registers called new-episode start time (new.s), new-episode base offset (new.b), new-episode fine-steering rate (new.f); and new-episode gross-steering rate (new.g); and the previous contents of these four registers are preserved by placing them into registers called old-episode start time (old.s), old-episode base offset (old.b), old-episode fine-steering rate (old.f), and old-episode gross-steering rate (old.g), respectively. The machine continues to use the values for the old episode until the new-episode start time (new.s) and then automatically switches to use the values for the new episode. The registers in use at any particular instant in time are called current start time (s), current base offset (b), and current total steering rate (r). These are collectively referred to as the current-episode registers. Additionally, when the STP-hardware-based-TOD-clock-steering facility is installed, the hardware steering rate (hr) is set to a value that corresponds to the current steering rate in a model-dependent manner.

TOD-Clock-Steering Registers

Figure 4-14 summarizes the TOD-clock-steering registers. The contents of all TOD-clock-steering registers are initialized to zero by power-on reset, except for the user-specified-clock offset, whose initial value may be part of configuration information.

Current Start Time (s): When the machine is operating in the old episode, the current start time is obtained from the old-episode start time (old.s); and when in the new episode, it is obtained from the new-episode start time (new.s). When the multiple-epoch facility is not installed, the current start time (s) is a 64-bit unsigned binary integer that has a resolution equal to bit 63 of the TOD clock. When the multiple-epoch facility is installed, the current start time is a 72-bit unsigned binary integer that has a resolution equal to bit 63 of the TOD clock; however, depending on the model, not all of the leftmost 8 bits may be implemented.

Current Base Offset (b): When the machine is operating in the old episode, the current base offset is obtained from the old-episode base offset (old.b); and when in the new episode, it is obtained from the new-episode base offset (new.b). When the multiple-epoch facility is not installed, the current base offset (b) is a 64-bit signed binary integer that has a resolution equal to bit 63 of the TOD clock. When the multiple-epoch facility is installed, the current base offset



Explanation:

N When the multiple-epoch facility is not installed, N is 63; thus the wider fields shown above are 64 bits. When the multiple-epoch facility is installed, N may be up to 71; thus the wider fields may be up to 72 bits. In either case, the field is right justified, with the rightmost bit having the resolution of a clock unit.

Figure 4-14. TOD-Clock-Steering Registers

is a 72-bit signed binary integer that has a resolution equal to bit 63 of the TOD clock; however, depending on the model, not all of the leftmost 8 bits may be implemented.

Current Steering Rates (f,g,r): When the machine is operating in the old episode, the current fine-steering rate (f) and current gross-steering rate (g) are obtained from the old-episode fine-steering rate (old.f) and gross-steering rate (old.g), respectively; when in the new episode, they are obtained from the new-episode fine-steering rate (new.f) and gross-steering rate (new.g), respectively.

The current total steering rate (r) is obtained from the sum of the current fine-steering rate (f) and the current gross-steering rate (g). A carry, if any, out of bit position 0, is ignored in this addition. The current total steering rate (r) is a 32-bit signed binary fixed-point value and considered to be scaled by a factor of 2^{-44}.

Programming Note: Bits 0 and 31 of the steering-rate represent steering rates of -2^{-13} and 2^{-44}, respectively. Thus, steering rates of ±122 parts per million (10.5 seconds per day) may be specified with a precision of 4.9 nanoseconds per day.

TOD Offset (d): When the multiple-epoch facility is not installed, the TOD offset (d) is a 64-bit signed binary integer with a resolution equal to bit 63 of the TOD clock. . When the multiple-epoch facility is installed, the TOD offset is a 72-bit signed binary integer with a resolution equal to bit 63 of the TOD clock.

When the STP-hardware-based-TOD-clock-steering facility is not installed, the contents of the TOD offset are added to the physical clock to obtain the system TOD clock. When the STP-hardware-based-TOD-clock-steering facility is installed, the contents of the TOD offset are subtracted from the system TOD clock to obtain the physical clock.

Depending on the model, rightmost bits of the TOD offset corresponding to bits beyond the resolution of the TOD clock may not be implemented and are treated as zeros.

UTC Information Block (UIB)

The UIB includes information to convert from a TOD clock timestamp to primary reference time (PRT) and then to convert PRT to UTC. The UIB can be inspected by executing PERFORM TIMING FACILITY FUNCTION specifying the PTFF-QUI (Query UTC Information) function. Conversion from TOD to PRT uses the TOD-to-PRT parameters and conversion from PRT to UTC uses the leap-second information.

The information in the UIB is provided by the server-time-protocol (STP) facility that also controls TOD-clock steering. When STP is not installed, all fields in the UIB are zero. STP manages a coordinated timing network (CTN) that provides the illusion of a single global TOD clock – known as coordinated server time (CST) – shared by several machines.

The format of the UIB is shown in Figure 4-15.

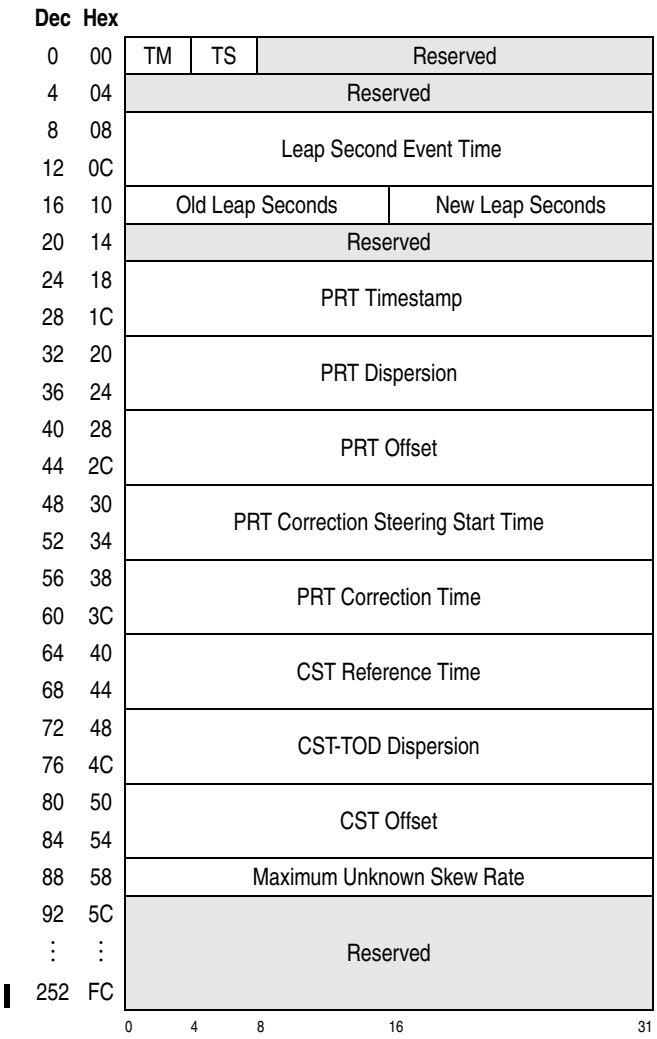


Figure 4-15. UTC Information Block

Timing Mode (TM): Bits 0-3 of byte 0 of the UIB contain a 4-bit code indicating the timing mode of the system. The codes are defined as follows:

Code	Meaning
0	Local Timing Mode
1	ETR Timing Mode
2	STP Timing Mode
3-15	Reserved

Timing State: Bits 4-7 of byte 0 of the UIB contain a 4-bit code indicating the timing state of the system. The codes are defined as follows:

Code	Meaning
0	Unsynchronized
1	Synchronized
2-15	Reserved

Leap Second Information: Three fields in the UIB provide information for converting from primary reference time to UTC. These are the leap-second-event-time, old-leap-seconds, and new-leap-seconds fields. For a primary reference time less than the leap-second-event time, the old-leap-seconds value applies; for a primary reference time equal to or greater than the leap-second-event time, the new-leap-seconds value applies. The applicable leap-second value is subtracted from the primary reference time (PRT) to form UTC.

Leap Second Event Time: Bytes 8-15 of the UIB, when valid, contain a 64-bit time in TOD-clock timestamp format that specifies the primary reference time at which the new-leap seconds is to take effect.

Old Leap Seconds: Bytes 16-17 of the UIB contain a 16-bit signed binary integer indicating the number of leap seconds in effect prior to the leap-second-event time. Thus, when the leap-second-event time is nonzero, the old-leap-seconds value is in effect for a primary reference time less than the leap-second-event time. When the leap-second-event time is zero, no change has been scheduled, and the old-leap-seconds and new-leap-seconds fields contain the same value. The value is provided in seconds with the low-order bit equaling one second.

New Leap Seconds: Bytes 18-19 of the UIB contain a 16-bit signed binary integer indicating the number of leap seconds in effect for a primary reference time equal to or greater than the leap-second-event time. The value is provided in seconds with the low-order bit equaling one second.

PRT Timestamp: Bytes 24-31 of the UIB, when valid, contain a 64-bit timestamp that specifies the time at which the primary-reference-time (PRT) parameters (PRT dispersion, PRT offset, PRT-correction-steering-start time, and PRT-correction time) were last updated within the CTN. This field has the same format as bits 0-63 of the TOD clock. The PRT-

update-event-time field is valid only when it is nonzero.

PRT Dispersion: Bytes 32-39 of the UIB, when valid, contain a 64-bit unsigned binary integer indicating the total maximum possible dispersion of the PRT offset provided at the most recent PRT-update event. This field has the same format as bits 0-63 of the TOD clock. The PRT-dispersion field is valid only when the PRT-timestamp field is nonzero.

PRT Offset: Bytes 40-47 of the UIB, when valid, contain a 64-bit signed binary integer indicating the PRT offset provided for the most recent PRT information. This field has the same format as bits 0-63 of the TOD clock. The value is the amount to be added to the system TOD to match the estimated time at the primary-reference time source. The margin of error of this value is equal to the PRT dispersion.

PRT Correction Steering Start Time: Bytes 48-55 of the UIB, when valid, contain a 64-bit timestamp having the same format as bits 0-63 of the TOD clock and indicating the time at which PRT-correction steering is to be, or has been, initiated within the CTN. A value of zero indicates that no PRT correction was planned at the time when the UIB was last updated.

PRT Correction Time: Bytes 56-63 of the UIB, when valid, contain a 64-bit unsigned binary integer indicating the estimated amount of time required to correct the PRT offset which is the amount of time PRT-correction will be in effect. This field has the same format as bits 0-63 of the TOD clock.The field is valid when the PRT-correction-steering-start time is nonzero.

CST Reference Time: Bytes 64-71 of the UIB, when valid, contain a 64-bit value in TOD-clock timestamp format indicating the value of the system TOD clock at the most recent time that the CST parameters (CST-TOD dispersion and CST offset) were updated. The field is valid when it is nonzero.

CST-TOD Dispersion: Bytes 72-79 of the UIB, when valid, contain a 64-bit unsigned binary integer indicating the CST-TOD dispersion computed at the most recent CST parameter update. The value has a resolution equal to bit 63 of the TOD clock. The CST-TOD-dispersion field is valid when the CST-reference-time field is nonzero.

CST Offset: Bytes 80-87 of the UIB, when valid, contain a 64-bit signed binary value that indicates the offset of the system TOD clock to the TOD clock at the machine selected as the time source. The CST offset is added to the system-TOD clock to form the Coordinated Server Time (CST) at the machine. For the add operation, a carry out of bit 0 ignored. Bit 63 of the CST offset has a resolution equal to that of bit 63 of the TOD clock.The CST-offset field is valid when the CST-reference-time field is nonzero. For stratum-1 servers, the field is set to zero (by definition, CST is the system TOD at the stratum-1 server).

Maximum Unknown Skew Rate: Bytes 88-91 of the UIB, when valid, contain a 32-bit unsigned binary integer indicating the absolute value of the unknown skew rate of the physical clock. The value has a resolution of one part per 2^{44}. The maximum-skew-rate field is valid when it is nonzero.

Clock Comparator

The clock comparator provides a means of causing an interruption when the TOD-clock value exceeds a value specified by the program.

In a configuration with more than one CPU, each CPU has a separate clock comparator.

The clock comparator has the same format as bits 0-63 of the TOD clock. The clock comparator nominally consists of bits 0-47, which are compared with the corresponding bits of the TOD clock. In some models, higher resolution is obtained by providing more than 48 bits. The bits in positions provided in the clock comparator are compared with the corresponding bits of the clock. When the resolution of the clock is less than that of the clock comparator, the contents of the clock comparator are compared with the clock value as this value would be stored by executing STORE CLOCK or STORE CLOCK FAST.

The clock comparator causes an external interruption with the interruption code 1004 hex; presentation of the interruption is subject to the clock-comparator subclass mask, bit 52 of control register 0. When the TOD-clock-steering facility is not installed, a request for a clock-comparator interruption exists whenever either of the following conditions exists:

1. The TOD clock is running and the value of the clock comparator is less than the value in the compared portion of the clock, both values being

considered unsigned binary integers. Comparison follows the rules of unsigned binary arithmetic.

2. The TOD clock is in the error state or the not-operational state.

When the TOD-clock-steering facility is installed, a request for a clock-comparator interruption exists whenever the physical clock is in the set state and the value of the clock comparator is less than the value in the compared portion of the logical TOD clock.

When the clock-comparator sign control, bit 10 of control register 0, is zero, comparison follows the rules of unsigned binary arithmetic. When the multiple-epoch facility is not installed in the configuration and the clock-comparator sign control is one, it is unpredictable whether the comparison follows the rules of unsigned or signed binary arithmetic. When the multiple-epoch facility is installed in the configuration and the clock-comparator sign control is one, comparison follows the rules of signed binary arithmetic.

A request for a clock-comparator interruption does not remain pending when the value of the clock comparator is made equal to or greater than that of the logical TOD clock or when the value of the logical TOD clock is made less than the clock-comparator value. The latter may occur as a result of the physical clock or logical TOD clock being set; when the multiple-epoch facility is not installed or when the facility is installed but the clock-comparator sign control is zero, the latter may occur as a result of the logical TOD clock wrapping to zero; when the multiple-epoch facility is installed and the clock-comparator sign control is one, the latter may occur as a result of the logical TOD clock, considered as a signed value, transitioning to the maximum negative value.

The clock comparator can be inspected by executing the instruction STORE CLOCK COMPARATOR and can be set to a specified value by executing the SET CLOCK COMPARATOR instruction.

When the multiple-epoch facility is installed, a change to the clock-comparator sign control may immediately cause the conditions for requesting a clock-comparator interruption to exist or no longer exist (depending on the values of the TOD clock and clock comparator). However, such a change may not immediately cause a request for a clock comparator to be made or withdrawn. See programming note 3 for additional information.

The contents of the clock comparator and clock-comparator sign control are initialized to zero by initial CPU reset.

Programming Notes:

1. An interruption request for the clock comparator persists as long as the clock-comparator value is less than that of the TOD clock or as long as the TOD clock is in the error state or the not-operational state. Therefore, one of the following actions must be taken after an external interruption for the clock comparator has occurred and before the CPU is again enabled for external interruptions:

   - The value of the clock comparator must be replaced.
   - The TOD clock must be set.
   - The clock-comparator-subclass mask must be set to zero.
   - The TOD clock must wrap to zero (applicable when the multiple-epoch facility is not installed, or when the facility is installed but the clock-comparator sign control is zero).

   Otherwise, loops of external interruptions are formed.

2. The instruction STORE CLOCK, STORE CLOCK EXTENDED, or STORE CLOCK FAST may store a value which is greater than that in the clock comparator, even though the CPU is enabled for the clock-comparator interruption. This is because the TOD clock may be incremented one or more times between when instruction execution is begun and when the clock value is accessed. In this situation, the interruption occurs when the execution of STORE CLOCK, STORE CLOCK EXTENDED, or STORE CLOCK FAST is completed.

3. When the multiple-epoch facility is installed in the configuration, the clock-comparator sign control provides the means by which a control program can specify what constitutes a discontinuity in the TOD clock for the purposes of clock-comparator checking: either transitioning from the maximum unsigned TOD-clock value to zero, or transition-

ing from the maximum positive signed TOD-clock value to the maximum negative value.

Assuming that a configuration is re-initialized at least once during a half of an epoch, it is recommended that the clock-comparator sign control be set to the contents of bit position 0 of the TOD clock when the program is initialized. This ensures that clock-comparator checking observes no discontinuity in the TOD clock.

To ensure consistent results when altering the clock-comparator sign control, the program should (a) disable clock-comparator interruptions, (b) set the clock-comparator sign control, (c) issue the SET CLOCK COMPARATOR instruction to set a new comparator value, and then (d) enable for clock-comparator interruptions as appropriate.

Once set, the clock-comparator sign control is intended to remain unchanged until the next initial CPU reset (such as at IPL). Dynamic changing of the clock-comparator sign control without following the recommendations in this note may result in false recognition of a clock-comparator condition that is withdrawn as a result of the changed control or in the delayed recognition of a clock-comparator condition that becomes pending as a result of the changed control.

4. When the clock-comparator sign control is zero, (a) the program can set the clock comparator to all zeros to ensure that an interruption condition is immediately present, and (b) the program can set the clock comparator to a value of all binary ones to ensure that a clock-comparator interruption is never recognized. When the multiple-epoch facility is installed in the configuration and the clock-comparator sign control is one, (a) the program can set the clock comparator to the maximum negative value (8000000000000000 hex) to ensure that an interruption condition is immediately present, and (b) the program can set the clock comparator value to the maximum positive value (7FFFFFFFFFFFFFFF hex) to ensure that a clock-comparator interruption is never recognized.

5. When the multiple-epoch facility is not installed in the configuration, bit 10 of control register 0 should always be set to zero to ensure consistent unsigned comparison of the clock comparator.

6. Figure 4-16 illustrates the effects of the clock-comparator sign control on the recognition of

clock-comparator interruption conditions, based on various clock-comparator and TOD-clock values.

Register Value	Clock-Comparator Sign Control (CR0.10)																					
	Zero (Unsigned)										One (Signed)											
	CLKC	TOD	CLKC	TOD	CLKC	TOD	CLKC	TOD	CLKC	TOD	CLKC	TOD	CLKC	TOD	CLKC	TOD	CLKC	TOD	CLKC	TOD	CLKC	TOD
00000000 00000000	✓																✓					
00000000 00000001																						
⋮																						
456789AB CDEF0123			✓																✓			
456789AB CDEF0124																						
⋮																						
7FFFFFFF FFFFFFFF					✓																✓	
80000000 00000000											✓											
⋮																						
CDEF1234 56789ABC							✓						✓									
CDEF1234 56789ABD																						
⋮																						
FFFFFFFF FFFFFFFF									✓						✓							

Explanation:

Pairs of columns represent a clock-comparator (CLKC) value, and a range of time-of-day (TOD) clock values for which a clock-comparator interruption condition does or does not exist.

For each pair of columns, the clock comparator is set to the value shown in the "Register Value" column corresponding to the row in which the check mark (✓) appears. TOD-clock cells that are not shaded indicate that a clock-comparator interruption condition does not exist for the corresponding TOD-clock value in the "Register Value" column. TOD-clock cells that are shaded (if any) indicate that a clock-comparator interruption condition exists for the corresponding values in the "Register Value" column.

Figure 4-16. Effects of the Clock-Comparator Sign Control on the Recognition of Clock-Comparator Interruption Conditions

CPU Timer

The CPU timer provides a means for measuring elapsed CPU time and for causing an interruption when a specified amount of time has elapsed.

In a configuration with more than one CPU, each CPU has a separate CPU timer.

The CPU timer is a binary counter with a format which is the same as that of bits 0-63 of the TOD clock, except that bit 0 is considered a sign. The CPU timer nominally is decremented by subtracting a one in bit position 51 every microsecond. In models having a higher or lower resolution, a different bit position is decremented at such a frequency that the rate of decrementing the CPU timer is the same as if a one were subtracted in bit position 51 every microsecond. The resolution of the CPU timer is such that the stepping rate is comparable to the instruction-execution rate of the model.

The CPU timer requests an external interruption with the interruption code 1005 hex whenever the CPU-timer value is negative (bit 0 of the CPU timer is one); presentation of the interruption is subject to the CPU-timer subclass mask, bit 53 of control register 0. The request does not remain pending when the CPU-timer value is changed to a nonnegative value.

When the TOD-clock-steering facility is not installed, and both the CPU timer and the TOD clock are running, the stepping rates are synchronized such that both are stepped at the same rate.

When the TOD-clock-steering facility is installed and the STP-hardware-based-TOD-clock-steering facility

is not installed, the CPU timer and TOD clock do not necessarily step at the same rate and may differ by as much as the maximum steering rate. When the TOD-clock-steering facility is installed and the STP-hardware-based-TOD-clock-steering facility is installed, the CPU timer and TOD clock step at the same rate.

Normally, decrementing the CPU timer is not affected by concurrent I/O activity. However, in some models the CPU timer may stop during extreme I/O activity and other similar interference situations. In these cases, the time recorded by the CPU timer provides a more accurate measure of the CPU time used by the program than would have been recorded had the CPU timer continued to step.

The CPU timer is decremented when the CPU is in the operating state or the load state. When the manual rate control is set to instruction step, the CPU timer is decremented only during the time in which the CPU is actually performing a unit of operation. However, depending on the model, the CPU timer may or may not be decremented when the TOD clock is in the error, stopped, or not-operational state.

Depending on the model, the CPU timer may or may not be decremented when the CPU is in the check-stop state.

The CPU timer can be inspected by executing the privileged instruction STORE CPU TIMER and can be set to a specified value by executing the privileged SET CPU TIMER instruction. The general instruction EXTRACT CPU TIME may be used in determining the amount of CPU time consumed by a task.

Depending on the model, the value of the CPU timer stored by an execution of STORE CPU TIMER may be the same as the value stored by a subsequent execution of STORE CPU TIMER on the same CPU when relatively few instructions are executed between the STORE CPU TIMER instructions. Similarly, depending on the model, the value of the CPU TIMER set by an execution of SET CPU TIMER may be the same value as that stored by a subsequent execution of STORE CPU TIMER on the same CPU when relatively few instructions are executed between the setting and storing of the timer.

The CPU timer is set to zero by initial CPU reset.

Programming Notes:

1. The CPU timer in association with a program may be used both to measure CPU-execution time and to signal the end of a time interval on the CPU.

2. The time measured for the execution of a sequence of instructions may depend on the effects of such things as I/O interference, the availability of pages, and instruction retry. Therefore, repeated measurements of the same sequence on the same installation may differ.

3. The fact that a CPU-timer interruption does not remain pending when the CPU timer is set to a positive value eliminates the problem of an undesired interruption. This would occur if, between the time when the old value is stored and a new value is set, the CPU is disabled for CPU-timer interruptions and the CPU timer value goes from positive to negative.

4. The fact that CPU-timer interruptions are requested whenever the CPU timer is negative (rather than just when the CPU timer goes from positive to negative) eliminates the requirement for testing a value to ensure that it is positive before setting the CPU timer to that value.

   As an example, assume that a program being timed by the CPU timer is interrupted for a cause other than the CPU timer, external interruptions are disallowed by the new PSW, and the CPU-timer value is then saved by STORE CPU TIMER. This value could be negative if the CPU timer went from positive to negative since the interruption. Subsequently, when the program being timed is to continue, the CPU timer may be set to the saved value by SET CPU TIMER. A CPU-timer interruption occurs immediately after external interruptions are again enabled if the saved value was negative.

   The persistence of the CPU-timer-interruption request means, however, that after an external interruption for the CPU timer has occurred, the value of the CPU timer must be replaced, the value in the CPU timer must wrap to a positive value, or the CPU-timer-subclass mask must be set to zero before the CPU is again enabled for external interruptions. Otherwise, loops of external interruptions are formed.

   Although an initial CPU reset causes the CPU timer to be set to zero, it also clears the CPU-

timer enablement mask (bit 53 of control register 0). Thus, even if an IPL-new PSW is enabled for external interruptions, a CPU-timer interruption will not immediately occur.

5. The instruction STORE CPU TIMER may store a negative value even though the CPU is enabled for the interruption. This is because the CPU-timer value may be decremented one or more times between when instruction execution is begun and when the CPU timer is accessed. In this situation, the interruption occurs when the execution of STORE CPU TIMER is completed.

Guarded-Storage Facility

The guarded-storage facility provides the means by which the program can designate an area of logical storage comprising from zero to sixty-four guarded-storage sections. The facility includes the following instructions:

- LOAD GUARDED (LGG)
- LOAD LOGICAL AND SHIFT GUARDED (LLGFSG)
- LOAD GUARDED STORAGE CONTROLS (LGSC)
- STORE GUARDED STORAGE CONTROLS (STGSC)

When the second operand of LGG or LLGFSG does not designate a guarded section of the guarded-storage area, the instruction performs its defined load operation. However, when the second operand of the instruction designates a guarded section of the guarded-storage area, control branches to a guarded-storage event handler with indications of the cause of the event.

The guarded-storage facility is intended to be used by various programming languages that implement storage-reclamation techniques often referred to as *garbage collection*. All other instructions that access a range of guarded storage are unaffected by the facility, only the LGG and LLGFSG instructions are capable of generating a guarded-storage event.

Guarded-Storage-Facility Registers

The guarded-storage facility is controlled by a bit in control register 2 and by the following three 64-bit registers:

- Guarded-storage-designation register
- Guarded-storage-section-mask register
- Guarded-storage-event parameter-list-address register

The contents of these three registers may be loaded and inspected by means of the LOAD GUARDED STORAGE CONTROLS and STORE GUARDED STORAGE CONTROLS instructions respectively.

Control Register 2

When the guarded-storage facility is installed, bit 59 of control register 2 is the guarded-storage-facility-enablement (GSFE) control. When bit 59 is zero, attempted execution of the following instructions results in a special-operation exception:

- LOAD GUARDED STORAGE CONTROLS
- STORE GUARDED STORAGE CONTROLS

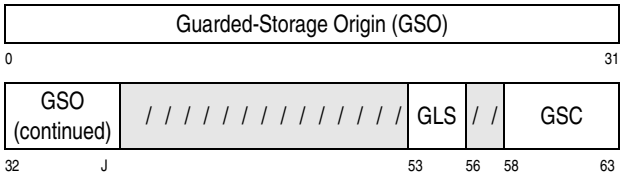
When the GSFE control is one, the guarded-storage facility is said to be enabled.

Execution of the LOAD GUARDED and LOAD LOGICAL AND SHIFT GUARDED instructions is not subject to the GSFE control. A guarded-storage event is only recognized when the GSFE control is one.

Programming Note: When facility indication 133 is one (indicating that the guarded-storage facility is installed in the configuration), the program can use the LOAD GUARDED and LOAD LOGICAL AND SHIFT GUARDED instructions, regardless of the GSFE control. However, guarded-storage events cannot be recognized without first loading guarded-storage controls, and the control program must set the GSFE control to one in order to successfully execute the LOAD GUARDED STORAGE CONTROLS instruction. Therefore, the program should examine an OS-provided indication of guarded-storage-facility enablement (rather than facility bit 133) to determine if the full capabilities of the facility are available.

Guarded-Storage-Designation (GSD) Register

The guarded-storage designation register is a 64-bit register that defines the attributes of the guarded-storage area as shown in Figure 4-17.



Explanation:

/	Reserved
GLS	Guarded load shift amount
GSC	Guarded-storage characteristic
J	The rightmost significant bit position of the GSO (that is, bit position 63 minus the value of GSC).

Figure 4-17. Guarded-Storage-Designation (GSD) Register

The fields of the GSD register are as follows:

Guarded-Storage Origin (GSO): The location of the guarded-storage area is specified by the leftmost bits of the GSD register. The number of leftmost bits is determined by value of the guarded-storage-characteristic (GSC) in bits 58-63 of the register. Bit positions 0 through (63 – GSC) of the guarded-storage-designation register, padded on the right with binary zeros in bit positions (64 – GSC) through 63, form the 64-bit logical address of the leftmost byte of the guarded-storage area.

Reserved: When the GSC is greater than 25, bit positions (64 – GSC) through 38 are reserved and should contain zeros; otherwise, the results of the guarded-storage-event detection are unpredictable (see "Guarded-Storage-Event Detection" on page 4-69). Bit positions 39-52 and 56-57 of the GSD register are reserved and should contain zeros; otherwise, the program may not operate compatibly in the future.

Guarded-Load Shift (GLS): Bits 53-55 of the guarded-storage-designation register contains a 3-bit unsigned binary integer that is used in the formation of the intermediate result of LOAD LOGICAL AND SHIFT GUARDED. Valid GLS values are 0-4; values 5-7 are reserved and may result in an unpredictable shift amount.

Guarded-Storage Characteristic (GSC): Bit positions 58-63 of the guarded-storage-designation register contain a 6-bit unsigned binary integer that is treated as a power of two. Valid GSC values are 25-56; values 0-24 and 57-63 are reserved and may result in unpredictable guarded-storage event detection. The GSC designates the following:

- The alignment of the guarded-storage origin. A GSC value of 25 indicates 32 M-byte alignment, a value of 26 indicates 64 M-byte alignment, and so forth.
- The guarded-storage-section size. A GSC value of 25 indicates 512 K-byte sections, a value of 26 indicates a 1 M-byte sections, and so forth.

The relationship between the guarded-storage characteristic, guarded-storage origin, and guarded-storage-section size is shown in Figure 4-18.

GSC	GSO Alignment	GSO GSD Bits	Section Size	GSC	GSO Alignment	GSO GSD Bits	Section Size
25	32 M	0-38	512 K	41	2 T	0-22	32 G
26	64 M	0-37	1 M	42	4 T	0-21	64 G
27	128 M	0-36	2 M	43	8 T	0-20	128 G
28	256 M	0-35	4 M	44	16 T	0-19	256 G
29	512 M	0-34	8 M	45	32 T	0-18	512 G
30	1 G	0-33	16 M	46	64 T	0-17	1 T
31	2 G	0-32	32 M	47	128 T	0-16	2 T
32	4 G	0-31	64 M	48	256 T	0-15	4 T
33	8 G	0-30	128 M	49	512 T	0-14	8 T
34	16 G	0-29	256 M	50	1 P	0-13	16 T
35	32 G	0-28	512 M	51	2 P	0-12	32 T
36	64 G	0-27	1 G	52	4 P	0-11	64 T
37	128 G	0-27	2 G	53	8 P	0-10	128 T
38	256 G	0-25	4 G	54	16 P	0-9	256 T
39	512 G	0-24	8 G	55	32 P	0-8	512 T
40	1 T	0-23	16 G	56	64 P	0-7	1 P

Explanation:

G	Gigabytes (2^{30})	GSO	Guarded-storage origin
GSC	Guarded-storage characteristic	M	Megabytes (2^{20})
		P	Petabytes (2^{50})
		T	Terabytes (2^{40})

Figure 4-18. Guarded-Storage-Origin (GSO) Alignment, GSO Bit Range, and Guarded-Storage Section Size based on Guarded-Storage Characteristic (GSC)

Guarded-Storage-Section-Mask (GSSM) Register

The guarded-storage-section mask is a 64-bit register, with each bit corresponding to one of the 64

guarded-storage sections within the guarded-storage area. Bit 0 of the register corresponds to the leftmost section, and bit 63 corresponds to the rightmost section. Each bit, called a *section-guard* bit, controls the access to the respective section of guarded-storage area by the LOAD GUARDED and LOAD LOGICAL AND SHIFT GUARDED instructions, as described below and in the respective instructions' descriptions.

When all 64 bits of the GSSM register are zero, guarded-storage events are not recognized.

Guarded-Storage-Event Parameter-List-Address (GSEPLA) Register

When a guarded-storage-event is recognized, the contents of the GSEPLA register are a 64-bit address that is used to locate the guarded-storage-event parameter list, as described in the section "Guarded-Storage-Event Parameter List (GSEPL)", below. When the CPU is not in the access-register mode, the GSEPLA is a logical address; when the CPU is in the access-register mode, the GSEPLA is a primary virtual address.

Guarded-Storage Control Block (GSCB)

The three guarded-storage registers may be set and inspected by means of the LOAD GUARDED STORAGE CONTROLS and STORE GUARDED STORAGE CONTROLS instructions, respectively. The storage operand for each of these instructions is a 32-byte guarded-storage control block (GSCB), and the contents of the guarded-storage registers occupy the last three eight-byte fields of the block, as shown in Figure 4-19.

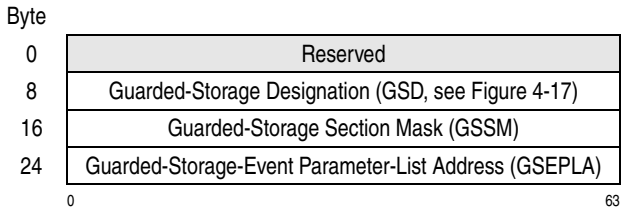


Figure 4-19. Guarded-Storage Control Block (GSCB)

When the GSCB is aligned on a doubleword boundary, CPU access to each of the three defined fields is block concurrent.

For LOAD GUARDED STORAGE CONTROLS, reserved bit positions of the GSCB should contain zeros; otherwise, the program may not operate compatibly in the future.

For STORE GUARDED STORAGE CONTROLS, reserved bit positions that are loaded with nonzero values may or may not be stored as zeros, and reserved values of the GLS and GSC fields of the GSD register may or may not be corrected to model-dependent values.

Guarded-Storage-Event Parameter List (GSEPL)

The guarded-storage-event parameter-list-address (GSEPLA) register contains a 64-bit address of the guarded-storage-event parameter list. When a guarded-storage event is recognized, the GSEPL is accessed using all 64 bits of the GSEPLA, regardless of the current addressing mode of the CPU. The GSEPL is accessed using the current translation mode, except that when the CPU is in the access-register mode, the GSEPL is accessed using the primary address space.

Figure 4-20 illustrates the fields of the guarded-storage-event parameter list.

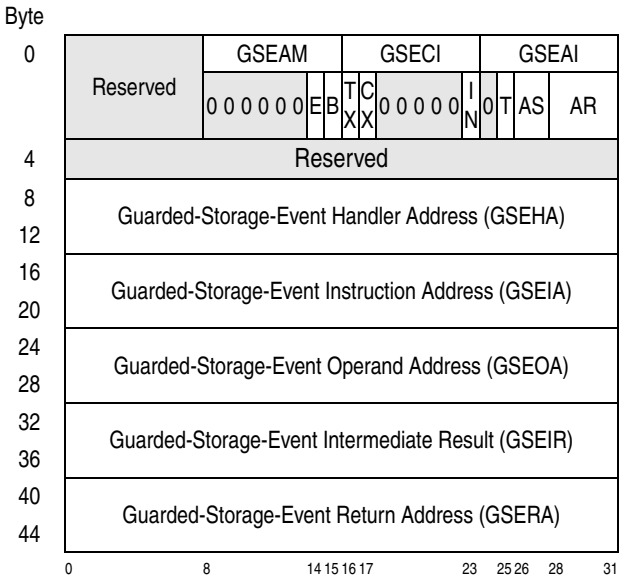


Figure 4-20. Guarded-Storage-Event Parameter List (GSEPL)

The contents of the guarded-storage-event parameter list are as follows:

Reserved: Bytes 0 and 4-7 of the GSEPL are reserved and set to zero when a guarded-storage event is recognized.

Guarded-Storage-Event Addressing Mode (GSEAM): Byte 1 of the GSEPL contains an indication of the addressing mode of the CPU when the guarded-storage-event was recognized, as follows:

Reserved: Bits 0-5 of the GSEAM are reserved and stored as zeros.

Extended-Addressing Mode (E): Bits 6 of the GSEAM contains the extended-addressing-mode bit, bit 31 of the program-status word.

Basic-Addressing Mode (B): Bits 7 of the GSEAM contains the basic-addressing-mode bit, bit 32 of the program-status word.

Guarded-Storage-Event Cause Indications (GSECI): Byte 2 of the GSEPL contains the guarded-storage-event cause indications. The GSECI is encoded as follows:

Transactional-Execution-Mode Indication (TX): When bit 0 of the GSECI is zero, the CPU was not in the transactional-execution mode when the guarded-storage event was recognized. When bit 0 of the GSECI is one, the CPU was in the transactional-execution mode when the guarded-storage event was recognized.

Constrained-Transactional-Execution-Mode Indication (CX): When bit 1 of the GSECI is zero, the CPU was not in the constrained-transactional-execution mode when the guarded-storage event was recognized. When bit 1 of the GSECI is one, the CPU was in the constrained-transactional-execution mode when the guarded-storage event was recognized. Bit 1 of the GSECI is meaningful only when bit 0 is one.

Reserved: Bits 2-6 of the GSECI are reserved and set to zero when a guarded-storage event is recognized.

Instruction Cause (IN): Bit 7 of the GSECI indicates the instruction that caused the guarded-storage event. When bit 7 is zero, the event was caused by the execution of the LOAD GUARDED instruction. When bit 7 is one, the event was caused by the execution of the LOAD LOGICAL AND SHIFT GUARDED instruction.

Guarded-Storage-Event Access Information (GSEAI): Byte 3 of the GSEPL contains information describing the following CPU attributes:

Reserved: Bit 0 of the GSEAI is reserved and set to zero when a guarded-storage event is recognized.

DAT Mode (T): Bit 1 of the GSEAI indicates the current dynamic-address-translation mode (that is, the T bit is a copy of PSW bit 5).

Address-Space Indication (AS): Bits 2-3 of the GSEAI indicate the current address-space controls (that is, the AS field is a copy of bits 16-17 of the PSW). The AS field is meaningful only when the DAT is enabled (that is, when the T bit is one); otherwise, the AS field is unpredictable.

Access-Register Number (AR): When the CPU is in the access-register mode, bits 4-7 of the GSEAI indicate the access-register number used by the LGG or LLGFSG instruction causing the event (that is, the AR field is a copy of the B_2 field of the LGG or LLGFSG instruction). When the CPU is not in the access-register mode, the AR field is unpredictable.

Guarded-Storage-Event Handler Address (GSEHA): Bytes 8-15 of the GSEPL contain the guarded-storage-event handler address. The GSEHA field is considered to be a branch address and is subject to the action described in “Formation of the Branch Address” on page 5-13. When a guarded-storage event is recognized, the GSEHA field forms the branch address that is used to complete the execution of the LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED instruction.

A guarded-storage event is considered to be a PER successful-branching event. If the PER branch-address control is one, the GSEHA is the value compared with control registers 10 and 11.

Guarded-Storage-Event Instruction Address (GSEIA): Bytes 16-23 of the GSEPL contain the guarded-storage-event instruction address. When a guarded-storage event is recognized, the address of the instruction causing the event is stored into the GSEIA field. The address placed in the GSEIA is either that of the LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED instruction, or that of the execute-type instruction whose target is a LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED instruction.

Storing of the GSEIA field is subject to the current addressing mode when the event is detected. In the 24-bit addressing mode, bits 0-39 of the GSEIA are set to zeros. In the 31-bit addressing mode, bits 0-32 of the GSEIA are set to zeros.

Guarded-Storage-Event Operand Address (GSEOA): Bytes 24-31 of the GSEPL contain the guarded-storage-event operand address. When a guarded-storage event is recognized, the second-operand address of a LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED instruction causing the event is stored into the GSEOA field.

The content of the GSEOA field is subject to the current addressing mode when the event is detected. In the 24-bit addressing mode, bits 0-39 of the GSEOA are set to zeros. In the 31-bit addressing mode, bits 0-32 of the GSEOA are set to zeros.

If transactional execution is aborted due to the recognition of a guarded-storage event, the GSEOA field contains the operand address formed during transactional execution. This is true even if the operand address was formed using one or more general registers that were altered during transactional execution, and regardless of whether the register(s) were restored when transactional execution was aborted.

Guarded-Storage-Event Intermediate Result (GSEIR): Bytes 32-39 of the GSEPL contain the guarded-storage-event intermediate result. When a guarded-storage event is recognized, the intermediate result formed by a LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED instruction is stored into the GSEIR field.

If the detection of a guarded-storage event causes transactional execution to be aborted, and the model refetches the LGG or LLGFSG second operand to determine if the event persists, the refetching of the second operand and the intermediate-result formation are subject to the addressing mode that was in effect when the event was initially detected.

Engineering Notes:

Guarded-Storage-Event Return Address (GSERA): Bytes 40-47 of the GSEPL contain the guarded-storage-event return address.

When a guarded-storage event is recognized while the CPU is in the transaction-execution mode, the instruction address of the transaction-abort PSW is placed into the GSERA. In the constrained transactional-execution mode, the instruction address designates the TBEGINC instruction. In the nonconstrained transactional-execution mode, the instruction address designates the instruction following the TBEGIN instruction. The GSERA is subject to the addressing mode in the transaction-abort PSW (that is, the addressing mode prior to entering the transactional-execution mode).

When a guarded-storage event is recognized while the CPU is not in the transactional-execution mode, the contents of the GSERA are identical to the GSEIA.

During the execution of LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED, the GSEPL is accessed only if a guarded-storage event is recognized. Multiple accesses may be made to any field of the GSEPL when a guarded-storage event is recognized.

Accesses to the GSEPL during guarded-storage-event processing are considered to be side-effect accesses. Store-type access exceptions are recognized for any byte of the GSEPL including the GSEHA field and reserved fields. If an access exception other than addressing is recognized while accessing the GSEPL, the side-effect-access indication, bit 54 of the translation-exception identification at real location 168-175, is set to one, and the LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED instruction causing the guarded-storage event is nullified.

When DAT is on, the GSEPL is accessed using the current address-space control (ASC) mode, except when the CPU is in the access-register mode; in the access-register mode, the GSEPL is accessed in the primary address space.

Programming Notes:

1. The expected usage is that the program does not switch ASC mode between the establishment of GS controls and the recognition of a GS event. If the program switches ASC mode, then the following apply:

   - The GSEPL should be mapped to common addresses in both the space where it was

established and in the space where the GS event was recognized.

- If a GS event is recognized in the access-register mode, the GS event handler program may need to examine the GSEAI field to determine the appropriate ALET with which to access the guarded-storage operand.

2. When a nonconstrained transaction is aborted due to a guarded-storage event, the addressing mode from the transaction-abort PSW becomes effective. The addressing mode that was in effect at the time of the guarded-storage event can be determined by inspecting the GSEAM field in the GSE parameter list.

   The addressing mode cannot be changed by a constrained transaction; thus, if a constrained transaction is aborted due to a guarded-storage event, the addressing mode is necessarily the same as when the TBEGINC instruction was executed.

Guarded-Storage Facility Operation

Guarded-Storage-Event Detection

Guarded-storage-event detection uses two values formed from the intermediate result of the LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED instruction:

- Guarded-storage-operand comparand (GSOC)
- Guarded-storage-mask index (GSMX)

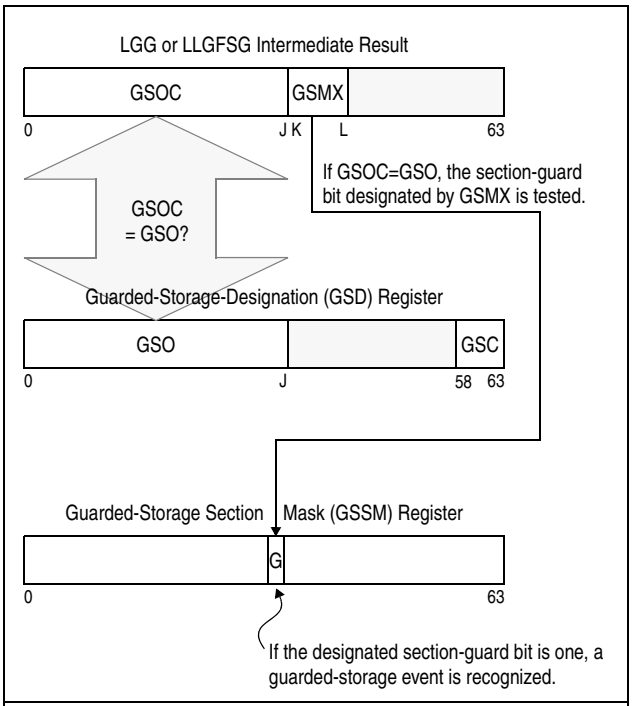
The guarded-storage-operand comparand (GSOC) is formed from the intermediate result of the LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED instruction (see “LOAD GUARDED” on page 7-274 for a description of the intermediate result). Specifically, the GSOC comprises bit positions 0 through (63 - GSC) of the intermediate result, inclusive (where GSC is the guarded-storage characteristic in bit positions 58-63 of the guarded-storage-designation register).

The GSOC is compared with the guarded-storage origin (GSO, in the corresponding bit positions of the GSD register). When the GSOC is not equal to the GSO, a guarded-storage event is not recognized, and the execution of the LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED is completed by placing the intermediate result into general register R_1.

When the GSOC is equal to the GSO, the six bits of the intermediate result to the right of the GSOC form an unsigned binary integer called the guarded-storage mask index (GSMX). The section-guard bit of the guarded-storage-section-mask (GSSM) register corresponding to the GSMX is examined. If the section-guard bit is zero, a guarded-storage event is not recognized, and the execution of the LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED is completed by placing the intermediate result into general register R_1. However, if the section-guard bit is one, then a guarded-storage event is recognized.

Guarded-storage-event detection is not performed when either (a) the guarded-storage facility is not enabled (by means of bit 59 of control register 2), or (b) all bit positions of the guarded-storage section mask (GSSM) register contain zeros.

Detection of the guarded-storage event is illustrated in Figure 4-21.



Explanation:

G	Section-guard bit in the GSSM register.
GSC	Guarded-storage characteristic (bits 58-63 of the GSD register).

Figure 4-21. Guarded-Storage-Event Detection

GSD	Guarded-storage-designation register.
GSMX	Guarded-storage-mask index (bits K through L of the LGG and LLGFSG intermediate result).
GSO	Guarded-storage origin (bits 0 through J of the GSD)
GSOC	Guarded-storage-operand comparand (bits 0 through J of the LGG and LLGFSG intermediate result).
GSSM	Guarded-storage-section-mask register.
J	The rightmost significant bit position of the GSO (that is, bit position 63 minus the value of GSC).
K	Bit position J + 1
L	Bit position J + 6

Figure 4-21. Guarded-Storage-Event Detection

Guarded-Storage-Event Processing

When a guarded-storage event is recognized while the CPU is in the transactional-execution mode, the following occurs:

1. The transaction is aborted with abort code 19, as described in "Transaction Abort Processing" on page 5-102. If the transaction diagnostic block (TDB) address is not valid, or if the TDB address is valid and accessible, condition code 2 is set in the transaction-abort PSW. If the TDB address is valid, but the TDB is not accessible, condition code 1 is set in the transaction-abort PSW.

2. Depending on the model, the second operand of the LOAD GUARDED or LOAD LOGICAL GUARDED AND SHIFT instruction may be refetched to determine whether the guarded-storage-event condition still exists. If the model refetches the second operand, the access to the second operand and the formation of the intermediate result are both subject to the addressing mode that was in effect at the time the event was initially detected.

   - When the second operand is refetched and the guarded-storage-event condition no longer exists, the following applies:

     – Normal transaction-abort processing concludes by the loading of the transaction-abort PSW.

     – Guarded-storage-event processing does not occur. However, the transaction abort code remains 19.

     – PER basic events associated with an LGG or LLGFSG instruction (or an execute-type instruction whose operand is an LGG or LLGFSG) are not recognized.

   - When the second operand is not refetched, or when it is refetched and the guarded-storage-event condition persists, guarded-storage-event processing continues as described below (instead of loading the transaction-abort PSW).

Regardless of whether the CPU was in the transactional-execution mode when a guarded-storage event is recognized, the guarded-storage-event parameter-list-address (GSEPLA) register is used to locate the guarded-storage-event parameter list (GSEPL). The content of the GSEPLA register is a 64-bit address, and 64 bits of the address are used regardless of the current addressing mode. The GSEPL is accessed using the current translation mode, except that when the CPU is in the access-register mode, the GSEPL is accessed using the primary address space.

If an access exception is recognized when accessing the GSEPL, processing is as follows:

- A program interruption occurs.
- If the CPU was not in the transactional-execution mode, then the instruction address in the program-old PSW is set as follows:

  – If the exception condition results in nullification, the instruction address points to the instruction causing the guarded-storage event (that is, the address of the LGG or LLGFSG, or the address of the execute-type instruction whose target is the LGG or LLGFSG).

  – If the exception condition results in suppression or termination, the instruction address points to the next-sequential instruction following the instruction that caused the guarded-storage event.

  If the CPU was in the transactional-execution mode, the transaction-abort PSW is placed into the program-old PSW. The transactional-execution-aborted event indication, bit 6 of the program-interruption code, is set to one.

- For all access-exception conditions except addressing, the side-effect-access indication, bit 54 of the translation-exception identification at real locations 168-175, is set to one. (The TEID is not stored for addressing exceptions.)

- The remaining guarded-storage-event processing, described below, does not occur when the GSEPL is not accessible.

If the GSEPL is accessible, the following actions are performed using the fields of the GSEPL:

- Bytes 0 and 4-7 of the GSEPL are set to zeros.
- An indication of the addressing mode is placed into the guarded-storage-event addressing mode (GSEAM, byte 1 of the GSEPL), as follows:
  - Bits 0-5 of the GSEAM are set to zeros.
  - Bits 6 and 7 of the GSEAM are set to bits 31 and 32 of the PSW at the time the guarded-storage event was recognized.
- An indication of the cause of the event is placed into the guarded-storage-event cause-indication field (GSECI, byte 2 of the GSEPL), as follows:
  - If the CPU was in the transactional-execution mode when the guarded-storage event was recognized, bit 0 of the GSECI is set to one; otherwise, bit 0 of byte 2 is set to zero.
  - If the CPU was in the constrained transactional-execution mode when the guarded-storage event was recognized, bit 1 of the GSECI is set to one; otherwise, bit 1 of the GSECI is set to zero.
  - Bits 2-6 of the GSECI are set to zeros.
  - Bit 7 of the GSECI is set to designate the instruction that caused the guarded-storage event. A value of zero means the event was caused by a LGG instruction; a value of one means the event was caused by a LLGFSG instruction.
- An indication of the PSW DAT, addressing-mode, and address-space controls are placed into the guarded-storage-event access-indication field (GSEAI, byte 3 of the GSEPL), as follows:
  - Bit 0 of the GSEAI is reserved and set to zero.
  - The current translation mode, bit 5 of the PSW, is placed into bit 1 of the GSEAI.
  - If DAT is on, bits 16-17 of the PSW are placed into bits 2-3 of the GSEAI. If DAT is off, bits 2-3 of the GSEAI are unpredictable.
  - If the CPU is in the access-register mode, the access-register number corresponding to the B_2 field of the LGG or LLGFSG instruction causing the event is placed into bits 4-7 of the GSEAI. If the CPU is not in the AR mode, bits 4-7 of the GSEAI are unpredictable.
- The instruction address in the PSW is replaced by the contents of the guarded-storage-event handler-address field (GSEHA, bytes 8-15 of the GSEPL). The GSEHA field is considered to be a branch address and is subject to the action described in “Formation of the Branch Address” on page 5-13. The current addressing mode is unchanged.
- The address of the instruction causing the guarded-storage event is placed into the guarded-storage-event instruction-address field (GSEIA, bytes 16-23 of the GSEPL). The address placed in the GSEIA is either that of the LGG or LLGFSG instruction, or that of the execute-type instruction whose target is a LGG or LLGFSG. The GSEIA is also placed into the breaking-event address register.
- The second-operand address of the LGG or LLGFSG instruction is placed into the guarded-storage-event operand-address (GSEOA, bytes 24-31 of the GSEPL). If transactional execution was aborted due to the recognition of a guarded-storage event, the GSEOA field contains the operand address formed during transactional execution.
- The intermediate result of the LGG or LLGFSG instruction is placed into the guarded-storage-event intermediate-result field (GSEIR, bytes 32-39 of the GSEPL). If transactional execution is aborted due to the recognition of a guarded-storage event, the GSEIR field is formed using the guarded-storage-operand address (GSEOA) field.
- If the guarded-storage event was recognized during transactional execution, it is model dependent whether the GSEIR is formed from the value that was transactionally fetched or the value that was fetched after the transaction was aborted. In either case, the second operand of the LGG or LLGFSG instruction is fetched, and the intermediate result is formed, using the addressing mode that was in effect when the GSE was ini-

tially detected. (See programming note 3 on page page 4-72.)

- If the CPU was in the transactional-execution mode when guarded-storage event was recognized, the instruction address of the transaction-abort PSW is placed in the guarded-storage-event return address field (GSERA, bytes 40-47 of the GSEPL). If the CPU was in the constrained transactional-execution mode, the GSERA designates the TBEGINC instruction. If the CPU was in the nonconstrained transactional-execution mode, the GSERA designates the instruction following the TBEGIN instruction.

  If the CPU was not in the transactional-execution mode when the guarded-storage event was recognized, the content of the GSERA field is identical to that of the GSEIA field.

Finally, the LGG or LLGFSG instruction is considered to have completed without altering general register R_1.

Programming Notes:

1. Programming languages that implement a storage-coalescing technique known as garbage collection may benefit from the guarded-storage facility. In such a programming model, a reference to a program object is performed by first loading a pointer to the object. The LOAD GUARDED and LOAD LOGICAL GUARDED AND SHIFT instructions provide the means by which the program can load a pointer to an object and determine whether the pointer is usable. If no guarded-storage event (GSE) is recognized, the pointer can be used to reference the object. However, if a GSE is recognized, it may indicate that the current pointer designates a storage location that is being reorganized, in which case the object may have been relocated elsewhere. The GSE-handler routine may then modify the pointer to designate the object's new location, and then branch back to location designated by the GSEIA to resume normal program execution.

2. In response to a GSE that is recognized when the CPU is in the transactional-execution mode, the program's GSE handler can attempt to correct the condition that caused the event (that is, update the operand of the LGG or LLGFSG), and then re-execute the transaction by branching to the location designated by the GSERA. If nonconstrained transactional execution was aborted, the program should set the condition code to either 2 or 3 prior to branching to the GSERA, depending on whether the condition causing the event was or was not corrected, respectively. If constrained transactional execution was aborted, then the program should not branch to the location designated by the GSERA unless the condition causing the event has been corrected; otherwise, a program loop may result.

3. On models that refetch the second-operand of the LGG or LLGFSG instruction during GSE processing, the following applies:

   - If another CPU or the I/O subsystem modifies the second-operand location in between GSE detection and GSE processing, a GSE may no longer exist, or a GSE may persist but the guarded-storage-event-intermediate-result (GSEIR) field may not contain the value that caused the event.

   - If the second-operand location initially contains a value that will cause a GSE, and during transactional execution, the program changes the location to a different value that would also cause a GSE, then the transaction is aborted due to GSE detection, the contents of the location revert to their initial value (prior to transactional execution), and the GSEIR will contain the initial value prior to transactional execution, not the value at GSE detection.

   - If the second-operand location initially contains a value that will not cause a GSE, and during transactional execution, the location is changed to a value that causes a GSE, then the transaction is aborted due to GSE detection, the contents of the location revert to their initial value (prior to transactional execution), and the GSE condition no longer exists. For a constrained transaction, or for a nonconstrained transaction where the abort handler does not correct this situation, a program loop will likely result.

   To ensure reliable contents of the guarded-storage-event-intermediate-result (GSEIR) field, a program executing in the transactional-execution mode should use the NONTRANSACTIONAL STORE instruction if it modifies the second-operand location of a LOAD GUARDED instruction

that is subsequently executed in the same transaction.

4. Similar to other instructions that alter the PSW instruction address, a specification exception is recognized if the PSW instruction address (loaded from the GSEHA field) is odd following a guarded-storage event.

5. During GSE processing, the CPU may recognize an access exception when attempting to update the guarded-storage-event parameter list (GSEPL). Such an access exception may be totally innocuous, for example, due to the GSEPL being temporarily paged out to auxiliary storage by the OS. Assuming the OS remedies the exception, it will load the program-old PSW to resume execution of the interrupted program.

   If an access exception is recognized when accessing the GSEPL, and the CPU was not in the transactional-execution mode, the instruction address of the program-old PSW will be set as follows:

   - If the exception resulted in nullification, the instruction address will point to the LGG or LLGFSG instruction that caused the GSE (or the execute-type instruction whose operand was the LGG or LLGFSG).
   - If the exception resulted in suppression or termination, the instruction address will point to the next-sequential instruction following the instruction that caused the GSE for suppressing or terminating exceptions.

   If an access exception is recognized when accessing the GSEPL, and the CPU was in the nonconstrained transactional-execution mode, the program-old PSW will designate the instruction following the outermost TBEGIN; if the CPU was in the constrained transactional-execution mode, the program-old PSW will designate the TBEGINC instruction.

   If the CPU was in the nonconstrained transactional-execution mode and a TDB is stored, abort code 19 indicates that transactional execution was aborted due to a GSE. However, a transaction abort-handler routine cannot assume that abort code 19 necessarily indicates that the GSE-handler routine has corrected the cause of the GSE (because of the possible access-exception condition when accessing the GSEPL). In this scenario, an abort-handler routine may need to re-execute the transaction multiple times to allow for OS resolution of one or more translation exceptions and to allow the GSE handler to correct the cause of the GSE.

Externally Initiated Functions

Resets

Note: Certain externally-initiated functions are described as resetting the CPU into the ESA/390 architectural mode (or simply, ESA/390 mode). When the ESA/390-compatibility-mode facility is installed in the configuration, the same reset operations will reset the CPU into the ESA/390-compatibility mode. In the following discussion of resets and initial-program loading, wherever the phrase *ESA/390 or ESA/390-compatibility mode* is used, it indicates the respective mode in which the configuration was originally established.

Five reset functions are provided:

- CPU reset
- Initial CPU reset
- Subsystem reset
- Clear reset
- Power-on reset

CPU reset provides a means of clearing equipment-check indications and any resultant unpredictability in the CPU state with the least amount of information destroyed. In particular, it is used to clear check conditions when the CPU state is to be preserved for analysis or resumption of the operation. If a CPU reset is caused by the activation of the load-normal or load-with-dump key, (a) if the configuration-z/Architecture-architectural-mode (CZAM) facility is not installed, it sets the architectural mode to the ESA/390 or ESA/390 compatibility mode, and (b) if the multithreading facility is installed and enabled, it is disabled. When CPU reset changes the architectural mode to the ESA/390 or ESA/390 compatibility mode from the z/Architecture architectural mode, it saves the current PSW so that PSW can be restored by a SIGNAL PROCESSOR set-architecture order that changes the architectural mode back to z/Architecture. When CPU reset changes the architectural mode to the ESA/390-compatibility mode from the

z/Architecture architectural mode, it also saves bits 0-31 of the of control registers so that they can be restored by a SIGP set-architecture order that changes the architectural mode back to z/Architecture. See ". CPU Reset" on page 4-77 for details.

Initial CPU reset provides the functions of CPU reset together with initialization of the following:

- Current PSW
- Captured-z/Architecture PSW
- CPU timer
- Clock comparator
- Breaking-event-address register
- Control registers
- Captured z/Architecture control registers (when resetting into the ESA/390-compatibility mode)
- Floating-point-control register
- Prefix register
- TOD programmable register

If the CZAM facility is not installed, initial CPU reset sets the architectural mode to the ESA/390 or ESA/390-compatibility mode if it is caused by activation of the load-normal or load-with-dump key. If multithreading is enabled when initial CPU reset is caused by activation of the load-normal or load-with-dump key, the initial-CPU-reset functions are performed for the lowest-numbered CPU of a core, and CPU reset is performed for all other CPUs in the core.

Subsystem reset provides a means for clearing floating interruption conditions as well as for invoking I/O-system reset.

Clear reset causes initial CPU reset and subsystem reset to be performed and, additionally, clears or initializes all storage locations and registers in all CPUs in the configuration, with the exception of the TOD clock and epoch index. Such clearing is useful in debugging programs and in ensuring user privacy. Clear reset also releases all locks used by the PERFORM LOCKED OPERATION instruction. If the CZAM facility is not installed, clear reset sets the architectural mode to the ESA/390 or ESA/390-compatibility mode. Clearing does not affect external storage, such as direct-access storage devices used by the control program to hold the contents of unaddressable pages.

CPU power-on reset causes initial CPU reset to be performed and clears the contents of general registers, access registers, floating-point registers, and vector registers to zeros with valid checking-block code. Locks used by PERFORM LOCKED OPERATION and associated with the CPU are released unless they are held by a CPU already powered on. The power-on-reset sequences for the TOD clock, epoch index, main storage, and the channel subsystem may be included as part of the CPU power-on sequence, or the power-on sequence for these units may be initiated separately. If CPU power-on reset establishes the configuration, then it sets the architectural mode to one of the following:

- The ESA/390 mode (when neither the ESA/390-compatibility-mode facility nor the CZAM facility are installed)
- The ESA/390 compatibility mode (when the ESA/390 compatibility mode is installed)
- The z/Architecture mode when the CZAM facility is installed

If CPU power-on reset does not establish the configuration, then it sets the architectural mode to that of the CPUs already in the configuration.

CPU reset, initial CPU reset, subsystem reset, and clear reset may be initiated manually by using the operator facilities (see Chapter 12, "Operator Facilities"). Initial CPU reset is part of the initial-program-loading function. Figure 4-22 on page 4-75 summarizes how these four resets are manually initiated. Power-on reset is performed as part of turning power on. The reset actions are tabulated in Figure 4-23 on page 4-75. For information concerning which resets can be performed by the SIGNAL PROCESSOR

instruction, see “Signal-Processor Orders” on page 4-84.

Key Activated	Function Performed on[1]		
	CPU on Which Key Was Activated	Other CPUs in Config.	Remainder of Config.
System-reset-normal key	CPU reset	CPU reset	Subsystem reset
System-reset-clear key	Clear reset[2]	Clear reset[2]	Clear reset[3]
Load-normal or load-with-dump key	Initial CPU reset[4], followed by IPL	CPU reset	Subsystem reset
Load-clear or load-clear-list-directed key	Clear reset[2], followed by IPL	Clear reset[2]	Clear reset[3]

Explanation:

1 Activation of a system-reset or load key may change the configuration, including the connection with I/O, storage units, and other CPUs.
2 Only the CPU elements of this reset apply.
3 Only the non-CPU elements of this reset apply.
4 When multithreading is enabled, initial CPU reset applies only to the lowest-numbered CPU of a core; CPU reset applies to all other CPUs of the core.

Figure 4-22. Manual Initiation of Resets

Area Affected	Reset Function				
	Subsystem Reset	CPU Reset	Initial CPU Reset	Clear Reset	Power-On Reset
CPU	U	S	S[1]	S[1]	S
PSW	U	U/V#	C*[1]	C*[1]	C*[1]
Captured-z/Architecture-PSW register	U	U/sv	C	C	C
Prefix register	U	U/V	C	C	C
CPU timer	U	U/V	C	C	C
Clock comparator	U	U/V	C	C	C
TOD programmable register	U	U/V	C	C	C
Control registers	U	U/V[6]	I	I	I
Captured z/Architecture control registers	U	U/cv	I	I	I
Breaking-event-address register	U	U/V	K	K	K
Floating-point-control register	U	U/V	C	C	C
Access registers	U	U/V	U/V	C	C
General registers	U	U/V	U/V	C	C
Floating-point registers	U	U/V	U/V	C	C
Vector registers	U	U/V	U/V	C	C
Storage keys	U	U	U	C	C[2]
Volatile main storage	U	U	U	C	C[2]
Nonvolatile main storage	U	U	U	C	U
Expanded storage	U[3]	U[3]	U[3]	U[3]	C[2]
Epoch index	U	U	U	U	T[2]
TOD clock	U[4]	U[4]	U[4]	U[4]	T[2]
TOD clock steering registers	U	U	U	U	C
Floating interruption conditions	C	U	U	C	C[2]
I/O system	R	U	U	R	R[5]

Figure 4-23. Summary of Reset Actions (Part 1 of 3)

Area Affected	Reset Function				
	Subsystem Reset	CPU Reset	Initial CPU Reset	Clear Reset	Power-On Reset
PERFORM LOCKED OPERATION locks	U	U	U	RC	RP
Transaction nesting depth	U	C	C	C	C
Guarded-storage registers	U	U/V	U/V	I	I

Explanation:

\# If the architectural mode is changed from z/Architecture to ESA/390 (that is, neither the ESA/390-compatibility-mode facility nor the configuration-z/Architecture-architectural-mode [CZAM] facility is installed, and the reset is due to activation of the load-normal or load-with-dump key on another CPU), the 16-byte PSW first is placed in the captured-z/Architecture-PSW register, and then does not remain unchanged. Instead, it is changed to an eight-byte PSW, and the bits of the eight-byte PSW are set as follows. Bits 0-11, 13-23, and 25-32 are set equal to the same bits of the 16-byte PSW, bit 12 is set to one, bit 24 is set to zero, and bits 33-63 are set equal to bits 97-127 of the 16-byte PSW. The PSW is invalid in the ESA/390 mode if PSW bit 31 is one. It is unpredictable whether the PSW is invalid in the ESA/390-compatibility mode if PSW bit 31 is one.

* Clearing the contents of the PSW to zero causes the PSW to be invalid if the architectural mode is ESA/390.

1 When the IPL sequence follows the reset function on that CPU, the CPU does not necessarily enter the stopped state, and the PSW is not necessarily cleared to zeros.

2 When these units are separately powered, the action is performed only when the power for the unit is turned on.

3 Access to change expanded storage at the time a reset function is performed may cause the contents of the 4 K-byte block in expanded storage to be unpredictable. Access to examine expanded storage does not affect the contents of the expanded storage.

4 Access to the TOD clock by means of STORE CLOCK at the time a reset function is performed does not cause the value of the TOD clock to be affected.

5 When the channel subsystem is separately powered or consists of multiple elements which are separately powered, the reset action is applied only to those subchannels, channel paths, and I/O control units and devices on those paths associated with the element which is being powered on.

6 When CPU reset results in the configuration entering the ESA/390-compatibility mode, bits 0-31 of the control registers are set to zeros.

C The condition or contents are cleared. If the area affected is a field, the contents are set to zero with valid checking-block code.

I The state or contents are initialized. If the area affected is a field, the contents are set to the initial value with valid checking-block code. For guarded-storage controls, all fields are set to zeros.

K The breaking-event-address register is initialized to 0000000000000001 hex with valid checking-block code.

R I/O-system reset is performed in the channel subsystem. As part of this reset, system reset is signaled to all I/O control units and devices attached to the channel subsystem.

RC All locks in the configuration are released.

RP All locks in the configuration are released except for ones held by CPUs already powered on.

S The CPU is reset; current operations, if any, are terminated; the ALB and TLB are cleared of entries; interruption conditions in the CPU are cleared; and the CPU is placed in the stopped state. The effect of performing the start function is unpredictable when the stopped state has been entered by means of a reset. If the reset is initiated by the system-reset-clear, load-normal, load-with-dump, load-clear, or load-clear-list-directed key or by a CPU power-on reset that establishes the configuration, the architectural mode is set to one of (a) the ESA/390 mode (when neither the ESA/390-CM facility nor the CZAM facility is installed), (b) the ESA/390-compatibility mode (when the ESA/390 CM facility is installed), or (c) the z/Architecture mode (if the CZAM facility is installed) and the multithreading facility is disabled. Otherwise, the architectural mode and the state of the multithreading facility are unchanged, except that power-on reset sets the architectural mode to that of the CPUs already in the configuration.

T The epoch index and TOD clock are initialized to the current time of day and validated; the TOD clock enters the set state.

Figure 4-23. Summary of Reset Actions (Part 2 of 3)

Area Affected	Reset Function				
	Subsystem Reset	CPU Reset	Initial CPU Reset	Clear Reset	Power-On Reset
U	The state, condition, or contents of the field remain unchanged. However, the result is unpredictable if an operation is in progress that changes the state, condition, or contents of the field at the time of reset.				
U/cv	When the ESA/390-compatibility-mode is installed: if the reset is due to activation of load-normal or load-with-dump key, the captured z/Architecture control registers are set from bits 0-31 of the current control registers (before the control registers are initialized); otherwise, the captured-z/Architecture-control registers are unchanged.				
U/sv	The captured-z/Architecture-PSW register remains unchanged if the reset is due to activation of the system-reset-normal key or the SIGNAL PROCESSOR CPU-reset order, or it is set with the value of the current 16-byte PSW if the reset is due to activation of the load-normal or load-with-dump key.				
U/V	The contents remain unchanged, provided the field is not being changed at the time the reset function is performed. However, on some models the checking-block code of the contents may be made valid. The result is unpredictable if an operation is in progress that changes the contents of the field at the time of reset.				

Figure 4-23. Summary of Reset Actions (Part 3 of 3)

. CPU Reset

CPU reset causes the following actions:

1. The execution of the current instruction or other processing sequence, such as an interruption, is terminated, and all program-interruption and supervisor-call-interruption conditions are cleared.
2. If the CPU was in the transactional-execution mode, the transaction nesting depth is set to zero, and the CPU leaves the transactional-execution mode.
3. Any pending external-interruption conditions which are local to the CPU are cleared. Floating external-interruption conditions are not cleared.
4. Any pending machine-check-interruption conditions and error indications which are local to the CPU and any check-stop states are cleared. Floating machine-check-interruption conditions are not cleared. Any machine-check condition which is reported to all CPUs in the configuration and which has been made pending to a CPU is said to be local to the CPU.
5. All copies of prefetched instructions or operands are cleared. Additionally, any results to be stored because of the execution of instructions in the current checkpoint interval are cleared.
6. The ART-lookaside buffer and translation-lookaside buffer are cleared of entries.
7. If the reset is caused by activation of the load-normal or load-with-dump key on any CPU in the configuration, the following actions occur:

   a. If the configuration-z/Architecture-architectural-mode (CZAM) facility is not installed, the current PSW is saved in the captured-z/Architecture-PSW register for subsequent use by a SIGNAL PROCESSOR set-architecture order that restores the z/Architecture mode.

   b. When the ESA/390-compatibility-mode is installed, the captured z/Architecture control registers are set from bits 0-31 of the current control registers (before the control registers are initialized).

   c. If the configuration is enabled for multithreading, the following actions occur:

      1) CPU address contraction is performed as described on page 4-84.

      2) Multithreading is disabled.

   d. If the configuration-z/Architecture-architectural-mode (CZAM) facility is not installed, then the following occurs:

      - The architectural mode of the CPU (and of all other CPUs in the configuration because of the initial CPU reset or CPU resets performed by them) is changed from the z/Architecture mode to the ESA/390 or ESA/390-compatibility mode.

- The current PSW is changed from 16 bytes to eight bytes. The bits of the eight-byte PSW are set as follows: bits 0-11 and 13-32 are set equal to the same bits of the 16-byte PSW, bit 12 is set to one, and bits 33-63 are set equal to bits 97-127 of the 16-byte PSW.

  If the resulting architectural mode is the ESA/390-compatibility mode, bits 0-31 of all control registers are set to zeros.

A CPU reset caused by activation of the system-reset-normal key or by the SIGNAL PROCESSOR CPU-reset order, and any CPU reset in the ESA/390 or ESA/390-compatibility mode, (a) do not change the architectural mode, and (b) do not affect the captured-z/Architecture-PSW register, the captured-z/Architecture control registers (when the ESA/390-compatibility-mode is installed), or the enablement of the multithreading facility.

8. The CPU is placed in the stopped state after actions 1-7 have been completed. When the CCW-type IPL sequence follows the reset function on that CPU, the CPU enters the load state at the completion of the reset function and does not necessarily enter the stopped state during the execution of the reset operation. When the list-directed IPL sequence follows the reset function on that CPU, the CPU enters the operating state and does not necessarily enter the stopped state during the execution of the reset operation.

Registers, storage contents, and the state of conditions external to the CPU remain unchanged by CPU reset. However, the subsequent contents of the register, location, or state are unpredictable if an operation is in progress that changes the contents at the time of the reset. A lock held by the CPU when executing PERFORM LOCKED OPERATION is not released by CPU reset.

When the reset function in the CPU is initiated at the time the CPU is executing an I/O instruction or is performing an I/O interruption, the current operation between the CPU and the channel subsystem may or may not be completed, and the resultant state of the associated channel-subsystem facility may be unpredictable.

Programming Notes:

1. Most operations which would change a state, a condition, or the contents of a field cannot occur when the CPU is in the stopped state. However, some signal-processor functions and some operator functions may change these fields. To eliminate the possibility of losing a field when CPU reset is issued, the CPU should be stopped, and no operator functions should be in progress.

2. If the architectural mode is changed to the ESA/390 or ESA/390-compatibility mode and bit 31 of the current PSW is one, the PSW is invalid.

Initial CPU Reset

Initial CPU reset combines the CPU reset functions with the following clearing and initializing functions:

1. If the reset is caused by activation of the load-normal or load-with-dump key, then the following is performed:

   a. If the multithreading facility is enabled, CPU address contraction is performed and the multithreading facility is disabled.

   b. If the configuration-z/Architecture-architectural-mode (CZAM) facility is not installed, the architectural mode of the CPU (and of all other CPUs in the configuration) is set to the ESA/390 or ESA/390-compatibility mode.

2. The contents of the current PSW, captured-z/Architecture-PSW, prefix, CPU timer, clock comparator, and TOD programmable register are set to zero. When the IPL sequence follows the reset function on that CPU, the contents of the PSW are not necessarily set to zero.

3. The contents of the control registers are set to their initial z/Architecture values. All 64 bits of the control registers are set regardless of whether the CPU is in the ESA/390, ESA/390-compatibility, or the z/Architecture architectural mode. When the ESA/390-compatibility-mode facility is installed, the contents of the captured-z/Architecture-control registers are set to bits 0-31 of their respective control registers' initial z/Architecture values.

4. The contents of the floating-point-control register are set to zero.

5. The contents of the breaking-event-address register are initialized to 0000000000000001 hex.

These clearing and initializing functions include validation.

Setting the current PSW to zero when the CPU is in the ESA/390 or ESA/390-compatibility architectural mode at the end of the operation causes the PSW to be invalid, since PSW bit 12 must be one in that mode. Thus, in this case if the CPU is placed in the operating state after a reset without first introducing a new PSW, a specification exception is recognized.

Programming Note: When the multithreading facility is installed and enabled, an initial-CPU reset that is caused by the execution of a SIGP initial-CPU-reset order does not cause multithreading to be disabled.

Subsystem Reset

Subsystem reset operates only on those elements in the configuration which are not CPUs. It performs the following actions:

1. I/O-system reset is performed by the channel subsystem (see "I/O-System Reset" on page 17-13).
2. All floating interruption conditions in the configuration are cleared.
3. The warning-track-interruption facility is unregistered.

As part of I/O-system reset, pending I/O-interruption conditions are cleared, and system reset is signaled to all control units and devices attached to the channel subsystem (see "I/O-System Reset" on page 17-13). The effect of system reset on I/O control units and devices and the resultant control-unit and device state are described in the appropriate System Library publication for the control unit or device. A system reset, in general, resets only those functions in a shared control unit or device that are associated with the particular channel path signaling the reset.

Clear Reset

Clear reset combines the initial-CPU-reset function with an initializing function which causes the following actions:

1. If the multithreading facility is enabled, CPU address contraction is performed and the multithreading facility is disabled.
2. If the configuration-z/Architecture-architectural-mode (CZAM) facility is not installed, the architectural mode of all CPUs in the configuration is set to the ESA/390 or ESA/390-compatibility mode.
3. The access, general, floating-point, and vector registers of all CPUs in the configuration are set to zero. All 64 bits of the general registers and all 128 bits of the vector registers are set to zero regardless of whether the CPU was in the ESA/390, ESA/390-compatibility, or z/Architecture architectural mode when the clear-reset function was initiated.
4. The contents of the main storage in the configuration and the associated storage keys are set to zero with valid checking-block code.
5. The locks used by any CPU in the configuration when executing the PERFORM LOCKED OPERATION instruction are released.
6. A subsystem reset is performed.
7. A new set of wrapping keys and their associated verification patterns are generated.
8. The guarded-storage-facility registers are set to zero.

Validation is included in setting registers and in clearing storage and storage keys.

Programming Notes:

1. The architectural mode is not changed by activation of the system-reset-normal key or by execution of a SIGNAL PROCESSOR CPU-reset or initial-CPU-reset order. All CPUs in the configuration are always in the same architectural mode.
2. For the CPU-reset operation not to affect the contents of fields that are to be left unchanged, the CPU must not be executing instructions and must be disabled for all interruptions at the time of the reset. Except for the operation of the CPU timer and for the possibility of a machine-check interruption occurring, all CPU activity can be stopped by placing the CPU in the wait state and by disabling it for I/O and external interruptions. To avoid the possibility of causing a reset at the time that the CPU timer is being updated or a machine-check interruption occurs, the CPU must be in the stopped state.

3. CPU reset, initial CPU reset, subsystem reset, and clear reset do not affect the value and state of the TOD clock and epoch index.

4. The conditions under which the CPU enters the check-stop state are model-dependent and include malfunctions that preclude the completion of the current operation. Hence, if CPU reset or initial CPU reset is executed while the CPU is in the check-stop state, the contents of the PSW, registers, and storage locations, including the storage keys and the storage location accessed at the time of the error, may have unpredictable values, and, in some cases, the contents may still be in error after the check-stop state is cleared by these resets. In this situation, a clear reset is required to clear the error.

Power-On Reset

The power-on-reset function for a component of the machine is performed as part of the power-on sequence for that component.

The power-on sequences for the TOD clock, epoch index, main storage, expanded storage, and channel subsystem may be included as part of the CPU power-on sequence, or the power-on sequence for these units may be initiated separately. The following sections describe the power-on resets for the CPU, TOD clock, epoch index, TOD-clock-steering registers, main storage, expanded storage, and channel subsystem. See also "I/O Support Functions" on page 17-1, and the appropriate System Library publication for the channel subsystem, control units, and I/O devices.

CPU Power-On Reset: The power-on reset causes initial CPU reset to be performed and may or may not cause I/O-system reset to be performed in the channel subsystem. The contents of general registers, access registers, floating-point registers, and vector registers are cleared to zeros with valid checking-block code. Locks used by PERFORM LOCKED OPERATION and associated with the CPU are released unless they are held by a CPU already powered on.

If CPU power-on reset establishes the configuration, then it sets the architectural mode to one of the following:

- The ESA/390 mode (when neither the ESA/390-compatibility-mode facility nor the CZAM facility is installed)
- The ESA/390-compatibility mode (when the ESA/390-compatibility-mode facility is installed)
- The z/Architecture architectural mode (when the CZAM facility is installed)

If CPU power-on reset does not establish the configuration, then it sets the architectural mode to that of the CPUs already in the configuration, and the CPU address is expanded if the multithreading facility is already enabled in the configuration.

TOD-Clock and Epoch Index Power-On Reset: The power-on reset causes the value of the TOD clock and epoch index to be set to the current time of day with valid checking-block code and causes the clock to enter the set state.

When the TOD-clock-steering facility is installed, the TOD clock is never reported to be in the not-set state, as the TOD clock is placed in the set state with meaningful values as part of the initial-machine-loading (IML) process.

TOD-Clock-Steering-Registers Power-On Reset: The power-on reset causes the value of the TOD-clock-steering registers to be set to zero with valid checking-block code.

Main-Storage Power-On Reset: For volatile main storage (one that does not preserve its contents when power is off) and for storage keys, power-on reset causes zeros with valid checking-block code to be placed in these fields. The contents of nonvolatile main storage, including the checking-block code, remain unchanged.

Expanded-Storage Power-On Reset: The contents of expanded storage are cleared to zeros with valid checking-block code.

Channel-Subsystem Power-On Reset: The channel-subsystem power-on reset causes I/O-system reset to be performed in the channel subsystem. (See "I/O-System Reset" on page 17-13.)

Operation Note: Functions equivalent to a power-on reset are performed when a configuration is activated in a logical partition or virtual machine.

Initial Program Loading

Initial program loading (IPL) provides a manual means for causing a program to be read from a designated device and for initiating execution of that program.

Some models may provide additional controls and indications relating to IPL; this additional information is specified in the System Library publication for the model.

There are two types of IPL: CCW-type IPL and list-directed IPL. CCW-type IPL is provided by all machine configurations. List-directed IPL may be provided, depending on the model.

CCW-type IPL is described below. List-directed IPL is described in "List-Directed IPL" on page 17-19.

CCW-Type IPL

CCW-type IPL is initiated manually by setting the load-unit-address controls to a four-digit number to designate an input device and by subsequently activating the load-clear or load-normal key. On some models, load-normal and load-clear keys are provided for each primary CPU in the configuration. On other models, a single load-normal and load-clear key are provided for the entire configuration, and apply to the lowest-numbered primary online CPU that is not in the check-stop state. In the description which follows, the term "this CPU" refers to either (a) the CPU in the configuration for which the load-clear load-normal key was activated or (b) the lowest-numbered online primary CPU (as described above). When the multithreading facility is enabled, "this CPU" refers to the CPU which results from CPU address contraction ("CPU-Address Contraction" on page 4-84).

Activating the load-clear key causes a clear reset to be performed on the configuration.

Activating the load-normal key causes an initial CPU reset to be performed on this CPU, a CPU reset to be propagated to all other CPUs in the configuration, and a subsystem reset to be performed on the remainder of the configuration.

When the configuration-z/Architecture-architectural-mode (CZAM) facility is not installed, activating the load-clear key or the load-normal key sets the architectural mode to the ESA/390 or ESA/390-compatibility mode.

In the loading part of the operation, after the resets have been performed, this CPU then enters the load state. This CPU does not necessarily enter the stopped state during the execution of the reset operations. The load indicator is on while the CPU is in the load state.

Subsequently, a channel-program read operation is initiated from the I/O device designated by the load-unit-address controls. The effect of executing the channel program is as if a format-0 CCW beginning at absolute storage location 0 specified a read command with the modifier bits zeros, a data address of zero, a byte count of 24, the chain-command and SLI flags ones, and all other flags zeros.

The details of the channel-subsystem portion of the IPL operation are defined in "Initial Program Loading" on page 17-16.

When the IPL I/O operation is completed successfully, the subsystem-identification word for the IPL device is stored in absolute storage locations 184-187, zeros are stored in absolute storage locations 188-191, and a new PSW is loaded from absolute storage locations 0-7. If the PSW loading is successful and no machine malfunctions are detected, this CPU leaves the load state, and the load indicator is turned off. If the rate control is set to the process position, the CPU enters the operating state, and the CPU operation proceeds under control of the new PSW. If the rate control is set to the instruction-step position, the CPU enters the stopped state, with the manual indicator on, after the new PSW is loaded.

If the IPL I/O operation or the PSW loading is not completed successfully, the CPU remains in the load state, and the load indicator remains on. The contents of absolute storage locations 0-7 are unpredictable.

Store Status

The store-status operation places an architectural-mode identification and the contents of the CPU registers, except for the TOD clock, in assigned storage locations.

Figure 4-24 lists the fields that are stored when the CPU is in the z/Architecture architectural mode, their length, and their locations in main storage.

Field	Length in Bytes	Absolute Address
Architectural-mode id	1	163
Floating-point registers 0-15	128	4608
General registers 0-15	128	4736
Current PSW	16	4864
Prefix	4	4888
Floating-point control register	4	4892
TOD programmable register	4	4900
CPU timer	8	4904
Zeros	1	4912
Bits 0-55 of clock comparator	7	4913
Access registers 0-15	64	4928
Control registers 0-15	128	4992

Figure 4-24. Assigned Storage Locations for Store Status in the z/Architecture Architectural Mode

Figure 4-25 lists the fields that are stored when the CPU is in the ESA/390-Compatibility mode, their lengths, and their locations in main storage.

Field	Length in Bytes	Absolute Address
Architectural-mode id	1	163
CPU timer	8	216
Clock comparator	8	224
Current PSW	8	256
Prefix	4	264
Access registers 0-15	64	288
Floating-point registers 0, 2, 4, 6	32	352
General registers 0-15 (bits 32-63)	64	384
Control registers 0-15 (bits 32-63)	64	448

Figure 4-25. Assigned Storage Locations for Store Status in the ESA/390-Compatibility Mode

In the ESA/390-compatibility mode, when the extended-save-area control, bit 34 of control register 14, is one, and bits 1-19 of the word at absolute locations 212-215 are not all zeros, then other fields are stored in a store-status extended save area. Figure 4-12 lists the fields that are stored, their lengths, and their offsets within the area. Bytes 144-4095 of the extended save area remain unchanged.

During the execution of the store-status operation, the store-status architectural-mode identification is stored at absolute location 163, as follows:

Field	Length in Bytes	Byte Offset
Floating-point registers 0-15	128	0
Floating-point control register	4	128
Reserved (stored as zeros)	12	132
Unchanged	3,952	144

Figure 4-26. Store-Status Extended Save Area

- Zeros are stored in bit positions 0-6.
- When the CPU is in the ESA/390 architectural mode or ESA/390-compatibility mode, a zero is stored in bit position 7.
- When the CPU is in the z/Architecture architectural mode, a one is stored in bit position 7. When the configuration-z/Architecture-architectural-mode (CZAM) facility is installed, a one is always stored in bit position 7.

When the CPU is in the z/Architecture architectural mode, bits 0-55 of the clock comparator are stored beginning at absolute location 4913, and zeros are stored at absolute location 4912. When the CPU is in the ESA/390-compatibility mode, the entire 64-bit clock comparator is stored beginning at absolute location 224.

The contents of the registers are not changed. If an error is encountered during the operation, the CPU enters the check-stop state.

The store-status operation can be initiated manually by use of the store-status key (see “Store-Status Key” on page 12-5). The store-status operation can also be initiated at the addressed CPU by executing SIGNAL PROCESSOR, specifying the stop-and-store-status order. Execution of SIGNAL PROCESSOR specifying the store-status-at-address order permits the same status information, except for the store-status architectural-mode identification, to be stored at a designated address (see “Signal-Processor Orders” on page 4-84).

Multiprocessing

The multiprocessing facility provides for the interconnection of CPUs, via a common main storage, in order to enhance system availability and to share

data and resources. The multiprocessing facility includes the following facilities:

- Shared main storage
- CPU-to-CPU interconnection
- TOD-clock synchronization

Associated with these facilities is an external-interruption condition (malfunction alert), which is described in "Malfunction Alert" on page 6-13; and control-register positions for the TOD-clock-sync-control bit and for the mask for the external-interruption condition, which are listed in "Control Registers" on page 4-8.

The channel subsystem, including all subchannels, in a multiprocessing configuration can be accessed by all CPUs in the configuration. I/O-interruption conditions are floating and can be accepted by any CPU in the configuration.

Shared Main Storage

The shared-main-storage facility permits more than one CPU to have access to common main-storage locations. All CPUs having access to a common main-storage location have access to the entire 4 K-byte block containing that location and to the associated storage key. The channel subsystem and all CPUs in the configuration refer to a shared main-storage location using the same absolute address.

CPU-Address Identification

Each CPU has a number assigned, called its CPU address. A CPU address uniquely identifies one CPU within a configuration. The CPU is designated by specifying this address in the CPU-address field of SIGNAL PROCESSOR. The CPU signaling a malfunction alert, emergency signal, or external call is identified by storing this address in the CPU-address field with the interruption. The CPU address is assigned by the configuration-definition process and, except as indicated below, is not changed as a result of reconfiguration changes. The program can determine the address of the CPU by using STORE CPU ADDRESS.

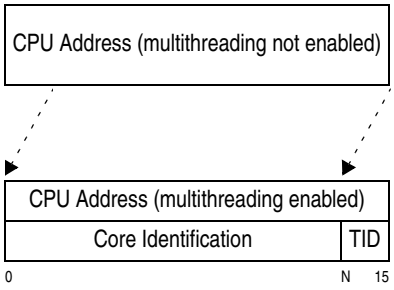
CPU-Address Expansion

When multithreading is enabled, the CPU address comprises a core identification (core ID), concatenated with an identification of a CPU within the core; the CPU identification within a core is commonly called a thread identification (thread ID, or TID). Within a configuration, all cores provide the same number of CPUs; however, depending on the model and CPU type, some CPUs in a core may not be operational.

Based on the program-specified maximum thread identification specified in bits 59-63 of the parameter register used by the SIGNAL PROCESSOR set-multithreading order, a fixed number of bits are required to represent the thread identification; this number of bits is referred to as the *TID width*.

The core ID is formed from the rightmost bits of the CPU address before multithreading is enabled. The core ID is shifted left by TID-width bits, resulting in the leftmost bits of the CPU address after multithreading is available. The thread ID requires the same TID-width number of bits, and occupies the rightmost bits of the CPU address after multithreading is enabled. Thread IDs are assigned in a contiguous range of numbers beginning with zero and extending to a maximum of 31; however, a model may implement fewer CPUs per core.

Figure 4-27 illustrates the adjustment of the CPU address when multithreading is enabled.



Explanation:

N Leftmost bit position of the thread ID; also, number of bits in the core ID. N = 16 – (TID width). See Figure 4-28 for details.

TID Thread identification

Figure 4-27. Adjustment of the CPU Address when Multithreading is Enabled.

Figure 4-28 illustrates the relationship of the program-specified maximum thread identification, the

TID width and the CPU-address bits comprising the core identification and thread identification.

Program-Specified Maximum TID[1]	TID Width	CPU Address Bits	
		Core ID	Thread ID
0	0	0-15	–
1	1	0-14	15
2-3	2	0-13	14-15
4-7	3	0-12	13-15
8-15	4	0-11	12-15
16-31	5	0-10	11-15

Explanation:

1 Specified in bits 56-63 of the parameter register for the SIGP set-multithreading order. The number of CPUs per core is one more than the program-specified maximum thread identification.

– Not applicable; multithreading is not enabled.

Figure 4-28. Program-Specified Maximum TID, TID Width, Core ID and Thread ID Fields of the CPU Address

CPU-Address Contraction

When a reset function disables multithreading, (a) the CPU address(es) of the CPU(s) having the thread-ID zero are shifted to the right by the same TID-width number of bits used during enablement, (b) zeros are inserted in the TID-width number of bits on the left of the address, and (c) the CPU address reverts to its original non-multithreading format. All CPUs in a core having nonzero thread IDs when multithreading is enabled are no longer operational when multithreading is disabled.

When multithreading is not enabled, the CPU address remains unchanged from the value assigned by the configuration-definition process. In this case, the thread identification does not exist.

CPU Signaling and Response

The CPU-signaling-and-response facility consists of SIGNAL PROCESSOR and a mechanism to interpret and act on several order codes. The facility provides for communications among CPUs, including transmitting, receiving, and decoding a set of assigned order codes; initiating the specified operation; and responding to the signaling CPU. A CPU can address SIGNAL PROCESSOR to itself. SIGNAL PROCESSOR is described in "SIGNAL PROCESSOR" on page 10-133.

Signal-Processor Orders

Note: Certain SIGP orders are described as resetting the CPU into the ESA/390 architectural mode (or simply, ESA/390 mode). When the ESA/390-compatibility-mode facility is installed in the configuration, the same reset operations will reset the CPU into the ESA/390-compatibility mode. In the following discussion of SIGP orders, wherever the phrase ESA/390 or ESA/390-compatibility mode is used, it indicates the respective mode in which the configuration was originally activated.

The signal-processor orders are specified in bit positions 56-63 of the second-operand address of SIGNAL PROCESSOR and are encoded as shown in Figure 4-29.

Code		Order
(Dec)	(Hex)	
0	00	Unassigned
1	01	Sense
2	02	External call
3	03	Emergency signal
4	04	Start
5	05	Stop
6	06	Restart
7	07	Unassigned
8	08	Unassigned
9	09	Stop and store status
10	0A	Unassigned
11	0B	Initial CPU reset
12	0C	CPU reset
13	0D	Set prefix
14	0E	Store status at address
15-16	0F-10	Unassigned
17	11	Store extended status at address (ESA/390-compatibility mode only)
18	12	Set architecture
19	13	Conditional Emergency Signal (z/Architecture only)
20	14	
21	15	Sense Running Status (z/Architecture only)
22	16	Set multithreading (z/Architecture only)
23	17	Store additional status at address (z/Architecture only)
24-255	17-FF	Unassigned

Figure 4-29. Encoding of SIGNAL PROCESSOR Orders

The orders are defined as follows:

Sense

The addressed CPU presents its status to the issuing CPU (see "Status Bits" on page 4-94 for a definition of the bits). No other action is caused at the addressed CPU. The status, if not all zeros, is stored in the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction, and condition code 1 is set; if all status bits are zeros, condition code 0 is set.

External Call

An external-call external-interruption condition is generated at the addressed CPU. The interruption condition becomes pending during the execution of SIGNAL PROCESSOR. The associated interruption occurs when the CPU is enabled for that condition and does not necessarily occur during the execution of SIGNAL PROCESSOR. The address of the CPU sending the signal is provided with the interruption code when the interruption occurs. Only one external-call condition can be kept pending in a CPU at a time. The order is accepted only when the addressed CPU is in the stopped or the operating state.

Emergency Signal

An emergency-signal external-interruption condition is generated at the addressed CPU. The interruption condition becomes pending during the execution of SIGNAL PROCESSOR. The associated interruption occurs when the CPU is enabled for that condition and does not necessarily occur during the execution of SIGNAL PROCESSOR. The address of the CPU sending the signal is provided with the interruption code when the interruption occurs. At any one time the receiving CPU can keep pending one emergency-signal condition for each CPU in the configuration, including the receiving CPU itself. The order is accepted only when the addressed CPU is in the stopped or the operating state.

Start

The addressed CPU performs the start function (see "CPU States" on page 4-2). The CPU does not necessarily enter the operating state during the execution of SIGNAL PROCESSOR. The order is effective only when the addressed CPU is in the stopped state. The effect of performing the start function is unpredictable when the stopped state has been entered by reset.

Stop

The addressed CPU performs the stop function (see "CPU States" on page 4-2). The CPU does not necessarily enter the stopped state during the execution of SIGNAL PROCESSOR. The order is effective only when the CPU is in the operating state.

Restart

The addressed CPU performs the restart operation (see "Restart Interruption" on page 6-56). The CPU does not necessarily perform the operation during the execution of SIGNAL PROCESSOR. The order is effective only when the addressed CPU is in the stopped or the operating state.

Stop and Store Status

The addressed CPU performs the stop function, followed by the store-status operation (see "Store Status" on page 4-81). The CPU does not necessarily complete the operation, or even enter the stopped state, during the execution of SIGNAL PROCESSOR. The order is effective only when the addressed CPU is in the stopped or the operating state.

Initial CPU Reset

The addressed CPU performs initial CPU reset (see "Resets" on page 4-73). The execution of an initial-CPU reset initiated by SIGNAL PROCESSOR does not affect the architectural mode or other CPUs, does not disable multithreading, and does not cause I/O to be reset. The reset operation is not necessarily completed during the execution of SIGNAL PROCESSOR.

CPU Reset

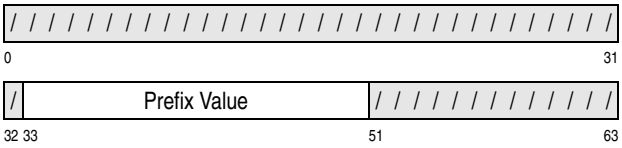
The addressed CPU performs CPU reset (see "Resets" on page 4-73). The execution of a CPU reset initiated by SIGNAL PROCESSOR does not affect the architectural mode or other CPUs, does not disable multithreading, and does not cause I/O to be reset. The reset operation is not necessarily completed during the execution of SIGNAL PROCESSOR.

Set Prefix

The contents of bit positions 33-50 of the parameter register of the SIGNAL PROCESSOR instruction are treated as a prefix value, which replaces bits 33-50 of the prefix register of the addressed CPU. Bits 0-32 and 51-63 of the parameter register are ignored. The order is accepted only if the addressed CPU is in the stopped state, the value to be placed in the prefix

register designates an 8K block which is available in the configuration, and no other condition precludes accepting the order. Verification of the stopped state of the addressed CPU and of the availability of the designated storage is performed during execution of SIGNAL PROCESSOR. If accepted, the order is not necessarily completed during the execution of SIGNAL PROCESSOR.

The parameter register has the following format:



The set-prefix order is completed as follows:

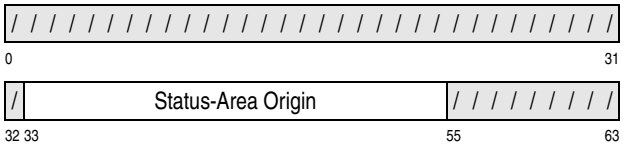
- If the addressed CPU is not in the stopped state, the order is not accepted. Instead, bit 54 (incorrect state) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- The value to be placed in the prefix register of the addressed CPU is tested for the availability of the designated storage. The absolute address of an 8 K-byte area of storage is formed by appending 13 zeros to the right and 33 zeros to the left of bits 33-50 of the parameter value. This address is treated as a 64-bit absolute address regardless of whether the sending and receiving CPUs are in the 24-bit, 31-bit, or 64-bit addressing mode. The two 4 K-byte blocks of storage within the new prefix area are accessed. The accesses to the blocks are not subject to protection, and the associated reference bits may or may not be set to one. If either block is not available in the configuration, the order is not accepted by the addressed CPU, bit 55 (invalid parameter) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- The value is placed in the prefix register of the addressed CPU.
- The ALB and TLB of the addressed CPU are cleared of their contents.
- A serializing and checkpoint-synchronizing function is performed on the addressed CPU following insertion of the new prefix value.

Store Status at Address

The contents of bit positions 33-54 of the parameter register of the SIGNAL PROCESSOR instruction are used as the origin of a 512-byte area on a 512-byte boundary in absolute storage into which the status of the addressed CPU is stored. Bits 0-32 and 55-63 of the parameter register are ignored.

The order is accepted only if the addressed CPU is in the stopped state, the status-area origin designates a location which is available in the configuration, and no other condition precludes accepting the order. Verification of the stopped state of the addressed CPU and of the availability of the designated storage is performed during execution of SIGNAL PROCESSOR. If accepted, the order is not necessarily completed during the execution of SIGNAL PROCESSOR.

The parameter register has the following format:



The store-status-at-address order is completed as follows:

- If the addressed CPU is not in the stopped state, the order is not accepted. Instead, bit 54 (incorrect state) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- The address of the area into which status is to be stored is tested for availability. The absolute address of a 512-byte area of storage is formed by appending 9 zeros to the right and 33 zeros to the left of bits 33-54 of the parameter value. This address is treated as a 64-bit absolute address regardless of whether the sending and receiving CPUs are in the 24-bit, 31-bit, or 64-bit addressing mode. The 512-byte block of storage at this address is accessed. The access is not subject to protection, and the associated reference bit may or may not be set to one. If the block is not available in the configuration, the order is not accepted by the addressed CPU, bit 55 (invalid parameter) of the general register designated by the R_1 field of the SIGNAL PROCESSOR

instruction is set to one, and condition code 1 is set.

- The status of the addressed CPU is placed in the designated area. The information stored, and the format of the area receiving the information, are the same as for the stop-and-store-status order, except that each field, rather than being stored at an offset from the beginning of absolute storage, is stored in the designated area at the offsets listed in the figures below, except that an architectural-mode identification and the vector registers are not stored. Figure 4-30 illustrates the status fields in the z/Architecture architectural mode. Bytes 288-291 and 312-319 of the designated area remain unchanged.

Field	Length in Bytes	Offset in Bytes
Floating-point registers 0-15	128	0
General registers 0-15	128	128
Current PSW	16	256
Prefix	4	280
Floating-point-control register	4	284
TOD programmable register	4	292
CPU timer	8	296
Zeros	1	304
Bits 0-55 of clock comparator	7	305
Access registers 0-15	64	320
Control registers 0-15	128	384

Figure 4-30. Location of Status Fields in Designated Area in the z/Architecture Architectural Mode

Figure 4-30 illustrates the status fields in the ESA/390-compatibility mode.

Field	Length in Bytes	Offset in Bytes
CPU timer	8	216
Clock comparator	8	224
Current PSW	8	256
Prefix	4	264
Access registers 0-15	64	288
Floating-point registers 0, 2, 4, 6	32	352
General registers 0-15 (bits 32-63)	64	384
Control registers 0-15 (bits 32-63)	64	448

Figure 4-31. Location of Status Fields in Designated Area in the ESA/390-compatibility Mode

- A serialization and checkpoint-synchronization function is performed on the addressed CPU following storing of the status.

Programming Note: The architectural mode of the CPU that stored status in a designated area normally is indicated by bit 12 of the PSW stored at offset 256 in the area. The PSW is stored at the same offset, 256, in the ESA/390 mode, ESA/390-compatibility mode, and the z/Architecture mode. Bit 12 is one in an ESA/390 or ESA/390-compatibility mode PSW and zero in a z/Architecture PSW. The store-status-at-address order does not store the architectural-mode identification that is stored at absolute location 163 by the store-status operation and the stop-and-store-status order.

Store Extended Status at Address

The contents of bit positions 33-54 of the parameter register of the SIGNAL PROCESSOR instruction are used as the origin of a 512-byte save area. Bits 0-32 and 55-63 of the parameter register are ignored. The contents of bit positions 1-19 of bytes 212-215 of the save area are used as the origin of a 4,096-byte extended save area. Bits 0 and 20-31 of bytes 212-215 are ignored.

Status of the addressed CPU is stored in the designated save area and extended save area.

The order is accepted only if the CPU is in the ESA/390-compatibility mode, the addressed CPU is in the stopped state, the save-area and extended-save-area origins designate locations that are available in the configuration, the extended-save-area origin is not 0, and no other condition precludes accepting the order. Verification of the ESA/390-compatibility mode, the stopped state of the addressed CPU, and the availability of the designated storage is performed during the execution of SIGNAL PROCESSOR. If accepted, the order is not necessarily completed during the execution of SIGNAL PROCESSOR.

The parameter register has the following format:

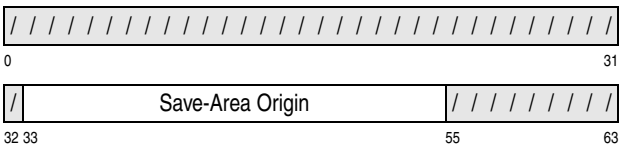


Figure 4-32 lists the fields in the save area, their lengths, and their offsets from the beginning of the

area. The field in byte positions 212-215 is provided by the program. The other fields are stored during the execution of the operation specified by the order.

Field	Length in Bytes	Offset in Bytes
Extended-save-area address	4	212
CPU timer	8	216
Clock comparator	8	224
Current PSW	8	256
Prefix	4	264
Access registers 0-15	64	288
Floating-point registers 0, 2, 4, 6	32	352
General registers 0-15 (bits 32-63)	64	384
Control registers 0-15 (bits 32-63)	64	448

Figure 4-32. Save-Area Locations for Store-Extended-Status-at-Address Order

The extended-save-area address in bytes 212-215 of the save area has the following format:

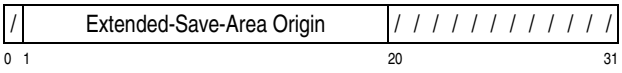


Figure 4-33 lists the fields that are stored in the extended-save area, their lengths, and their offsets from the origin of the area.

Field	Length in Bytes	Byte Offset
Floating-point registers 0-15	128	0
Floating-point control register	4	128
Reserved (stored as zeros)	12	132
Unchanged	3,952	144

Figure 4-33. Extended-Save Area Locations for Store-Extended-Status-at-Address Order.

The store-extended-status-at-address order is completed as follows.

- If the ESA/390-compatibility-mode facility is not installed, the order is not accepted. Instead, bit 62 (invalid order) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- If the addressed CPU is not in the stopped state, the order is not accepted. Instead, bit 54 (incorrect state) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- The save area is tested for the availability of the designated storage. The absolute address of the save area is formed by appending 33 zeros to the left and nine zeros to the right of bits 33-54 of the parameter value. This address is treated as a 64-bit absolute address regardless of the addressing modes of the sending and receiving CPUs. The 512-byte block of storage at this address is accessed. The access is not subject to protection, and the associated reference bit may or may not be set to one. If the block is not available in the configuration, the order is not accepted by the addressed CPU, bit 55 (invalid parameter) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- The extended save area is tested for having a nonzero address and for the availability of the designated storage. The absolute address of the extended save area is formed by appending 33 zeros to the left and 12 zeros to the right of bits 1-19 of bytes 212-215 of the save area. This address is treated as a 64-bit absolute address regardless of the addressing modes of the sending and receiving CPUs. If the address is not zero, the 4,096-byte block of storage at the address is accessed. The access is not subject to protection, and the associated reference bit may or may not be set to one. If the address is zero or the block is not available in the configuration, the order is not accepted by the addressed CPU, bit 55 (invalid parameter) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- Status of the addressed CPU is stored in the save area, as indicated in Figure 4-32 on page 4-88, and in the extended save area as indicated in Figure 4-33 on page 4-88. Bytes 0-211, 232-255, and 268-287 of the save area and bytes 144-4095 of the extended save area remain unchanged.
- A serialization and checkpoint-synchronization function is performed on the addressed CPU following storing of the status.

Set Architecture

When the configuration-z/Architecture-architectural-mode (CZAM) facility is not installed, the contents of bit positions 56-63 of the parameter register are used as a code specifying an architectural mode to which all CPUs in the configuration are to be set, as follows:

- Code 0 specifies the ESA/390 architectural mode or ESA/390-compatibility mode, with z/Architecture- unique information captured or preserved.
- Code 1 specifies the z/Architecture architectural mode.
- Code 2 specifies the z/Architecture architectural mode, with captured z/Architecture-unique information restored.

Bits 0-55 of the parameter register are ignored. The contents of the CPU-address register of the SIGNAL PROCESSOR instruction are ignored; all CPUs in the configuration are considered to be addressed.

The order is accepted only if all of the following conditions are true:

- The code is 0, 1, or 2.
- The CZAM facility is not installed.
- The CPU is not already in the mode specified by the code.
- Each of all other CPUs is in either the stopped or the check-stop state.
- No other condition precludes accepting the order.

If accepted, the order is completed by all CPUs during the execution of SIGNAL PROCESSOR. In no case can different CPUs be in different architectural modes.

The set-architecture order is completed as follows:

- If the code in the parameter register is not 0, 1, or 2, if the CZAM facility is installed, or if the CPU is already in the architectural mode specified by the code, the order is not accepted. Instead, bit 55 (invalid parameter) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- If the code in the parameter register is 0 and multithreading is enabled, return to the ESA/390 architectural mode or ESA/390-compatibility mode is not possible and the order is not accepted. Instead, bit 54 (incorrect state) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- If it is not true that all other CPUs in the configuration are in the stopped or check-stop state, the order is not accepted. Instead, bit 54 (incorrect state) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- The architectural mode of all CPUs in the configuration is set as specified by the code, as follows:
  - If the order changes the architectural mode from ESA/390 to z/Architecture and the code is 1, then, for each CPU in the configuration, the eight-byte current PSW is changed to a 16-byte PSW, and the bits of the 16-byte PSW are set as follows: bits 0-11 and 13-32 are set equal to the same bits of the eight-byte PSW, bit 12 and bits 33-96 are set to zeros, and bits 97-127 are set equal to bits 33-63 of the eight-byte PSW. Also, for each CPU in the configuration, bit 19 of the ESA/390 prefix, which becomes bit 51 of the z/Architecture prefix register, is set to zero.

    If the order changes the architectural mode from ESA/390-compatibility mode to z/Architecture and the code is 1, then for each CPU in the configuration, bit 51 of the prefix register is set to zero.

    If the order changes the architectural mode from ESA/390 or ESA/390-compatibility mode to z/Architecture and the code is 2, the PSW of the CPU executing SIGNAL PROCESSOR and the prefix values of all CPUs are set as in the code-1 case. For each of all other CPUs in the configuration, the PSW is set with the value of the captured-z/Architecture-PSW register — a PSW saved when the CPU last went from the z/Architecture mode to the ESA/390 or ESA/390-compatibility mode because of a set-architecture order with code 0 or a CPU reset due to activation of the load-normal or load-with-dump key. However, the captured-z/Architecture-PSW register has been set to all zeros if the CPU

performed a reset, other than CPU reset, either at the time of the architectural-mode transition or subsequently.

If the configuration is switching from the ESA/390-compatibility mode, for each of all other CPUs in the configuration, bits 0-31 of the control registers are set with the values of the captured-z/Architecture-control registers.

– If the order changes the architectural mode from z/Architecture to ESA/390 or ESA/390-compatibility, then, for each CPU in the configuration, (1) the current PSW, which is the updated PSW in the case of the CPU executing SIGNAL PROCESSOR, is saved in the captured-z/Architecture-PSW register, and (2) the 16-byte current PSW is changed to an eight-byte PSW by setting the bits of the eight-byte PSW as follows: bits 0-11 and 13-32 are set equal to the same bits of the 16-byte PSW, bit 12 is set to one, and bits 33-63 are set equal to bits 97-127 of the 16-byte PSW. Bit 51 of the z/Architecture prefix, which becomes bit 19 of the ESA/390 prefix, remains zero.

  If the order changes the architectural mode from z/Architecture to the ESA/390-compatibility mode, then, for each CPU in the configuration, (a) bits 0-31 of the control registers are saved in the captured-z/Architecture-control registers, and (b) bits 0-31 of all control registers are set to zeros.

- The ALBs and TLBs of all CPUs in the configuration are cleared of their contents.
- A serialization and checkpoint-synchronization function is performed on all CPUs in the configuration.

If the order changes the architectural mode from z/Architecture to ESA/390 or ESA/390-compatibility and the SIGNAL PROCESSOR instruction causes occurrence of an instruction-fetching PER event, only the rightmost 31 bits of the address of the instruction are stored in the ESA/390 PER-address field.

Programming Notes:

1. If the set-architecture order changes the architectural mode from z/Architecture to ESA/390 and bit 31 of the PSW is one, the PSW is invalid.

   If the order changes the architectural mode from z/Architecture to the ESA/390-compatibility mode, it is unpredictable whether the PSW is considered to be invalid if bit 31 is one.

2. When a program is intended to operate in the z/Architecture architectural mode but may be loaded in any of the ESA/390, ESA/390-compatibility, or z/Architecture architectural mode, the program can implement a common code path to issue the SIGNAL PROCESSOR set-architecture command to set the architectural mode to z/Architecture. If the program was loaded in a configuration in which the configuration-z/Architecture-architectural-mode facility is installed, then the SIGP instruction will complete with condition code 1, and the status in general register R_1 will indicate invalid parameter (that is, bit 55 of the register will be one). Assuming that the SIGP parameter value is valid (1 or 2), then the program can safely assume that the invalid parameter condition means that the configuration was already in the z/Architecture architectural mode.

Conditional Emergency Signal

An emergency-signal external-interruption condition may be generated at the addressed CPU. The contents of bit positions 48-63 of the parameter register of the SIGNAL PROCESSOR instruction are treated as a check-ASN value which is used as part of the condition checking of the addressed CPU. If signaling is permitted, the interruption condition becomes pending during the execution of SIGNAL PROCESSOR. The associated interruption occurs when the CPU is enabled for that condition and does not necessarily occur during the execution of SIGNAL PROCESSOR. The address of the CPU sending the signal is provided with the interruption code when the interruption occurs. At any one time the receiving CPU can keep pending one emergency-signal condition for each CPU in the configuration, including the receiving CPU itself.

The conditional-emergency-signal order is recognized only when the CPU is in the z/Architecture architectural mode. Otherwise, an unassigned-order condition is recognized.

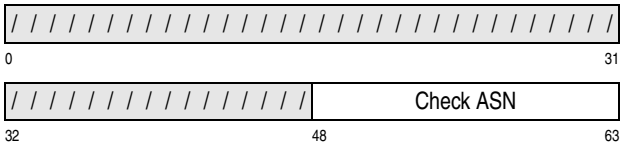
The order is accepted only when the addressed CPU is in the stopped or the operating state and one of the following conditions of the addressed CPU can be determined:

- The PSW is disabled for external interruptions, I/O interruptions, or both.
- The CPU is in the wait state and the instruction address in the PSW is not zero.
- The CPU is not in the wait state, and the specified check-ASN value equals an ASN of the CPU (primary, secondary, both), regardless of the setting of bit positions 16-17 of the PSW.

When making the above determination of possible conditions is precluded, the conditional nature of the order is not effective, and instead the order is accepted as if an emergency-signal order had been specified.

If the order is accepted, the interruption is made pending, and condition code 0 is set. Otherwise, the interruption is not made pending, bit 54 (incorrect state) of the status register is set to one, and condition code 1 is set.

The parameter register has the following format:



Sense Running Status

When the multithreading facility is not enabled, the following applies:

- If the addressed CPU is running, condition code 0 is set.
- If the addressed CPU is not running, status bit 53 is set to one and all other status bits are set to zero in the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction, and condition code 1 is set.

When the multithreading facility is enabled, the following applies:

- If any CPU of the core in which the addressed CPU is a member is running, condition code 0 is set.
- If all CPUs of the core in which the addressed CPU is a member are not running, status bit 53 is set to one and all other status bits are set to zero in the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction, and condition code 1 is set.

No other action is caused at the addressed CPU.

The sense-running-status order is recognized when the CPU is in the z/Architecture architectural mode. It is unpredictable whether the order is recognized when the CPU is in the ESA/390-compatibility mode; if the order is not recognized, an unassigned-order condition is recognized.

Programming Note: When the multithreading facility is enabled, a sense-running-status order addressed to a stopped CPU will complete with condition code 0 if any other CPU in the same core is running.

Set Multithreading

The set-multithreading order enables the multithreading facility.

Bit positions 59-63 of the parameter register contain the program-specified maximum thread identification to be provided in the configuration. The program-specified maximum thread identification is one less than the number of CPUs to be made addressable in each core. For example, a value of 3 in bit positions 59-63 indicates that four threads are to be provided. Bits 0-58 of the parameter register are reserved and should contain zeros; otherwise, the program may not operate compatibly in the future.

The contents of the CPU-address register of the SIGNAL PROCESSOR instruction are ignored; all CPUs in the configuration are considered to be addressed.

If accepted, the order is completed by all CPUs during the execution of SIGNAL PROCESSOR. The set-multithreading order is completed as follows:

- If the multithreading facility is not installed or the CPU is not in the z/Architecture architectural mode, or an external control prevents the use of multithreading, the order is not accepted. Instead, bit 62 (invalid order) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- If it is not true that all other CPUs in the configuration are in the stopped or check-stop state, or if the configuration is already enabled for multithreading, the order is not accepted. Instead, bit 54 (incorrect state) of the general register desig-

nated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.

- If the program-specified maximum thread identification in bit positions 59-63 of the parameter register contains a value not supported by the model, the order is not accepted. Instead, bit 55 (invalid parameter) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- When the program-specified maximum thread identification is nonzero, the following is performed:
  - The configuration is enabled for multithreading, resulting in CPU-address expansion as described in "CPU-Address Expansion" on page 4-83.
  - The meaning of a CPU-type topology-list entry (TLE) of a SYSIB 15.1.x in the STORE SYSTEM INFORMATION instruction is changed (see "**Fields Specific to a CPU-type TLE**" on page 10-157).
  - The ALBs and TLBs of all CPUs in the configuration are cleared of their contents.
  - The MT-diagnostic counter set is set to the active state.

  When the program-specified maximum thread identification is zero, the configuration is not enabled for multithreading.
- A serialization and checkpoint-synchronization function is performed on all CPUs in the configuration.

Upon successful completion, all CPUs other than the CPU executing the set-multithreading order remain in the stopped or check-stop state. However, if a CPU was in the check-stop state before multithreading is enabled, it is unpredictable whether the CPUs having nonzero thread IDs in the same core are placed in the stopped or check-stopped state.

The architected register context (that is, the contents of the PSW, CPU timer, clock comparator, general registers, floating-point registers and floating-point-control register, vector registers, control registers, access registers, prefix register, and TOD-programmable register) of each CPU before multithreading is enabled becomes the architected register context of the CPU having TID zero of each respective core after multithreading is enabled. Similarly, the architected register context of the CPU having TID zero of each core of an MT-enabled configuration becomes the architected register context of each respective CPU when multithreading is disabled as a result of the activation of the load-normal or load-with-dump key.

The architected register context of all CPUs having a nonzero thread identification are retained when the multithreading facility is disabled as a result of the activation of the load-normal or load-with-dump key. If the multithreading facility is subsequently re-enabled without an intervening clear reset, the architected register context of all CPUs having a nonzero thread identification are restored.

When multithreading is re-enabled after having been disabled by the activation of the load-normal or load-with-dump key, if the value of the program-specified maximum thread identification in bits 59-63 of the parameter register differs from that used in the preceding enablement, then the architected register context of all CPUs having nonzero thread IDs is unpredictable.

Store Additional Status at Address

Additional status information is stored in the status area designated by the parameter register.

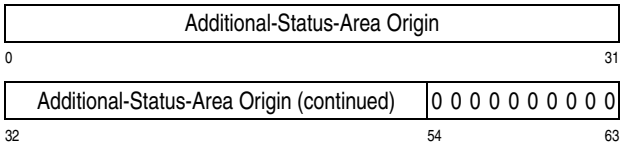
When the guarded-storage facility is not installed, the contents of bit positions 0-53 of the parameter register of the SIGNAL PROCESSOR instruction are used as the origin of a 1024-byte area on a 1024-byte boundary in absolute storage into which the additional status of the addressed CPU is stored. Bits 54-63 of the parameter register must be zero.

When the guarded-storage facility is installed, bit positions 60-63 of the parameter register contain a length characteristic (LC) that specifies the size and alignment of the additional-status area as a power of two; an LC value of zero is treated as 10. Bits 0 through 63–LC of the parameter register, with LC zeros appended on the right, are used as the absolute address of a 2^{LC}-byte area, aligned on a 2^{LC}-byte boundary, into which the additional status of the addressed CPU is stored. Bit positions 64-LC through 59 of the parameter register must contain zeros. The valid length-characteristic values are the same as those in the machine-check-extended-save-

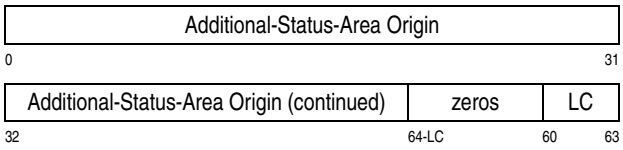
area designation, as shown in Figure 3-18 on page 3-81; all other LC values are reserved.

The order is accepted only if the addressed CPU is in the stopped state, the status-area origin designates a location which is available in the configuration, and no other condition precludes accepting the order. Verification of the stopped state of the addressed CPU and of the availability of the designated storage is performed during execution of SIGNAL PROCESSOR. If accepted, the order is not necessarily completed during the execution of SIGNAL PROCESSOR.

When the guarded-storage facility is not installed, the parameter register has the following format:



When the guarded-storage facility is installed, the parameter register has the following format:



The store-additional-status-at-address order is completed as follows:

- The order is accepted only if at least one of the facilities that produce additional status is installed, including any of the following:
  - Vector facility for z/Architecture
  - Guarded-storage facility
- If neither of these facilities is installed, the order is not accepted. Instead, bit 62 (invalid order) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- If the addressed CPU is not in the stopped state, the order is not accepted. Instead, bit 54 (incorrect state) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- If the guarded-storage facility is installed, and a reserved LC value is specified, or if any of the reserved bit positions in the parameter register do not contain zero, the order is not accepted by the addressed CPU, bit 55 (invalid parameter) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set. When the guarded-storage facility is not installed, bit positions 54-63 of the parameter register are reserved. When the guarded-storage facility is installed, bit positions 64-LC through 59 are reserved.
- The address of the area into which status is to be stored is tested for availability. When the guarded-storage facility is not installed, the absolute address of a 1024-byte area of storage is formed by appending 10 zeros to the right of bits 0-53 of the parameter value. When the guarded-storage facility is installed, the absolute address of a 2^{LC}-byte area of storage is formed by appending LC zeros to the right of bits 0 through 63-LC of the parameter register.

  This address is treated as a 64-bit absolute address regardless of whether the sending and receiving CPUs are in the 24-bit, 31-bit, or 64-bit addressing mode. The block of storage at this address is accessed. The access is not subject to protection, and the associated reference bit may or may not be set to one. If the block is not available in the configuration, the order is not accepted by the addressed CPU, bit 55 (invalid parameter) of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- A serialization and checkpoint-synchronization function is performed on the addressed CPU following storing of the status.

The format of the additional-status area is identical to that of the machine-check extended save area as described in “Machine-Check Extended Save Area (MCESA)” on page 11-24. If the minimum length required to store a facility’s registers is not specified in the length code, then those registers are not stored in the additional-status area.

Programming Note: For a discussion of the relative performance of the SIGNAL PROCESSOR orders, see the programming note following the instruction “SIGNAL PROCESSOR” on page 10-133.

Conditions Determining Response

Conditions Precluding Interpretation of the Order Code

The following situations preclude the initiation of the order. The sequence in which the situations are listed is the order of priority for indicating concurrently existing situations:

1. The access path to the addressed CPU is busy because a concurrently executed SIGNAL PROCESSOR is using the CPU-signaling-and-response facility. The CPU which is concurrently executing the instruction can be any CPU in the configuration other than this CPU, and the CPU address can be any address, including that of this CPU or an invalid address. The order is rejected. Condition code 2 is set.

2. The addressed CPU is not operational; that is, it is not provided in the installation, it is not in the configuration, its use is restricted based on model or CPU type, it is in any of certain customer-engineer test modes, or its power is off. The order is rejected. Condition code 3 is set. This condition cannot arise as a result of a SIGNAL PROCESSOR instruction executed by a CPU addressing itself.

3. One of the following conditions exists at the addressed CPU:

   a. A previously issued start, stop, restart, stop-and-store-status, set-prefix, store-status-at-address order, or store-additional-status-at-address has been accepted by the addressed CPU, and execution of the function requested by the order has not yet been completed.

   b. A manual start, stop, restart, or store-status function has been initiated at the addressed CPU, and the function has not yet been completed. This condition cannot arise as a result of a SIGNAL PROCESSOR instruction executed by a CPU addressing itself.

   If the currently specified order is sense, external call, emergency signal, start, stop, restart, stop and store status, set prefix, store status at address, set architecture, set multithreading, or store additional status at address, then the order is rejected, and condition code 2 is set. If the currently specified order is one of the reset orders, or an unassigned or not-implemented order, the order code is interpreted as described in "Status Bits" on page 4-94.

4. One of the following conditions exists at the addressed CPU:

   a. A previously issued initial-CPU-reset or CPU-reset order has been accepted by the addressed CPU, and execution of the function requested by the order has not yet been completed.

   b. A manual-reset function has been initiated at the addressed CPU, and the function has not yet been completed. This condition cannot arise as a result of a SIGNAL PROCESSOR instruction executed by a CPU addressing itself.

   If the currently specified order is sense, external call, emergency signal, start, stop, restart, stop and store status, set prefix, store status at address, set multithreading, or set architecture, then the order is rejected, and condition code 2 is set. If the currently specified order is one of the reset orders, or an unassigned or not-implemented order, either the order is rejected and condition code 2 is set or the order code is interpreted as described in "Status Bits" on page 4-94.

When any of the conditions described in items 3 and 4 exists, the addressed CPU is referred to as "busy." Busy is not indicated if the addressed CPU is in the check-stop state or when the operator-intervening condition exists. A CPU-busy condition is normally of short duration; however, the conditions described in item 3 may last indefinitely because of a string of interruptions. In this situation, however, the CPU does not appear busy to any of the reset orders.

When the conditions described in items 1 and 2 above do not apply and operator-intervening and receiver-check status conditions do not exist at the addressed CPU, reset orders may be accepted regardless of whether the addressed CPU has completed a previously accepted order. This may cause the previous order to be lost when it is only partially completed, making unpredictable whether the results defined for the lost order are obtained.

Status Bits

Various status conditions are defined whereby the issuing and addressed CPUs can indicate their responses to the specified order. The status condi-

tions and their bit positions in the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction are shown in Figure 4-34.

Bit Position	Status Condition
32	Equipment check
33-52	Unassigned; zeros stored
53	Not running
54	Incorrect state
55	Invalid parameter
56	External-call pending
57	Stopped
58	Operator intervening
59	Check stop
60	Unassigned; zero stored
61	Inoperative
62	Invalid order
63	Receiver check

Figure 4-34. Status Conditions

The status condition assigned to bit position 32, and to bit position 55 when the order is set architecture or set multithreading, are generated by the CPU executing SIGNAL PROCESSOR. The remaining status conditions are generated by the addressed CPU.

When the invalid-parameter condition exists for the set-architecture or set multithreading order, bit 55 of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, all other bits in bit positions 32-63 are set to zeros, bits 0-31 of the register remain unchanged, and condition code 1 is set. No other action is taken.

When the equipment-check condition exists, except when the invalid-parameter condition exists for the set-architecture or set multithreading order, bit 32 of the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction is set to one, unassigned bits in bit positions 32-63 of the status register are set to zeros, the other status bits are unpredictable, and bits 0-31 of the register remain unchanged. In this case, condition code 1 is set independent of whether the access path to the addressed CPU is busy and independent of whether the addressed CPU is not operational, is busy, or has presented zero status.

When the access path to the addressed CPU is not busy and the addressed CPU is operational and does not indicate busy to the currently specified order, the addressed CPU presents its status to the issuing CPU. These status bits are of two types:

1. Status bits 53, 54, and 55 when the order is neither set architecture nor set-multithreading, 56-59, and 61 indicate the presence of the corresponding conditions in the addressed CPU at the time the order code is received. Except in response to the sense order and the sense-running-status order, each condition is indicated only when the condition precludes the successful execution of the specified order, although invalid parameter is not necessarily indicated when any other precluding condition exists. In the case of sense, all existing status conditions except not-running are indicated; the operator-intervening condition is indicated if it precludes the execution of any installed order.

2. Status bits 62 and 63 indicate that the corresponding conditions were detected by the addressed CPU during reception of the order.

If the presented status is all zeros, the addressed CPU has accepted the order, and condition code 0 is set at the issuing CPU; if the presented status is not all zeros, the order has been rejected, the status is stored at the issuing CPU in the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction, zeros are stored in the unassigned positions in bit positions 32-63 of the register, bits 0-31 of the register remain unchanged, and condition code 1 is set.

When the order is set architecture or set multithreading, “the addressed CPU” refers to each of the other CPUs in the configuration. Those CPUs, in an unpredictable order, are tested for a condition that causes setting of condition code 1, 2, or 3. Conditions are prioritized for a single CPU as if it were the only CPU addressed, but there is no prioritization across CPUs. If a condition is recognized, no further CPUs are tested, the condition code corresponding to the condition is set, and the execution of SIGNAL PROCESSOR is completed.

The status conditions are defined as follows:

Equipment Check: This condition exists when the CPU executing the instruction detects equipment malfunctioning that has affected only the execution of this instruction and the associated order. The order code may or may not have been transmitted and may or may not have been accepted, and the status bits

provided by the addressed CPU may be in error. This condition is not detected if the invalid-parameter condition for the set-architecture or set-multithreading order is detected.

Not Running: This condition exists when the addressed CPU is not running. The condition, when present, is indicated only in response to the sense-running-status order. This condition is not reported as a result of a SIGNAL PROCESSOR instruction executed by a CPU addressing itself; condition code 0 is always set in this case. A CPU is running when it is assigned to a physical CPU.

Incorrect State: The condition exists in the following cases:

1. A set-prefix, store-status-at-address, or store-additional-status-at-address order has been rejected because the addressed CPU is not stopped.
2. A set-architecture, or set-multithreadingorder has been rejected because not all other CPUs are stopped or in the check-stop state.
3. A set-multithreading order or a set-architecture order to set the ESA/390 architectural mode or ESA/390-compatibility mode has been rejected because multithreading is currently enabled.
4. A conditional emergency-signal order has been rejected because the addressed CPU is not in a required state.

When applicable, this status is generated during execution of SIGNAL PROCESSOR and is indicated concurrently with other indications of conditions which preclude execution of the order, except that this status is not generated if an invalid-parameter condition exists for a set-architecture order.

Invalid Parameter: This condition exists in the following cases:

1. The parameter value supplied with a set-prefix, store-status-at-address, or store-additional-status-at-address order designates a storage location which is not available in the configuration. When applicable, this status is generated during execution of SIGNAL PROCESSOR, except that it is not necessarily generated when another condition precluding execution of the order also exists.
2. The parameter value supplied with a set-architecture order either is not 0 or 1 or specifies the current architectural mode. When applicable, this status is generated during execution of SIGNAL PROCESSOR, and no other status is generated.
3. The parameter value supplied with a set-multithreading order specifies a value not supported by the model.
4. The parameter value supplied with a store-additional-status-at-address order contains a nonzero value in bit positions 54-63 of the parameter register.
5. The configuration-z/Architecture-architectural-mode (CZAM) facility is installed, and a set-architecture order was specified.

External Call Pending: This condition exists when an external-call interruption condition is pending in the addressed CPU because of a previously issued SIGNAL PROCESSOR order. The condition exists from the time an external-call order is accepted until the resultant external interruption has been completed or a CPU reset occurs. The condition may be due to the issuing CPU or another CPU. The condition, when present, is indicated only in response to sense and to external call.

Stopped: This condition exists when the addressed CPU is in the stopped state. The condition, when present, is indicated only in response to sense. This condition cannot be reported as a result of a SIGNAL PROCESSOR instruction executed by a CPU addressing itself.

Operator Intervening: This condition exists when the addressed CPU is executing certain operations initiated from local or remote operator facilities. The particular manually initiated operations that cause this condition to be present depend on the model and on the order specified. The operator-intervening condition may exist when the addressed CPU uses reloadable control storage to perform an order and the required licensed internal code has not been loaded by the IML function. The operator-intervening condition, when present, can be indicated in response to all orders. Operator intervening is indicated in response to sense if the condition is present and precludes the acceptance of any of the installed orders. The condition may also be indicated in response to unassigned or uninstalled orders. This condition cannot arise as a result of a SIGNAL PRO-

CESSOR instruction executed by a CPU addressing itself.

Check Stop: This condition exists when the addressed CPU is in the check-stop state. The condition, when present, is indicated only in response to sense, external call, emergency signal, conditional emergency signal, start, stop, restart, set prefix, store status at address, and stop and store status. The condition may also be indicated in response to unassigned or uninstalled orders. This condition cannot be reported as a result of a SIGNAL PROCESSOR instruction executed by a CPU addressing itself.

Inoperative: This condition indicates that the execution of the operation specified by the order code requires the use of a service processor which is inoperative. The failure of the service processor may have been previously reported by a service-processor-damage machine-check condition. The inoperative condition cannot occur for the conditional emergency signal, emergency-signal, external-call, sense, or sense-running-status order codes.

Invalid Order: This condition exists during the communications associated with the execution of SIGNAL PROCESSOR when an unassigned or uninstalled order code is decoded.

Receiver Check: This condition exists when the addressed CPU detects malfunctioning of equipment during the communications associated with the execution of SIGNAL PROCESSOR. When this condition is indicated, the order has not been initiated, and, since the malfunction may have affected the generation of the remaining receiver status bits, these bits are not necessarily valid. A machine-check condition may or may not have been generated at the addressed CPU.

The following chart summarizes which status conditions are presented to the issuing CPU in response to each order code.

	Status Condition									
Order	53 – Not running	54 – Incorrect state	55 – Invalid parameter	56 – External call pending	57 – Stopped	58 – Operator intervening#	59 – Check stop	61 – Inoperative	62 – Invalid order	63 – Receiver check≠
Sense	0	0	0	X	X	X	X	0	0	X
External call	0	0	0	X	0	X	X	0	0	X
Emergency signal	0	0	0	0	0	X	X	0	0	X
Start	0	0	0	0	0	X	X	X	0	X
Stop	0	0	0	0	0	X	X	X	0	X
Restart	0	0	0	0	0	X	X	X	0	X
Stop and store status	0	0	0	0	0	X	X	X	0	X
Initial CPU reset	0	0	0	0	0	X	0	X	0	X
CPU reset	0	0	0	0	0	X	0	X	0	X
Set prefix	0	X	X	0	0	X	X	X	0	X
Store status at address	0	X	X	0	0	X	X	X	0	X
Set architecture	0	X	X	0	0	X	0	X	X	X
Conditional emergency signal	0	X	0	0	0	X	X	0	X	X
Sense running status	X	0	0	0	0	0	0	0	X	0
Set multithreading	0	X	X	0	0	X	0	X	X	X
Store additional status at address	0	X	X	0	0	X	X	X	X	X
Unassigned order	0	0	0	0	0	X	E	X	1	0

Explanation

#	The current state of the operator-intervening condition may depend on the order code that is being interpreted.
≠	If a one is presented in the receiver-check bit position, the values presented in the other bit positions are not necessarily valid.
0	A zero is presented in this bit position regardless of the current state of this condition.
1	A one is presented in this bit position.
X	A zero or a one is presented in this bit position, reflecting the current state of the corresponding condition.
E	Either a zero or the current state of the corresponding condition is indicated.

Figure 4-35. Status Conditions in Response to Order Code

If the presented status bits are all zeros, the order has been accepted, and the issuing CPU sets condition code 0. If one or more ones are presented, the order has been rejected, and the issuing CPU stores the status in the general register designated by the R_1 field of the SIGNAL PROCESSOR instruction and sets condition code 1.

Programming Notes:

1. All SIGNAL PROCESSOR orders except set architecture (which in effect is addressed to all other CPUs and affects all CPUs) can be addressed to this same CPU. The following are examples of functions obtained by a CPU addressing SIGNAL PROCESSOR to itself:

   a. *Sense* indicates whether an external-call condition is pending.

   b. *External call* and *emergency signal* cause the corresponding interruption conditions to be generated. *External call* can be rejected because of a previously generated external-call condition.

   c. *Start* sets condition code 0 and has no other effect.

   d. *Stop* causes the CPU to set condition code 0, take pending interruptions for which it is enabled, and enter the stopped state.

   e. *Restart* provides a means to store the current PSW.

   f. *Stop and store status* causes the machine to stop and store all current status.

2. Two CPUs can simultaneously execute SIGNAL PROCESSOR, with each CPU addressing the other. When this occurs, one CPU, but not both, can find the access path busy because of the transmission of the order code or status bits associated with SIGNAL PROCESSOR that is being executed by the other CPU. Alternatively, both CPUs can find the access path available and transmit the order codes to each other. In particular, two CPUs can simultaneously stop, restart, or reset each other.

3. To obtain status from another CPU which is in the check-stop state by means of the store-status-at-address or store-additional-status-at-address order, a CPU reset operation should first be used to bring the CPU to the stopped state. This reset order does not alter the status, and, depending on the nature of the malfunction, provides the best chance of establishing conditions in the addressed CPU which allow status to be obtained.

Facility Indications

Facilities installed in a configuration are indicated by facility bits stored by the STORE FACILITY LIST (STFL) and STORE FACILITY LIST EXTENDED (STFLE) instructions.

STORE FACILITY LIST stores an indication of up to 32 facilities in the word at real location 200. STFL is described in Chapter 10, “Control Instructions”.

STORE FACILITY LIST EXTENDED stores a variable number of doublewords containing facility bits in a program-specified location. STFLE is described in Chapter 7, “General Instructions”.

Figure 4-36 shows the meanings of the assigned facility bits.

Bit	Meaning When Bit Is One
0	The instructions marked “N3” in the instruction-summary figures in Chapters 7 and 10 are installed.
1	The z/Architecture architectural mode is installed.
2	The z/Architecture architectural mode is active. When bits 2 and 168 are both zero, the ESA/390 architectural mode is active. When bit 2 is zero and bit 168 is one, the ESA/390-compatibility mode is active.
3	The DAT-enhancement facility is installed in the z/Architecture architectural mode. The DAT-enhancement facility includes the INVALIDATE DAT TABLE ENTRY (IDTE) and COMPARE AND SWAP AND PURGE (CSPG) instructions.
4	INVALIDATE DAT TABLE ENTRY (IDTE) performs the invalidation-and-clearing operation by selectively clearing TLB segment-table entries when a segment-table entry or entries are invalidated. IDTE also performs the clearing-by-ASCE operation. Unless bit 4 is one, IDTE simply purges all TLBs. Bit 3 is one if bit 4 is one.
5	INVALIDATE DAT TABLE ENTRY (IDTE) performs the invalidation-and-clearing operation by selectively clearing TLB region-table entries when a region-table entry or entries are invalidated. Bits 3 and 4 are ones if bit 5 is one.

Figure 4-36. Assigned Facility Bits (Part 1 of 6)

Bit	Meaning When Bit Is One
6	The ASN-and-LX reuse facility is installed in the z/Architecture architectural mode.
7	The store-facility-list-extended facility is installed.
8	The enhanced-DAT facility 1 is installed in the z/Architecture architectural mode.
9	The sense-running-status facility is installed in the z/Architecture architectural mode.
10	The conditional-SSKE facility is installed in the z/Architecture architectural mode.
11	The configuration-topology facility is installed in the z/Architecture architectural mode.
12	Assigned to IBM internal use.
13	The IPTE-range facility is installed in the z/Architecture architectural mode.
14	The nonquiescing key-setting facility is installed in the z/Architecture architectural mode.
15	Assigned to IBM internal use.
16	The extended-translation facility 2 is installed.
17	The message-security assist is installed.
18	The long-displacement facility is installed in the z/Architecture architectural mode.
19	The long-displacement facility has high performance. Bit 18 is one if bit 19 is one.
20	The HFP-multiply-and-add/subtract facility is installed.
21	The extended-immediate facility is installed in the z/Architecture architectural mode.
22	The extended-translation facility 3 is installed in the z/Architecture architectural mode.
23	The HFP-unnormalized-extension facility is installed in the z/Architecture architectural mode.
24	The ETF2-enhancement facility is installed.
25	The store-clock-fast facility is installed in the z/Architecture architectural mode.
26	The parsing-enhancement facility is installed in the z/Architecture architectural mode.

Figure 4-36. Assigned Facility Bits (Part 2 of 6)

Bit	Meaning When Bit Is One
27	The move-with-optional-specifications facility is installed in the z/Architecture architectural mode.
28	The TOD-clock-steering facility is installed in the z/Architecture architectural mode.
30	The ETF3-enhancement facility is installed in the z/Architecture architectural mode.
31	The extract-CPU-time facility is installed in the z/Architecture architectural mode.
32	The compare-and-swap-and-store facility is installed in the z/Architecture architectural mode.
33	The compare-and-swap-and-store facility 2 is installed in the z/Architecture architectural mode.
34	The general-instructions-extension facility is installed in the z/Architecture architectural mode.
35	The execute-extensions facility is installed in the z/Architecture architectural mode.
36	The enhanced-monitor facility is installed in the z/Architecture architectural mode.
37	The floating-point extension facility is installed in the z/Architecture architectural mode. When bit 37 is one, bit 42 is also one.
38	The order-preserving-compression facility is installed in the z/Architecture architectural mode.
39	Assigned to IBM internal use.
40	The load-program-parameters facility is installed in the z/Architecture architectural mode.
41	The floating-point-support-enhancement facilities (DFP-rounding, FPR-GR-transfer, FPS-sign-handling, and IEEE-exception-simulation) are installed in the z/Architecture architectural mode.
42	The DFP (decimal-floating-point) facility is installed in the z/Architecture architectural mode.
43	The DFP (decimal-floating-point) facility has high performance. Bit 42 is one if bit 43 is one.
44	The PFPO instruction is installed in the z/Architecture architectural mode.

Figure 4-36. Assigned Facility Bits (Part 3 of 6)

Bit	Meaning When Bit Is One
45	The distinct-operands, fast-BCR-serialization, high-word, and population-count facilities, the interlocked-access facility 1, and the load/store-on-condition facility 1 are installed in the z/Architecture architectural mode.
46	Assigned to IBM internal use.
47	The CMPSC-enhancement facility is installed in the z/Architecture architectural mode.
48	The decimal-floating-point zoned-conversion facility is installed in the z/Architecture architectural mode.
49	The execution-hint, load-and-trap, and processor-assist facilities, and the miscellaneous-instruction-extensions facility 1 are installed in the z/Architecture architectural mode.
50	The constrained transactional-execution facility is installed in the z/Architecture architectural mode. This bit is meaningful only when bit 73 is one.
51	The local-TLB-clearing facility is installed in the z/Architecture architectural mode.
52	The interlocked-access facility 2 is installed.
53	The load/store-on-condition facility 2 and load-and-zero-rightmost-byte facility are installed in the z/Architecture architectural mode.
54	The entropy-encoding compression facility is installed in the z/Architecture architectural mode.
55	Assigned to IBM internal use.
57	The message-security-assist extension 5 is installed in the z/Architecture architectural mode.
58	The miscellaneous-instruction-extensions facility 2 is installed in the z/Architecture architectural mode.
59	Assigned to IBM internal use.
60	Assigned to IBM internal use.
62	Assigned to IBM internal use.
63	Assigned to IBM internal use.
64	Assigned to IBM internal use.
65	Assigned to IBM internal use.

Figure 4-36. Assigned Facility Bits (Part 4 of 6)

Bit	Meaning When Bit Is One
66	The reset-reference-bits-multiple facility is installed in the z/Architecture architectural mode.
67	The CPU-measurement counter facility is installed in the z/Architecture architectural mode.
68	The CPU-measurement sampling facility is installed in the z/Architecture architectural mode.
69	Assigned to IBM internal use.
70	Assigned to IBM internal use.
71	Assigned to IBM internal use.
72	Assigned to IBM internal use.
73	The transactional-execution facility is installed in the z/Architecture architectural mode. Bit 49 is one when bit 73 is one.
74	The store-hypervisor-information facility is installed in the z/Architecture architectural mode (see Reference [11.] on page xxx).
75	The access-exception-fetch/store-indication facility is installed in the z/Architecture architectural mode.
76	The message-security-assist extension 3 is installed in the z/Architecture architectural mode.
77	The message-security-assist extension 4 is installed in the z/Architecture architectural mode.
78	The enhanced-DAT facility 2 is installed in the z/Architecture architectural mode.
80	The decimal-floating-point packed-conversion facility is installed in the z/Architecture architectural mode.
128	Assigned to IBM internal use.
129	The vector facility for z/Architecture is installed in the z/Architecture architectural mode.
130	The instruction-execution-protection facility is installed in the z/Architecture architectural mode.
131	The side-effect-access facility is installed in the z/Architecture architectural mode.
133	The guarded-storage facility is installed in the z/Architecture architectural mode.

Figure 4-36. Assigned Facility Bits (Part 5 of 6)

Bit	Meaning When Bit Is One
134	The vector packed decimal facility is installed in the z/Architecture architectural mode. When bit 134 is one, bit 129 is also one.
135	The vector enhancements facility 1 is installed in the z/Architecture architectural mode. When bit 135 is one, bit 129 is also one.
138	The configuration-z/Architecture-architectural-mode facility is installed.
139	The multiple-epoch facility is installed in the z/Architecture architectural mode. Bits 25 and 28 are one when bit 139 is one.
140	Assigned to IBM internal use.
141	Assigned to IBM internal use.
142	The store-CPU-counter-multiple facility is installed (see Reference [10.] on page xxx).
144	The test-pending-external-interruption facility is installed in the z/Architecture architectural mode.
145	The insert-reference-bits-multiple facility is installed in the z/Architecture architectural mode.
146	The message-security-assist-extension 8 is installed in the z/Architecture architectural mode. Bit 76 is one when bit 146 is one.
147	Reserved for IBM use.
168	The ESA/390-compatibility-mode facility is installed in the configuration.

Figure 4-36. Assigned Facility Bits (Part 6 of 6)

A bit is set to one regardless of the current architectural mode if its meaning is true. A meaning applies to the current architectural mode unless it is said to apply to a specific architectural mode.

Unassigned bits are reserved for indication of new facilities; these bits may be stored as ones in the future.

STORE FACILITY LIST and STORE FACILITY LIST EXTENDED may report the absence of a facility, even though the execution of the facility's instructions may appear to indicate that the facility is installed. The under-reporting of facility indications may occur because one or more systems in the computing environment (to which the application may be relocated) do not have the facility installed.

Programming Note: Prior to the introduction of z/Architecture, determination of the presence of a facility was often accomplished by means of a trial execution of an instruction. If an operation exception was not recognized during the trial execution, then it could be assumed that the facility was present.

Similarly, certain instructions provide a query function to determine the availability of other functions of the instruction or related instructions. However, a program might attempt to determine the availability of a particular function by trial execution of the function.

With the advent of facility and function indications in z/Architecture, the technique of trial execution should be avoided — particularly if a workload may be relocated to another system in which a facility's instructions may not be present. In such an environment, STORE FACILITY LIST EXTENDED (STFLE) or an instruction's query or test function may provide a more accurate indication of facilities and functions that are available on all systems in the computing environment.

Chapter 5. Program Execution

Instructions 5-3
Operands 5-3
Instruction Formats 5-3
Register Operands 5-7
Immediate Operands 5-8
Storage Operands 5-9
Address Generation 5-10
Trimodal Addressing 5-10
Sequential Instruction-Address Generation . . 5-10
Operand-Address Generation 5-11
Formation of the Intermediate Value 5-11
Formation of the Operand Address 5-12
Branch-Address Generation 5-12
Formation of the Intermediate Value 5-12
Formation of the Branch Address 5-13
Instruction Execution and Sequencing 5-14
Decision Making 5-14
Loop Control 5-14
Subroutine Linkage without the Linkage Stack 5-14
Simple Branch Instructions 5-14
Other Linkage Instructions 5-17
Interruptions 5-22
Types of Instruction Ending 5-22
Completion 5-22
Suppression 5-22
Nullification 5-23
Termination 5-23
Interruptible Instructions 5-24
Point of Interruption 5-24
Unit of Operation 5-24
Execution of Interruptible Instructions 5-24
Condition-Code Alternative to Interruptibility 5-25
Exceptions to Nullification and Suppression . 5-26
Modification of DAT-Table Entries 5-27
Trial Execution for Editing Instructions and Translate Instruction 5-27
Authorization Mechanisms 5-27
Mode Requirements 5-28
Extraction-Authority Control 5-28
PSW-Key Mask 5-28
Secondary-Space Control 5-29
Subsystem-Linkage Control 5-29
ASN-Translation Control 5-29
Authorization Index 5-29
Instructions and Controls Related to ASN-and-LX Reuse 5-30
PC-Number Translation 5-33
PC-Number Translation Control 5-34
Control Register 0 5-34
Control Register 5 5-35
PC-Number Translation Tables 5-36
Linkage-Table Entries 5-36
Linkage-First-Table Entries 5-36
Linkage-Second-Table Entries 5-36
Entry-Table Entries 5-37
Table Summary 5-38
PC-Number-Translation Process 5-38
Obtaining the Linkage-Table or Linkage-First-Table Designation 5-42
Linkage-Table Lookup 5-42
Linkage-First-Table Lookup 5-42
Linkage-Second-Table Lookup 5-43
Linkage-Second-Table-Entry-Sequence-Number Comparison 5-43
Entry-Table Lookup 5-43
Recognition of Exceptions during PC-Number Translation 5-44
Home Address Space 5-44
Access-Register Introduction 5-45
Summary 5-45
Access-Register Functions 5-46
Access-Register-Specified Address Spaces 5-46
Access-Register Instructions 5-52
Access-Register Translation 5-53
Access-Register-Translation Control 5-53
Control Register 2 5-53
Control Register 5 5-54
Control Register 8 5-54
Access Registers 5-54
Access-Register-Translation Tables 5-55
Dispatchable-Unit Control Table and Access-List Designations 5-55
Access-List Entries 5-56
ASN-Second-Table Entries 5-57
Access-Register-Translation Process 5-59
Selecting the Access-List-Entry Token 5-62
Obtaining the Primary or Secondary Address-Space-Control Element 5-62
Checking the First Byte of the ALET 5-62
Obtaining the Effective Access-List Designation 5-62
Access-List Lookup 5-62
Locating the ASN-Second-Table Entry 5-63
Authorizing the Use of the Access-List Entry 5-63
Checking for Access-List-Controlled Protection 5-64

© Copyright IBM Corp. 2000, 2017

Obtaining the Address-Space-Control Element from the ASN-Second-Table Entry 5-64
Recognition of Exceptions during Access-Register Translation 5-64
ART-Lookaside Buffer 5-64
ALB Structure 5-64
Formation of ALB Entries 5-64
Use of ALB Entries 5-65
Modification of ART Tables 5-66
Subspace Groups 5-66
Subspace-Group Tables 5-66
Subspace-Group Dispatchable-Unit Control Table 5-66
Subspace-Group ASN-Second-Table Entries 5-68
Subspace-Replacement Operations 5-70
Linkage-Stack Introduction 5-70
Summary 5-71
Linkage-Stack Functions 5-71
Transferring Program Control. 5-71
Branching Using the Linkage Stack 5-73
Adding and Retrieving Information 5-74
Testing Authorization 5-74
Program-Problem Analysis. 5-75
Linkage-Stack Entry-Table Entries 5-75
Linkage-Stack Operations 5-76
Linkage-Stack-Operations Control 5-78
Control Register 0 5-78
Control Register 15 5-78
Linkage Stack. 5-79
Entry Descriptors 5-79
Header Entries 5-80
Trailer Entries 5-81
State Entries. 5-81
Stacking Process 5-84
Locating Space for a New Entry. 5-84
Forming the New Entry. 5-85
Updating the Current Entry. 5-86
Updating Control Register 15 5-86
Recognition of Exceptions during the Stacking Process 5-86
Unstacking Process 5-86
Locating the Current Entry and Processing a Header Entry 5-87
Checking for a State Entry 5-88
Restoring Information 5-88
Updating the Preceding Entry 5-89
Updating Control Register 15 5-89
Recognition of Exceptions during the Unstacking Process 5-89
Transactional-Execution Facility 5-89
Transactional-Execution Terminology 5-89
Transactional-Execution Facility Controls 5-92
Control Register Bits 5-92
Transaction-Diagnostic-Block Address (TDBA). 5-93
Transaction-Abort PSW (TAPSW). 5-93
Transaction Nesting Depth (TND) 5-93
Transaction Diagnostic Block (TDB) 5-93
Transactional-Execution Facility Instructions .5-97
Restricted Instructions 5-97
Transactional-Execution Facility Operation . .5-99
Transaction Initiation 5-99
Execution in the Transactional-Execution Mode 5-99
Normal Transaction Ending 5-99
Transaction Abort Conditions 5-99
Transaction Abort Processing 5-102
Program-Interruption Filtering on a Transaction Abort 5-103
Priority of Abort Conditions 5-107
Constrained Transaction 5-107
Monitor-Event Counting. 5-109
ESA/390-Compatibility-Mode Facility 5-111
Sequence of Storage References 5-113
Conceptual Sequence 5-113
Overlapped Operation of Instruction Execution. 5-114
Divisible Instruction Execution. 5-115
Interlocks for Virtual-Storage References . . .5-115
Interlocks between Instructions 5-116
Interlocks within a Single Instruction 5-116
Instruction Fetching 5-118
ART-Table and DAT-Table Fetches 5-119
Storage-Key Accesses 5-120
Storage-Operand References 5-123
Storage-Operand Fetch References 5-123
Storage-Operand Store References 5-123
Storage-Operand Update References. . . .5-124
Storage-Operand Consistency 5-125
Single-Access References 5-125
Multiple-Access References 5-125
Block-Concurrent References 5-126
Relation between Operand Accesses 5-129
Storage Operand References in the Transactional-Execution Mode 5-129
Other Storage References 5-129
Relation between Storage-Key Accesses . . .5-130
Serialization. 5-130
CPU Serialization 5-130
Specific-Operand Serialization 5-132
Channel-Program Serialization 5-133
Quiescing 5-133

Normally, operation of the CPU is controlled by instructions in storage that are executed sequentially, one at a time, left to right in an ascending sequence of storage addresses. A change in the sequential operation may be caused by a breaking event, interruptions, SIGNAL PROCESSOR orders, or manual intervention.

Instructions

Each instruction consists of two major parts:

- An operation code (op code), which specifies the operation to be performed
- The designation of the operands that participate.

Operands

Operands can be grouped in three classes: operands located in registers, immediate operands, and operands in storage. Operands may be either explicitly or implicitly designated.

Register operands can be located in general, floating-point, vector, access, or control registers, with the type of register identified by the op code. The register containing the operand is specified by identifying the register in a four-bit field, called the R field, in the instruction. For some instructions, an operand is located in an implicitly designated register, the register being implied by the op code. For vector registers, the register containing the operand is specified using a four-bit field with the addition of a register extension bit as the most significant bit.

Immediate operands are contained within the instruction, and the 8-bit, 16-bit, or 32-bit field containing the immediate operand is called the I field.

Operands in storage may have an implied length; be specified by a bitmask; be specified by a four-bit or eight-bit length specification, called the L field, in the instruction; or have a length specified by the contents of a general register. The addresses of operands in storage are specified by means of a format that uses the contents of a general register as part of the address. This makes it possible to:

1. Specify a complete address by using an abbreviated notation
2. Perform address manipulation using instructions which employ general registers for operands
3. Modify addresses by program means without alteration of the instruction stream
4. Operate independent of the location of data areas by directly using addresses received from other programs

The address used to refer to storage either is contained in a register designated by the R field in the instruction or is calculated from a base address, index, and displacement, specified by the B, X, and D fields, respectively, in the instruction.

When the CPU is in the access-register mode, a B or R field may designate an access register in addition to being used to specify an address.

To describe the execution of instructions, operands are designated as first and second operands and, in some cases, third, fourth, fifth, and sixth operands.

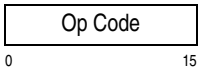
In general, two operands participate in an instruction execution, and the result replaces the first operand. However, CONVERT TO DECIMAL, TEST BLOCK, VECTOR UNPACK ZONED, and instructions with “store” in the instruction name (other than STORE THEN AND SYSTEM MASK and STORE THEN OR SYSTEM MASK) use the second-operand address to designate a location in which to store. TEST AND SET, COMPARE AND SWAP, COMPARE AND SWAP AND STORE, and COMPARE DOUBLE AND SWAP may perform an update on the second operand. Except when otherwise stated, the contents of all registers and storage locations participating in the addressing or execution part of an operation remain unchanged.

Instruction Formats

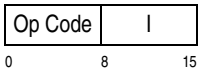
An instruction is one, two, or three halfwords in length and must be located in storage on a halfword boundary. Each instruction is in one of the following basic formats: E, I, IE, MII, RI, RIE, RIL, RIS, RR, RRD, RRE, RRF, RRS, RS, RSI, RSL, RSY, RX, RXE, RXF, RXY, S, SI, SIL, SIY, SMI, SS, SSE, SSF, VRI, VRR, VRS, VRV, VRX, and VSI. Figure 5-1 illustrates the various instruction formats and variations. Where a letter appears to the left of a format variant, it represents one of the variations in assembler syn-

tax as illustrated in the description of the instructions in chapters 7-10, 14, and 18-25.

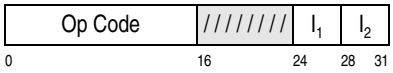
E Format



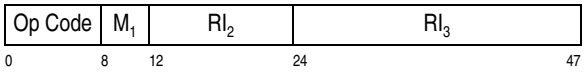
I Format



IE Format

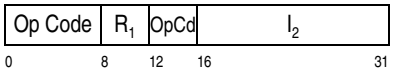


MII Format

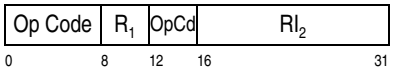


RI Formats

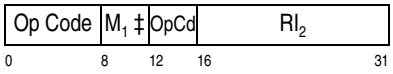
a.



b.

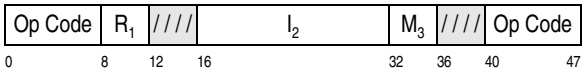


c.

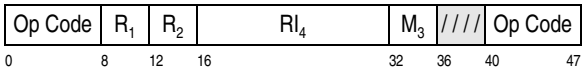


RIE Formats

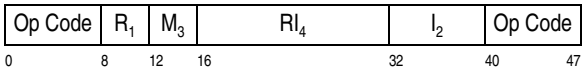
a.



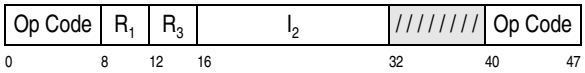
b.



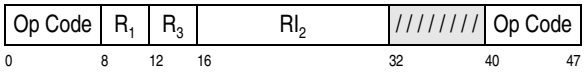
c.



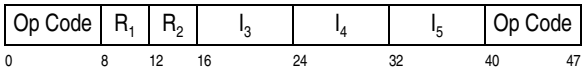
d.



e.



f.



g

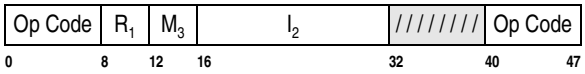
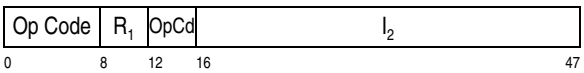


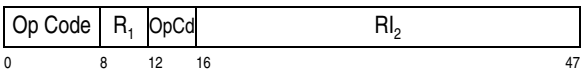
Figure 5-1. Basic Instruction Formats (Part 1 of 5)

RIL Formats

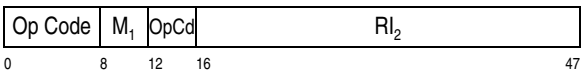
a.



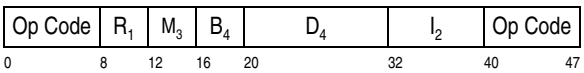
b.



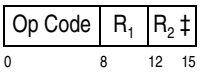
c.



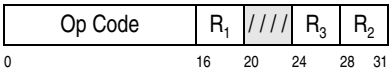
RIS Format



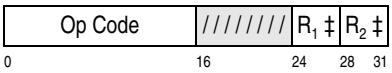
RR Format



RRD Format

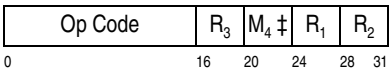


RRE Format

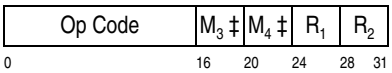


RRF Formats

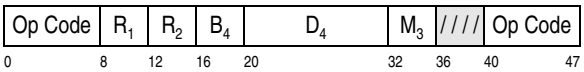
a,b



c-e

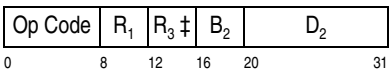


RRS Format

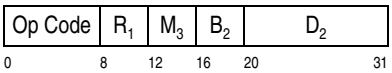


RS Formats

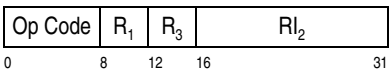
a.



b.

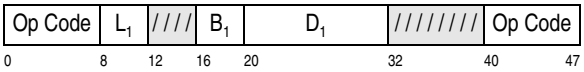


RSI Format



RSL Format

a.



b.

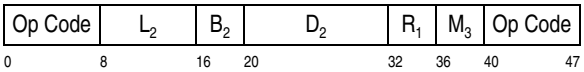
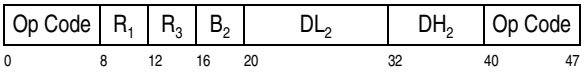


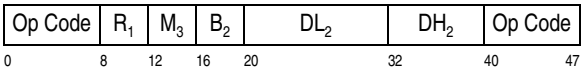
Figure 5-1. Basic Instruction Formats (Part 2 of 5)

RSY Formats

a.

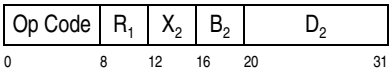


b.

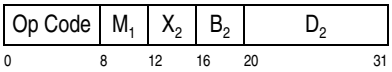


RX Formats

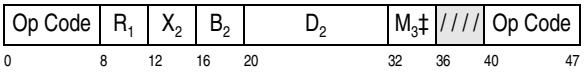
a.



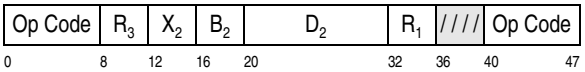
b.



RXE Format

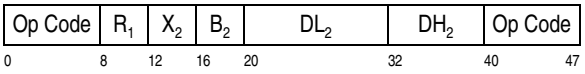


RXF Format

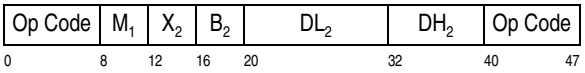


RXY Formats

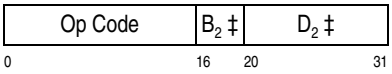
a.



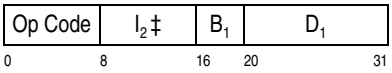
b.



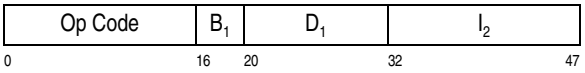
S Format



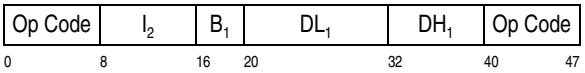
SI Format



SIL Format



SIY Format



SMI Format

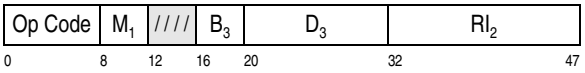
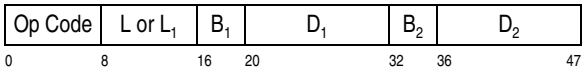


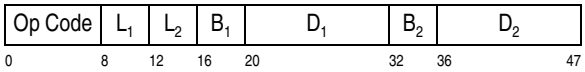
Figure 5-1. Basic Instruction Formats (Part 3 of 5)

SS Formats

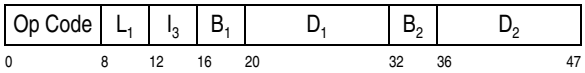
a.



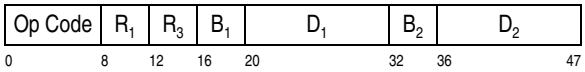
b.



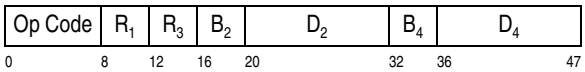
c.



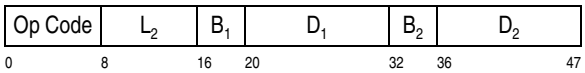
d.



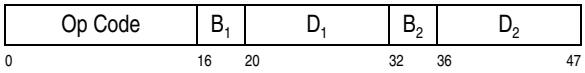
e.



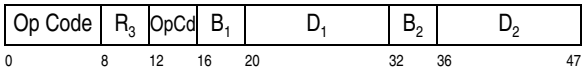
f.



SSE Format

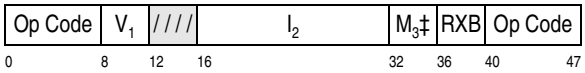


SSF Format

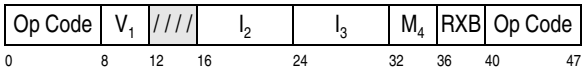


VRI Formats

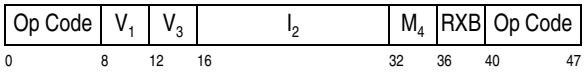
a.



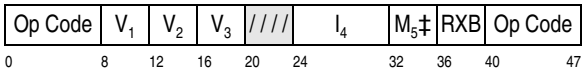
b.



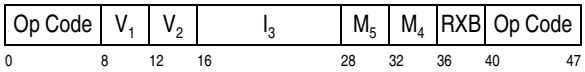
c.



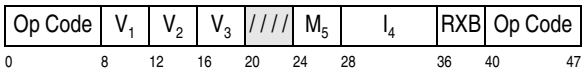
d.



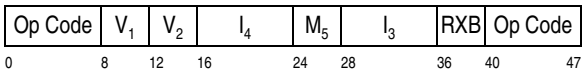
e.



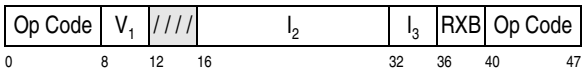
f.



g.



h.



i.

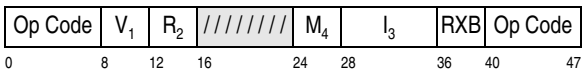
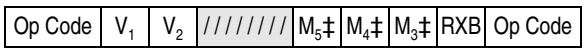
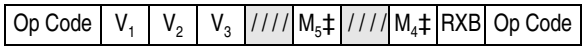
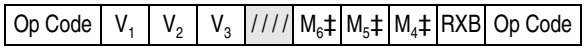


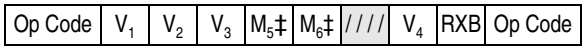
Figure 5-1. Basic Instruction Formats (Part 4 of 5)

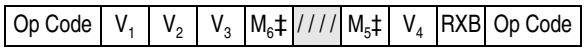
VRR Formats

a.


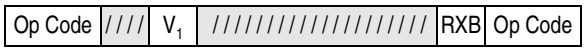
b.


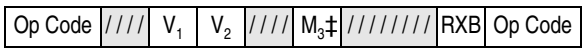
c.


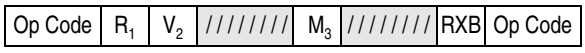
d.


e.


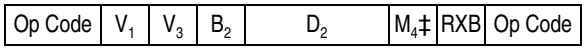
f.

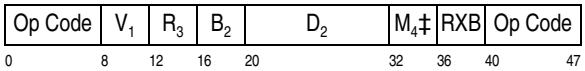
g.


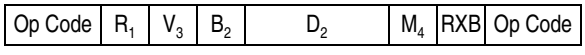
h.


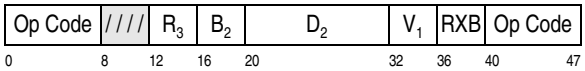
i.


VRS Format

a.


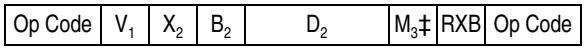
b.


c.


d.


VRV Format

VRX Format



VSI Format

Explanation:

‡ For certain instructions, this operand is not defined.

Figure 5-1. Basic Instruction Formats (Part 5 of 5)

Instruction fields shown in Figure 5-1 on page 5-4 as containing slashes (/) are currently unassigned. These fields in an instruction should contain zeros; otherwise, the program may not operate compatibly in the future.

Some instructions contain fields that vary slightly from the basic format, and in some instructions the operation performed does not follow the general rules stated in this section. All of these exceptions are explicitly identified in the individual instruction descriptions.

The format names indicate, in general terms, the classes of operands which participate in the operation and some details about fields:

- E denotes an operation using implied operands and an extended op-code field.
- I denotes an immediate operation.
- IE denotes an immediate-and-immediate operation.
- MII denotes a masked immediate-and-immediate operation.
- RI denotes a register-and-immediate operation and an extended op-code field.
- RIE denotes a register-and-immediate operation and a longer extended op-code field.
- RIL denotes a register-and-immediate operation, an extended op-code field, and a longer immediate field.
- RIS denotes a register-and-immediate operation and a storage operation.
- RR denotes a register-and-register operation.
- RRD denotes a register-and-register operation, an extended op-code field, and an additional R field.
- RRE denotes a register-and-register operation and an extended op-code field.
- RRF denotes a register-and-register operation, an extended op-code field, and an additional R field, M field, or both.
- RRS denotes a register-and-register operation and a storage operation.
- RS denotes a register-and-storage operation.
- RSI denotes a register-and-immediate operation.
- RSL denotes a storage operation (with an instruction format derived from the ESA/390 RSE format).
- RSY denotes a register-and-storage operation, an extended op-code field, and a long displacement field.
- RX denotes a register-and-indexed-storage operation.

- RXE denotes a register-and-indexed-storage operation and an extended op-code field.
- RXF denotes a register-and-indexed-storage operation, an extended op-code field, and an additional R field.
- RXY denotes a register-and-indexed-storage operation, an extended op-code field, and a long displacement field.
- S denotes an operation using an implied operand and storage.
- SI denotes a storage-and-immediate operation.
- SIL denotes a storage-and-immediate operation, with a 16-bit immediate field.
- SIY denotes a storage-and-immediate operation and a long displacement field.
- SMI denotes a masked storage-and-immediate operation.
- SS denotes a storage-and-storage operation.
- SSE denotes a storage-and-storage operation and an extended op-code field.
- SSF denotes a storage-and-storage operation and an extended op-code field.
- VRI denotes a vector register-and-immediate operation and an extended op-code field.
- VRR denotes a vector register-and-register operation and an extended op-code field.
- VRS denotes a vector register-and-storage operation and an extended op-code field.
- VRV denotes a vector register-and-vector-index-storage operation and an extended op-code field.
- VRX denotes a vector register-and-index-storage operation and an extended op-code field.
- VSI denotes a vector register-and-storage operation and an extended op-code field.

In the I, RR, RS, RSI, RX, SI, and SS formats, the first byte of an instruction contains the op code. In the E, RRE, RRF, S, SIL, and SSE formats, the first two bytes of an instruction contain the op code, except that for some instructions in the S format, the op code is in only the first byte. In the RI, RIL, and SSF formats, the op code is in the first byte and bit positions 12-15 of an instruction. In the RIE, RIS, RRS, RSL, RSY, RXE, RXF, RXY, SIY, VRI, VRR, VRS, VRX, and VSI formats, the op code is in the first byte and the sixth byte of an instruction.

The first two bits of the first or only byte of the op code specify the length and format of the instruction, as follows:

Bit Positions 0-1	Instruction Length (in Halfwords)	Instruction Format
00	One	I / E / RR
01	Two	RX
10	Two	RI / RRD / RRE / RRF / RS / RSI / RX / S / SI
11	Three	MII / RIE / RIL / RIS / RRS / RSL / RSY / RXE / RXF / RXY / SIL / SIY / SMI / SS / SSE / SSF / VRI / VRR / VRS / VRV / VRX / VSI

In the format illustration for each individual instruction description, the op-code field or fields show the op code as hexadecimal digits within single quotes. The hexadecimal representation uses 0-9 for the binary codes 0000-1001 and A-F for the binary codes 1010-1111.

The remaining fields in the format illustration for each instruction are designated by code names, consisting of one or two letters and possibly a subscript number. The subscript number denotes the operand to which the field applies.

Operation code 00 hex will never be assigned to an instruction implemented in the CPU.

Register Operands

In the RR, RRD, RRE, RRF, RX, RXE, RXF, RXY, RS, RSY, RSI, RI, RIE, and RIL formats, the contents of the register designated by the R_1 field are called the first operand. The register containing the first operand is sometimes referred to as the “first-operand location,” and sometimes as “register R_1”. In the RR, RRD, RRE, and RRF formats, the R_2 field designates the register containing the second operand, and the R_2 field may designate the same register as R_1. In the RRD, RRF, RXF, RS, RSY, RSI, and RIE formats, the use of the R_3 field depends on the instruction. In the RRF, RS, and RSY formats, the R_3 field may instead be an M_3 field specifying a mask. Certain forms of the RRF format include an M_4 field specifying a mask.

The R field designates a general or access register in the general instructions, a general register in the control instructions, and a floating-point register or a general register in the floating-point instructions.

However, in the instructions EXTRACT STACKED REGISTERS and LOAD ADDRESS EXTENDED, the R field designates both a general register and an access register, and, in the instructions LOAD CONTROL and STORE CONTROL, the R field designates a control register. (This paragraph refers only to register operands, not to the use of access registers in addressing storage operands.)

For access, floating-point, and vector registers, unless otherwise indicated in the individual instruction description, the register operand is one register in length (32 bits for an access register, 64 bits for a floating-point register, and 128 bits for a vector register), and the second operand is the same length as the first. For general and control registers, the register operand is in bit positions 32-63 of the 64-bit register or occupies the entire register, depending on the instruction.

The V field in VRI, VRR, VRS, VRV, VRX, and VSI formats, along with an extra bit designates a vector register operand.

All vector instructions have a field in bits 36-39 of the instruction labeled as RXB. This field contains the most significant bits for all of the vector register designated operands. Bits for register designations not specified by the instruction are reserved and should be set to zero; otherwise, the program may not operate compatibly in the future. The most significant bit is concatenated to the left of the four-bit register designation to create the five-bit vector register designation.

The bits of the RXB field are defined as follows:

0. Most significant bit for the vector register designation in bits 8-11 of the instruction.
1. Most significant bit for the vector register designation in bits 12-15 of the instruction.
2. Most significant bit for the vector register designation in bits 16-19 of the instruction.
3. Most significant bit for the vector register designation in bits 32-35 of the instruction.

Programming Note: Although only two RRF formats are shown in Figure 5-1, five variations are noted. This is because there are five variations in assembler-language syntax for the RRF-format instructions, as shown below.

Variation	Assembler Syntax
a	mnemonic $R_1,R_2[,R_3[,M_4]]$
b	mnemonic $R_1,R_3,R_2[,M_4]$
c	mnemonic $R_1,R_2[,M_3]$
d	mnemonic R_1,R_2,M_4
e	mnemonic $R_1,M_3,R_2[,M_4]$

Note on the Definition:

Immediate Operands

In the I format, the contents of the eight-bit immediate-data field, the I field of the instruction, are directly used as the operand.

In the SI format, the contents of the eight-bit immediate-data field, the I_2 field of the instruction, are used directly as the second operand. The B_1 and D_1 fields specify the first operand, which is one byte in length. In the SIY format, the operation is the same except that DH_1 and DL_1 fields are used instead of a D_1 field.

In the RI format for the instructions ADD HALFWORD IMMEDIATE, COMPARE HALFWORD IMMEDIATE, LOAD HALFWORD IMMEDIATE, and MULTIPLY HALFWORD IMMEDIATE, the contents of the 16-bit I_2 field of the instruction are used directly as a signed binary integer, and the R_1 field specifies the first operand, which is 32 or 64 bits in length, depending on the instruction. For the instruction TEST UNDER MASK (TMHH, TMHL, TMLH, TMLL), the contents of the I_2 field are used as a mask, and the R_1 field specifies the first operand, which is 64 bits in length.

For the instructions INSERT IMMEDIATE, AND IMMEDIATE, OR IMMEDIATE, and LOAD LOGICAL IMMEDIATE, the contents of the I_2 field are used as an unsigned binary integer or a logical value, and the R_1 field specifies the first operand, which is 64 bits in length.

For instructions in the RI, RIE, and RSI formats having an RI_2 field, or for instructions in the RIE format having an RI_4 field, the contents of the 16-bit RI_2 or RI_4 field are used as a signed binary integer designating a number of halfwords that are added to the address of the instruction to form the operand address. For instructions in the RIL format, the RI_2 field is 32 bits and is used in the same way.

For instructions in the MII format, the contents of the 12-bit RI_2 field and the 24-bit RI_3 field are used as signed binary integers designating a number of halfwords that are added to the address of the instruction to form the respective operand addresses.

For instructions in the SMI format, the contents of the 16-bit RI_2 field is used as signed binary integer designating a number of halfwords that are added to the address of the instruction to form the operand address.

For the RIE-format instructions COMPARE IMMEDIATE AND BRANCH RELATIVE and COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE, the contents of the 8-bit I_2 field are used directly as the second operand. For the RIE-format instructions COMPARE IMMEDIATE AND BRANCH, COMPARE IMMEDIATE AND TRAP, COMPARE LOGICAL IMMEDIATE AND BRANCH, and COMPARE LOGICAL IMMEDIATE AND TRAP, the contents of the 16-bit I_2 field are used directly as the second operand. For the RIE-format instructions COMPARE AND BRANCH RELATIVE, COMPARE IMMEDIATE AND BRANCH RELATIVE, COMPARE LOGICAL AND BRANCH RELATIVE, and COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE, the contents of the 16-bit RI_4 field are used as a signed binary integer designating a number of halfwords that are added to the address of the instruction to form the branch address.

For the RIL-format instructions ADD IMMEDIATE, ADD LOGICAL IMMEDIATE, ADD LOGICAL WITH SIGNED IMMEDIATE, COMPARE IMMEDIATE, COMPARE LOGICAL IMMEDIATE, LOAD IMMEDIATE, and MULTIPLY SINGLE IMMEDIATE, the contents of the 32-bit I_2 field are used directly as a the second operand.

For the RIS-format instructions, the contents of the 8-bit I_2 field are used directly as the second operand.

In the SIL format, the contents of the 16-bit I_2 field are used directly as the second operand. The B_1 and D_1 fields specify the first operand, as described below.

In the VRI-a format, the contents of the 16-bit I_2 field are used as a signed binary integer, except for VECTOR GENERATE BYTE MASK where the contents of the I_2 field are used as a bitmask. In the VRI-b format, the contents of the 8-bit I_2 and I_3 fields contain unsigned binary integers. In the VRI-c format, the I_2 field contains a 16-bit unsigned binary integer. In the VRI-d format, the contents of the 8-bit I_4 field contain an unsigned binary integer. In the VRI-e format the 12-bit I_3 field contains a bitmask.

Storage Operands

The use of B and R fields to designate access registers to refer to storage operands is described in “Access-Register-Specified Address Spaces” on page 5-46.

In the RSL, SI, SIL, SSE, and most SS formats, the contents of the general register designated by the B_1. field are added to the contents of the D_1 field to form the first-operand address. In the RS, RSY, S, SIY, SS, SSE, SSF, VRS, and VSI formats, the contents of the general register designated by the B_2 field are added to the contents of the D_2 field or DH_2 and DL_2 fields to form the second-operand address. In the RX, RXE, RXF, RXY, and VRX formats, the contents of the general registers designated by the X_2 and B_2 fields are added to the contents of the D_2 field or DH_2 and DL_2 fields to form the second-operand address. In the SMI format, the contents of the general register designated by the B_3 field are added to the contents of the D_3 field to form the third-operand address. In the RIS and RRS formats, and in one SS format, the contents of the general register designated by the B_4 field are added to the contents of the D_4 field to form the fourth-operand address.

The VRV format has a vector-element-specified index field.

When a general register contains a 24-bit or 32-bit length of a storage operand, the length is an unsigned binary integer, except that it is signed for COMPARE UNTIL SUBSTRING EQUAL, with a negative value treated as zero. Similarly, the contents of an L, L_1, or L_2 field of an instruction are an unsigned binary integer. For VRS format instructions which contain a length in a general register the length is an unsigned integer that specifies the number of additional operand bytes to the right of the byte designated by the second operand address.

In the SS format with a single, eight-bit length field, for the instructions AND (NC), EXCLUSIVE OR (XC), MOVE (MVC), MOVE NUMERICS, MOVE ZONES, and OR (OC), L specifies the number of additional operand bytes to the right of the byte designated by the first-operand address. Therefore, the length in bytes of the first operand is 1-256, corresponding to a length code in L of 0-255. Storage results replace the

first operand and are never stored outside the field specified by the address and length. In this format, the second operand has the same length as the first operand. There are variations of the preceding definition that apply to EDIT, EDIT AND MARK, PACK ASCII, PACK UNICODE, TRANSLATE, TRANSLATE AND TEST, UNPACK ASCII, and UNPACK UNICODE.

In the SS format with two length fields, in the RSL format, and in the VSI format, L_1 specifies the number of additional operand bytes to the right of the byte designated by the first-operand address. Therefore, the length in bytes of the first operand is 1-16, corresponding to a length code in L_1 of 0-15. Similarly, L_2 specifies the number of additional operand bytes to the right of the location designated by the second-operand address. Results replace the first operand and are never stored outside the field specified by the address and length. If the first operand is longer than the second, the second operand is extended on the left with zeros up to the length of the first operand. This extension does not modify the second operand in storage.

In the SS format with two R fields, as used by the MOVE TO PRIMARY, MOVE TO SECONDARY, and MOVE WITH KEY instructions, the contents of the general register specified by the R_1. field are a 32-bit unsigned value called the true length. The operands are both of a length called the effective length. The effective length is equal to the true length or 256, whichever is less. The instructions set the condition code to facilitate programming a loop to move the total number of bytes specified by the true length. The SS format with two R fields is also used to specify a range of registers and two storage operands for the LOAD MULTIPLE DISJOINT instruction and to specify one or two registers and one or two storage operands for the PERFORM LOCKED OPERATION instruction.

Address Generation

Trimodal Addressing

Bits 31 and 32 of the current PSW are the addressing-mode bits. Bit 31 is the extended-addressing-mode bit, and bit 32 is the basic-addressing-mode bit. These bits control the size of the effective address produced by address generation. When bits 31 and 32 of the current PSW both are zeros, the CPU is in the 24-bit addressing mode, and 24-bit instruction and operand effective addresses are generated. When bit 31 of the current PSW is zero and bit 32 is one, the CPU is in the 31-bit addressing mode, and 31-bit instruction and operand effective addresses are generated. When bits 31 and 32 of the current PSW are both one, the CPU is in the 64-bit addressing mode, and 64-bit instruction and operand effective addresses are generated.

Execution of instructions by the CPU involves generation of the addresses of instructions and operands. This section describes address generation as it applies to most instructions. In some instructions, the operation performed does not follow the general rules stated in this section. All of these exceptions are explicitly identified in the individual instruction descriptions.

In the ESA/390-compatibility mode, if the program issues an instruction that is defined to enable the 64-bit addressing mode, it is unpredictable whether an exception is recognized or the CPU enters the 64-bit addressing mode.

Sequential Instruction-Address Generation

When an instruction is fetched from the location designated by the current PSW, the instruction address is increased by the number of bytes in the instruction, and the instruction is executed. The same steps are then repeated by using the new value of the instruction address to fetch the next instruction in the sequence.

In the 24-bit addressing mode, instruction addresses wrap around, with the halfword at instruction address 2^{24} - 2 being followed by the halfword at instruction address 0. Thus, in the 24-bit addressing mode, any carry out of PSW bit position 104, as a result of updating the instruction address, is lost. In the 31-bit or 64-bit addressing mode, instruction addresses similarly wrap around, with the halfword at instruction address 2^{31} - 2 or 2^{64} - 2, respectively, followed by the halfword at instruction address 0. A carry out of PSW bit position 97 or 64, respectively, is lost.

Operand-Address Generation

Formation of the Intermediate Value

An operand address that refers to storage is derived from an intermediate value determined in one of the following ways:

- The intermediate value is calculated from the sum of two or three binary numbers: base address, index (when applicable), and displacement.
- The intermediate value is calculated from the sum of three binary numbers: base address, displacement, and vector element.
- The intermediate value is calculated from the sum of two binary numbers: the address of the instruction and an immediate (RI_2) field in the instruction specifying a number of halfwords.
- The intermediate value is contained in a register designated by an R field in the instruction.

Intermediate Value Designated by Base Address, Index, and Displacement: The base address (B) is a 64-bit number contained in a general register specified by the program in a four-bit field, called the B field, in the instruction. Base addresses can be used as a means of independently addressing each program and data area. In array-type calculations, it can designate the location of an array, and, in record-type processing, it can identify the record. The base address provides for addressing the entire storage. The base address may also be used for indexing.

For instructions having the RX, RXE, RXF, RXY, and VRX-formats, the index (X) is a 64-bit number contained in a general register designated by the program in a four-bit field, called the X field, in the instruction. These instructions permit double indexing; that is, the index can be used to provide the address of an element within an array. For other instruction formats which do not include an X field, the index is not applicable.

The displacement (D) is a 12-bit or 20-bit number contained in a field, called the D field, in the instruction. A 12-bit displacement is unsigned and provides for relative addressing of up to 4,095 bytes beyond the location designated by the base address. A 20-bit displacement is signed and provides for relative addressing of up to 524,287 bytes beyond the base-address location or of up to 524,288 bytes before it. In array-type calculations, the displacement can be used to specify one of many items associated with an element. In the processing of records, the displacement can be used to identify items within a record.

A 12-bit displacement is in bit positions 20-31 of instructions of certain formats (see Figure 5-1 on page 5-4). In instructions of some formats, a second 12-bit displacement also is in the instruction, in bit positions 36-47.

In the z/Architecture architectural mode, a 20-bit displacement is in instructions of only the RSY, RXY, or SIY format. In these instructions, the D field consists of a DL (low) field in bit positions 20-31 and of a DH (high) field in bit positions 32-39. When the long-displacement facility is installed, the numeric value of the displacement is formed by appending the contents of the DH field on the left of the contents of the DL field. When the long-displacement facility is not installed, the numeric value of the displacement is formed by appending eight zero bits on the left of the contents of the DL field, and the contents of the DH field are ignored.

In the ESA/390-compatibility mode, it is unpredictable whether the long-displacement facility is considered to be installed.

In forming the intermediate sum, the base address and index are treated as 64-bit binary integers. A 12-bit displacement is treated as a 12-bit unsigned binary integer, and 52 zero bits are appended on the left. A 20-bit displacement is treated as a 20-bit signed binary integer, and 44 bits equal to the sign bit are appended on the left. The three are added as 64-bit binary numbers, ignoring overflow. The sum is always 64 bits long and is used as an intermediate value to form the generated address. The bits of the intermediate value are numbered 0-63.

A zero in any of the B_1, B_2, X_2, B_3, or B_4 fields indicates the absence of the corresponding address component. For the absent component, a zero is used in forming the intermediate sum, regardless of the contents of general register 0. A displacement of zero has no special significance.

Intermediate Value Designated by Base Address, Displacement, and Vector Element: For VRV format instructions, a vector element is used in the formation of the intermediate value. This vector element is an unsigned binary integer value that is added to the base address and 12-bit displacement to form a

64-bit intermediate sum. The vector element is designated by a vector register and an element index. A zero V field accesses the element in vector register zero and does not imply a zero value.

Intermediate Value Designated by an Immediate Field: For the following instructions, the intermediate value is determined using an immediate (RI_2) field in the instruction:

- BRANCH PREDICTION PRELOAD
- BRANCH PREDICTION RELATIVE PRELOAD
- COMPARE HALFWORD RELATIVE LONG
- COMPARE LOGICAL RELATIVE LONG
- COMPARE RELATIVE LONG
- EXECUTE RELATIVE LONG
- LOAD ADDRESS RELATIVE LONG
- LOAD HALFWORD RELATIVE LONG
- LOAD LOGICAL HALFWORD RELATIVE LONG
- LOAD LOGICAL RELATIVE LONG
- LOAD RELATIVE LONG
- PREFETCH DATA RELATIVE LONG
- STORE HALFWORD RELATIVE LONG
- STORE RELATIVE LONG

For BRANCH PREDICTION PRELOAD, the RI_2 field contains a 16-bit signed binary integer; for BRANCH PREDICTION RELATIVE PRELOAD, the RI_2 field contains a 12-bit signed binary integer; and for each of the other instructions listed, the RI_2 field contains a 32-bit signed binary integer. The signed binary integer specifies the number of halfwords that is added to the address of the current instruction (or the address of the execute-type instruction if the instruction having the RI_2 field is the target of an execute-type instruction) to form the intermediate value.

When DAT is on, and the intermediate value is designated by an immediate field, the resulting operand address is in the same address space as that used to fetch instructions.

Intermediate Value in a Register Designated by an R Field: When an instruction description specifies that the contents of a general register designated by an R field are used to address an operand in storage, the register contents are used as the 64-bit intermediate value.

An instruction can designate the same general register both for address computation and as the location of an operand. Address computation is completed before registers, if any, are changed by the operation.

Formation of the Operand Address

Unless otherwise indicated in an individual instruction definition, the generated operand address designates the leftmost byte of an operand in storage.

The generated operand address is always 64 bits long, and the bits are numbered 0-63. The manner in which the generated address is obtained from the intermediate value depends on the current addressing mode. In the 24-bit addressing mode, bits 0-39 of the intermediate value are ignored, bits 0-39 of the generated address are forced to be zeros, and bits 40-63 of the intermediate value become bits 40-63 of the generated address. In the 31-bit addressing mode, bits 0-32 of the intermediate value are ignored, bits 0-32 of the generated address are forced to be zero, and bits 33-63 of the intermediate value become bits 33-63 of the generated address. In the 64-bit addressing mode, bits 0-63 of the intermediate value become bits 0-63 of the generated address.

Programming Note: Since a carry out of the most-significant address bit is ignored, negative values may be used in index and base-address registers. Bits 0-32 of these values are ignored in the 31-bit addressing mode, and bits 0-39 are ignored in the 24-bit addressing mode.

For instructions having the RSY, RXY, or SIY format, a negative value may be used for the displacement.

When the intermediate value is designated by an immediate field in the instruction, the immediate field may contain a negative value.

Branch-Address Generation

Formation of the Intermediate Value

For branch instructions, the address of the next instruction to be executed when the branch is taken is called the branch address. Depending on the branch instruction, the instruction format may be RR, RRE, RX, RXY, RS, RSY, RSI, RI, RIE, or RIL.

Except as noted below, in the RS, RSY, RX, and RXY formats, the branch address is specified by a base address, a displacement, and, in the RX and RXY formats, an index. In the RXY-format instruction BRANCH INDIRECT ON CONDITION, the branch address is formed from the contents of the eight-byte second operand in storage. For the RXY-format

LOAD GUARDED and LOAD LOGICAL AND SHIFT GUARDED instructions, when a guarded-storage event is recognized, the contents of the guarded-storage-event-handler address (in the guarded-storage-event parameter list) forms the branch address. In these formats, the generation of the intermediate value follows the same rules as for the generation of the operand-address intermediate value.

In the RR and RRE formats, the contents of the general register designated by the R_2 field are used as the intermediate value from which the branch address is formed. Unless otherwise specified, general register 0 cannot be designated as containing a branch address, and a value of zero in the R_2 field causes the instruction to be executed without branching.

The relative-branch instructions are in the RSI, RI, RIE, and RIL formats. In the RSI, RI, and RIE formats for the relative-branch instructions, the contents of the I_2 field are treated as a 16-bit signed binary integer designating a number of halfwords. In the RIL format, the contents of the I_2 field are treated as a 32-bit signed binary integer designating a number of halfwords. The branch address is the number of halfwords designated by the I_2 field added to the address of the relative-branch instruction.

The 64-bit intermediate value for a relative branch instruction in the RSI, RI, RIE, or RIL format is the sum of two addends, with overflow from bit position 0 ignored. In the RSI, RI, or RIE format, the first addend is the contents of the I_2 field with one zero bit appended on the right and 47 bits equal to the sign bit of the contents appended on the left, except that for COMPARE AND BRANCH RELATIVE, COMPARE IMMEDIATE AND BRANCH RELATIVE, COMPARE LOGICAL AND BRANCH RELATIVE and COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE, the first addend is the contents of the I_4 field, with bits appended as described above for the I_2 field. In the RIL format, the first addend is the contents of the I_2 field with one zero bit appended on the right and 31 bits equal to the sign bit of the contents appended on the left. In all formats, the second addend is the 64-bit address of the branch instruction. The address of the branch instruction is the instruction address in the PSW before that address is updated to address the next sequential instruction, or it is the address of the target of the execute-type instruction (EXECUTE or EXECUTE RELATIVE LONG) if an execute-type instruction is used. If an execute-type instruction is used in the 24-bit or 31-bit addressing mode, the address of the branch instruction is the target address with 40 or 33 zeros, respectively, appended on the left.

Formation of the Branch Address

The branch address is always 64 bits long, with the bits numbered 0-63. The branch address replaces bits 64-127 of the current PSW.

The manner in which the branch address is obtained from the intermediate value depends on the addressing mode. For those branch instructions which change the addressing mode, the new addressing mode is used. In the 24-bit addressing mode, bits 0-39 of the intermediate value are ignored, bits 0-39 of the branch address are made zeros, and bits 40-63 of the intermediate value become bits 40-63 of the branch address. In the 31-bit addressing mode, bits 0-32 of the intermediate value are ignored, bits 0-32 of the branch address are made zeros, and bits 33-63 of the intermediate value become bits 33-63 of the branch address. In the 64-bit addressing mode, bits 0-63 of the intermediate value become bits 0-63 of the branch address.

For several branch instructions, branching depends on satisfying a specified condition. When the condition is not satisfied, the branch is not taken, normal sequential instruction execution continues, and the branch address is not used. When a branch is taken, bits 0-63 of the branch address replace bits 64-127 of the current PSW. The branch address is not used to access storage as part of the branch operation.

A specification exception due to an odd branch address and access exceptions due to fetching of the instruction at the branch location are not recognized as part of the branch operation but instead are recognized as exceptions associated with the execution of the instruction at the branch location.

A branch instruction, such as BRANCH AND SAVE, can designate the same general register for branch-address computation and as the location of an operand. Branch-address computation is completed before the remainder of the operation is performed.

Some models may maintain information in internal tables about branch instructions and branch addresses for internal branch-prediction logic. Additionally, since the formation of branch addresses is similar to the formation of the target addresses of execute-type instructions, some models may also

maintain information about execute-type instructions and their execute targets in the same or similar internal tables. A model may also provide the BRANCH PREDICTION PRELOAD and BRANCH PREDICTION RELATIVE PRELOAD instructions which allow the program to identify branch instructions and associated branch addresses, and to identify execute-type instructions and associated execute targets, thereby affecting the preloading of such branch-prediction and execute-prediction information into such internal tables.

Instruction Execution and Sequencing

The program-status word (PSW), described in Chapter 4 “Control” contains information required for proper program execution. The PSW is used to control instruction sequencing and to hold and indicate the status of the CPU in relation to the program currently being executed. The active or controlling PSW is called the current PSW.

Branch instructions perform the functions of decision making, loop control, and subroutine linkage. A branch instruction affects instruction sequencing by introducing a new instruction address into the current PSW. The relative-branch instructions with a 16-bit I_2 field allow branching to a location at an offset of up to plus 64K - 2 bytes or minus 64K bytes relative to the location of the branch instruction, without the use of a base register. The relative-branch instructions with a 32-bit I_2 field allow branching to a location at an offset of up to plus 4G - 2 bytes or minus 4G bytes relative to the location of the branch instruction, without the use of a base register.

Decision Making

Facilities for decision making are provided by the BRANCH ON CONDITION, BRANCH RELATIVE ON CONDITION, and BRANCH RELATIVE ON CONDITION LONG instructions. These instructions inspect a condition code that reflects the result of a majority of the arithmetic, logical, and I/O operations. The condition code, which consists of two bits, provides for four possible condition-code settings: 0, 1, 2, and 3.

The specific meaning of any setting depends on the operation that sets the condition code. For example, the condition code reflects such conditions as zero, nonzero, first operand high, equal, overflow, and subchannel busy. Once set, the condition code remains unchanged until modified by an instruction that causes a different condition code to be set. See Appendix C, “Condition-Code Settings” for a summary of the instructions which set the condition code.

Loop Control

Loop control can be performed by the use of BRANCH ON CONDITION, BRANCH RELATIVE ON CONDITION, and BRANCH RELATIVE ON CONDITION LONG to test the outcome of address arithmetic and counting operations. For some particularly frequent combinations of arithmetic and tests, BRANCH ON COUNT, BRANCH ON INDEX HIGH, and BRANCH ON INDEX LOW OR EQUAL are provided, and relative-branch equivalents of these instructions are also provided. These branches, being specialized, provide increased performance for these tasks.

Subroutine Linkage without the Linkage Stack

This section describes only the methods for subroutine linkage that do not use the linkage stack. For the linkage extensions provided by the linkage stack, see “Linkage-Stack Introduction” on page 5-70. (Those extensions include a different method of operation of the PROGRAM CALL instruction and also the BRANCH AND STACK and PROGRAM RETURN instructions.)

Simple Branch Instructions

Subroutine linkage when a change of the addressing mode is not required is provided by the BRANCH AND LINK and BRANCH AND SAVE instructions. (This discussion of BRANCH AND SAVE applies also to BRANCH RELATIVE AND SAVE and BRANCH RELATIVE AND SAVE LONG.) Both of these instructions permit not only the introduction of a new instruction address but also the preservation of a return address and associated information. The return address is the address of the instruction following the branch instruction in storage, except that it is the address of the instruction following an execute-

type instruction that has the branch instruction as its target.

Both BRANCH AND LINK and BRANCH AND SAVE have an R_1 field. They form a branch address by means of fields that depend on the instruction. The operations of the instructions are summarized as follows:

- In the 24-bit addressing mode, both instructions place the return address in bit positions 40-63 of general register R_1 and leave bits 0-31 of that register unchanged. BRANCH AND LINK places the instruction-length code for the instruction and also the condition code and program mask from the current PSW in bit positions 32-39 of general register R_1. BRANCH AND SAVE places zeros in those bit positions.
- In the 31-bit addressing mode, both instructions place the return address in bit positions 33-63 and a one in bit position 32 of general register R_1, and they leave bits 0-31 of the register unchanged.
- In the 64-bit addressing mode, both instructions place the return address in bit positions 0-63 of general register R_1.
- In any addressing mode, both instructions generate the branch address under the control of the current addressing mode. The instructions place bits 0-63 of the branch address in bit positions 64-127 of the PSW. In the RR format, both instructions do not perform branching if the R_2 field of the instruction is zero.

It can be seen that, in the 24-bit or 31-bit addressing mode, BRANCH AND SAVE places the basic-addressing-mode bit, bit 32 of the PSW, in bit position 32 of general register R_1. BRANCH AND LINK does so in the 31-bit addressing mode.

The instructions BRANCH AND SAVE AND SET MODE and BRANCH AND SET MODE are for use when a change of the addressing mode is required during linkage. These instructions have R_1 and R_2 fields. The operations of the instructions are summarized as follows:

- BRANCH AND SAVE AND SET MODE sets the contents of general register R_1 the same as BRANCH AND SAVE. In addition, the instruction places the extended-addressing-mode bit, bit 31 of the PSW, in bit position 63 of the register.
- BRANCH AND SET MODE, if R_1 is nonzero, performs as follows. In the 24- or 31-bit mode, it places bit 32 of the PSW in bit position 32 of general register R_1, and it leaves bits 0-31 and 33-63 of the register unchanged. Note that bit 63 of the register should be zero if the register contains an instruction address. In the 64-bit mode, the instruction places bit 31 of the PSW (a one) in bit position 63 of general register R_1, and it leaves bits 0-62 of the register unchanged.
- When R_2 is nonzero, both instructions set the addressing mode and perform branching as follows. Bit 63 of general register R_2 is placed in bit position 31 of the PSW. If bit 63 is zero, bit 32 of the register is placed in bit position 32 of the PSW. If bit 63 is one, PSW bit 32 is set to one. Then the branch address is generated from the contents of the register, except with bit 63 of the register treated as a zero, under the control of the new addressing mode. The instructions place bits 0-63 of the branch address in bit positions 64-127 of the PSW. Bit 63 of general register R_2 remains unchanged and, therefore, may be one upon entry to the called program. If R_2 is the same as R_1, the results in the designated general register are as specified for the R_1 register.

The operations of the simple branch instructions are summarized in Figure 5-2 on page 5-16. For contrast, the figure also shows the BRANCH ON COUNT instruction, which is not for use in linkage, and the LOAD ADDRESS and LOAD ADDRESS EXTENDED instructions.

Programming Notes:

1. A called program that is entered in the 64-bit addressing mode can use bit 63 of the entry-point register to determine the instruction used to perform the call and, thus, the instruction that must be used to perform the return linkage. If bit 63 is zero, BRANCH AND SAVE (BAS or BASR) (or possibly BAL, BALR, BRAS, or BRASL) was used, the addressing mode of the caller is not indicated in the return register, and BRANCH ON CONDITION (BCR) must be used to return without changing the addressing mode during the return. If bit 63 of the entry-point register is one, BASSM or BSM was used, the addressing mode of the caller is indicated in the return register (or at least can be, in the case of BSM), and BSM must be used to return and restore the addressing mode of the caller.

Instruction	Format	In Mode	Address Placed in GR R_1				Branch or 2nd-Op Address		R_2 Bit 63	PSW Bit 31 Set to	PSW Bit 32 Set to
			Bits 0-31	Bit 32	Bits 33-62	Bits 63	Bits 0-32	Bits 33-63			
BALR*/BAL	RR/RX	24	U	***	***	IA	SIA	SIA	LSExc	U	U
		31	U	BAM	IA	IA	SIA	SIA	LSExc	U	U
		64	IA	IA	IA	IA	SIA	SIA	LSExc	U	U
BASR*/BAS/ BRAS/BRASL	RR/RX/ RI/RIL	24/31	U	BAM	IA	IA	SIA	SIA	LSExc	U	U
		64	IA	IA	IA	IA	SIA	SIA	LSExc	U	U
BASSM*	RR	24/31	U	BAM	IA	IA	SIA	SIA	0	0	$R_2$32
		24/31	U	BAM	IA	IA	SIA	SIA	1G0	1	1
		64	IA	IA	IA	1	SIA	SIA	0	0	$R_2$32
		64	IA	IA	IA	1	SIA	SIA	1G0	1	1
BSM**	RR	24/31	U	BAM	U	U	SIA	SIA	0	0	$R_2$32
		24/31	U	BAM	U	U	SIA	SIA	1G0	1	1
		64	U	U	U	1	SIA	SIA	0	0	$R_2$32
		64	U	U	U	1	SIA	SIA	1G0	1	1
BCTR*/BCT/ BCTGR*/BCTG	RR/RX/ RRE/RXY	24/31	NLA	NLA	NLA	NLA	SIA	SIA	LSExc	U	U
		64	NLA	NLA	NLA	NLA	SIA	SIA	LSExc	U	U
LA/LAE	RX/RX	24/31	U	0	Op2Ad	Op2Ad	FZ	SR1	0/1	U	U
		64	Op2Ad	Op2Ad	Op2Ad	Op2Ad	SR1	SR1	0/1	U	U

Explanation:

—	The address does not exist, or the bit has no special effect.
*	The action associated with the R_2 field is not performed if the field is zero.
**	The action associated with the R_1 or R_2 field is not performed if the field is zero.
***	The instruction-length code, condition code, and program mask are saved in bit positions 32-39 of the link address, and bits 40-63 of the updated instruction address are saved in bit positions 40-63.
0/1	Bit 63 can be zero or one.
1G0	Bit 63 is one and is left one, but the branch address is generated as if the bit is zero.
BAM	Bit 32 of the link address is set with the basic-addressing-mode bit, bit 32 of the PSW.
FZ	Bits 0-32 of the second-operand address are forced to zeros in the 24-bit or 31-bit addressing mode.
IA	Bits of the link address are set with the updated instruction address as shown.
LSExc	A late specification exception is recognized if the bit is one.
NLA	The instruction does not produce a link address. (The instruction is shown simply as an example of a non-linkage branch instruction.)
Op2Ad	Bits of the address in general register R_1 are set with the corresponding bits of the second-operand address as shown.
$R_2$32	The basic-addressing-mode bit, bit 32 of the PSW, is set with bit 32 of general register R_2
SIA	Bits 0-63 of the branch address are used to set the instruction address in the PSW. Bits 0-39 of the branch address are forced to zeros in the 24-bit addressing mode. Bits 0-32 are forced to zeros in the 31-bit addressing mode.
SR1	Bits of the second-operand address are used to set the corresponding bits of the address in the R_1 general register as shown. Bits 0-39 of the second-operand address are forced to zeros in the 24-bit addressing mode. Bits 0-32 are forced to zeros in the 31-bit addressing mode.
U	Unchanged.

Figure 5-2. Summary of Simple Branch Linkage Instructions and Other Instructions

2. When BSM is executed in the 24-bit or 31-bit addressing mode and used in a forward linkage to set the 64-bit mode, and the R_1 and R_2 of the instruction are the same value, bit 63 of the designated general register does not, upon entry to the called program, correctly indicate the mode of the calling program. (The bit is one, instead of zero, because the program set bit 63 of the R_2 register to one and the instruction does not change bit 63 of the R_1 register in the 24- or 31-bit mode.) BASSM always correctly indicates the addressing mode of the calling program.

3. If an entry point can be branched to in the 64-bit addressing mode either by BAS or BASR or by BASSM or BSM, and a USING statement provides addressability to the entry point, bit 63 of the entry-point register must be zeroed to ensure compatible operation regardless of whether the linkage instruction changes the addressing mode. For example, if general register 15 is the entry-point register, the following may be used to set bit 63 to zero.

   ```
   NILL    15,X'FFFE'
   ```

   If the entry point can be branched to in the 64-bit addressing mode only by BASSM or BSM, the entry-point USING statement can accommodate bit 63 being one – without having to zero the bit – as shown below.

   ```
   USING  *+1,15
   ```

Other Linkage Instructions

Note: The discussion in this section of PROGRAM TRANSFER applies equally to PROGRAM TRANSFER WITH INSTANCE, which differs from PROGRAM TRANSFER only as described in “ASN-Second-Table-Entry Instance Number and ASN Reuse” on page 3-25.

PROGRAM TRANSFER WITH INSTANCE is not further referred to in this section.

Linkage between a problem-state program and the supervisor or monitoring program is provided by means of the SUPERVISOR CALL and MONITOR CALL instructions.

The instructions PROGRAM CALL and PROGRAM TRANSFER provide the facility for linkage between programs of different authority and in different address spaces. PROGRAM CALL permits linkage to a number of preassigned programs that may be in either the problem or the supervisor state and may be in either the same address space or an address space different from that of the caller. It permits a change between the 24-bit and 31-bit addressing modes, and it permits an increase of PSW-key-mask authority, which authorizes the execution of the SET PSW KEY FROM ADDRESS instruction and also other functions. In general, PROGRAM CALL is used to transfer control to a program of higher authority. PROGRAM TRANSFER permits a change of the instruction address and address space and a change between the 24-bit and 31-bit addressing modes. PROGRAM TRANSFER also permits a reduction of PSW-key-mask authority and a change from the supervisor to the problem state. In general, it is used to transfer control from one program to another of equal or lower authority.

When a calling linkage is to increase authority, the calling linkage can be performed by PROGRAM CALL and the return linkage by PROGRAM TRANSFER. Alternatively, when the calling linkage is to decrease authority, the calling linkage can be performed by PROGRAM TRANSFER and the return linkage by PROGRAM CALL.

The operation of PROGRAM CALL is controlled by means of an entry-table entry, which is located as part of a table-lookup process during the execution of the instruction. The entry-table entry specifies either a basic (nonstacking) operation or the stacking operation described in “Linkage-Stack Introduction” on page 5-70. The instruction causes the primary address space to be changed only when the ASN in the entry-table entry is nonzero. When the primary address space is changed, the operation is called PROGRAM CALL with space switching (PC-ss). When the primary address space is not changed, the operation is called PROGRAM CALL to current primary (PC-cp).

PROGRAM TRANSFER specifies the address space which is to become the new primary address space. When the primary address space is changed, the operation is called PROGRAM TRANSFER with space switching (PT-ss). When the primary address space is not changed, the operation is called PROGRAM TRANSFER to current primary (PT-cp).

Basic PROGRAM CALL, and PROGRAM TRANSFER, can be executed successfully in either a basic (24-bit or 31-bit) addressing mode or the extended (64-bit) addressing mode. They do not provide a change between a basic addressing mode and the extended addressing mode.

The BRANCH AND SET AUTHORITY instruction can improve performance by replacing a PT-cp instruction used to perform a calling linkage in which PSW-key-mask authority is reduced, and by replacing a PC-cp instruction used to perform the associated return linkage in which PSW-key-mask authority is restored. BRANCH AND SET AUTHORITY also permits changes between the supervisor and problem states, and it can replace SET PSW KEY FROM ADDRESS by changing the PSW key during the link-

age. The calling-linkage operation is called BRANCH AND SET AUTHORITY in the base-authority state (BSA-ba), and the return-linkage operation is called BRANCH AND SET AUTHORITY in the reduced-authority state (BSA-ra).

The BRANCH IN SUBSPACE GROUP instruction allows linkage within a group of address spaces called a subspace group, where one address space in the group is called the base space and the others are called subspaces. It is intended that each subspace contain a different subset of the storage in the base space, that the base space and each subspace contain a subsystem control program, such as CICS, and application programs, and that each subspace contain the data for a single transaction being processed under the subsystem control program. The placement of the data for each transaction in a different subspace prevents a program that is being executed to process one particular transaction from erroneously damaging the data of other transactions. It is intended that the primary address space be the base space when the control program is being executed, and that it be the subspace for a transaction when an application program is being executed to process that transaction. BRANCH IN SUBSPACE GROUP changes not only the instruction address in the PSW but also the primary address-space-control element in control register 1. BRANCH IN SUBSPACE GROUP does not change the primary ASN in control register 4 or the primary-ASN-second-table-entry origin in control register 5, and, therefore, the base space and the subspaces all are associated with the same ASN, and the programs in those address spaces all are of equal authority.

Although a subspace is intended to be a subset of the base space as described above, BRANCH IN SUBSPACE GROUP does not require this, and the instruction may be useful in ways other than as described above.

BRANCH IN SUBSPACE GROUP uses an access-list-entry token (ALET) in an access register as an identifier of the address space that is to receive control. The instruction saves the updated instruction address to permit a return linkage, but it does not save an identifier of the address space from which control was transferred. However, an ALET equal to 00000000 hex, called ALET 0, can be used to return from a subspace to the base space, and an ALET equal to 00000001 hex, called ALET 1, can be used to return from the base space to the subspace that last had control.

The SET ADDRESSING MODE (SAM24, SAM31, SAM64) instruction can assist in linkage by setting the 24-bit, 31-bit, or 64-bit addressing mode either before or after a linkage operation.

The RESUME PROGRAM instruction is intended for use by a problem-state interruption-handling program to return to the interrupted program. The interruption-handling program can use LOAD ACCESS MULTIPLE and LOAD MULTIPLE instructions to restore the contents of the interrupted program's access and general registers from a save area, except for the contents of one access-and-general register pair. The interruption-handling program then can use RESUME PROGRAM to restore the contents of certain PSW fields, including the instruction address, and also the contents of the remaining access-and-general pair from the save area, with that pair first being used by RESUME PROGRAM to address the save area.

The TRAP instruction (TRAP2, TRAP4) can overlay instructions in an application program and give control to a trap program for performing fix-ups of data used by the application program. The RESUME PROGRAM instruction can be used to return control from the trap program to the application program.

The linkage instructions provided and the functions performed by each are summarized in Figure 5-3 on page 5-19.

Programming Note: This note describes the simple branch-type linkage instructions that were included in 370-XA and carried forward to ESA/370, ESA/390, and z/Architecture. To give the reader a better understanding of the utility and intended usage of these linkage instructions, the following paragraphs in this note describe various program linkages and conventions and the use of the linkage instructions in these situations.

The linkage instructions were originally provided to permit System/370 programs to operate with no modification or only slight modification on 370/XA (and successor) systems and also to provide additional function for those programs which were designed to take advantage of the 31-bit addressing of 370/XA. The instructions provided the capability for both old and new programs to coexist in storage and to communicate with each other. The instructions now have been enhanced to permit usage of the 64-bit addressing of z/Architecture.

Instruction	Format	Instruction Address PSW Bits 64-127		Basic Addr. Mode PSW Bit 32		Extended Addr. Mode PSW Bit 31		Problem State PSW Bit 15		PASN CR4 Bits 48-63		PSW-Key Mask Changed in CR3	Trace
		Save	Set	Save	Set	Save	Set	Save	Set	Save	Set		
BALR	RR	Yes*	R_2[1]	BAM31	-	-	-	-	-	-	-	-	R_2[1]
BAL	RX	Yes*	Yes*	BAM31	-	-	-	-	-	-	-	-	-
BASR	RR	Yes	R_2[1]	BAM	-	-	-	-	-	-	-	-	R_2[1]
BAS	RX	Yes	Yes	BAM	-	-	-	-	-	-	-	-	-
BASSM	RR	Yes	R_2[1]	BAM	R_2[1]	Yes	R_2[1]	-	-	-	-	-	-
BRAS	RI	Yes	Yes	BAM	-	-	-	-	-	-	-	-	-
BRASL	RIL	Yes	Yes	BAM	-	-	-	-	-	-	-	-	-
BSA-ba	RRE	Yes	Yes	BAM	BAM	-	-	Yes	Yes[4]	-	-	"AND" R_1[5]	Yes
BSA-ra	RRE	R_1[1]	Yes	R_1[1] BAM	BAM			-	Yes	-	-	Yes	Yes
BSG	RRE	Yes	Yes	R_1[1] BAM	BAM	-	-	-	-	-	-[3]	-	Yes
BSM	RR	-	R_2[1]	R_1[1] BAM	R_2[1]	R_1[1] EAM64	R_2[1]	-	-	-	-	-	-
MC#[2]	SI	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-	-	-	-
PC-cp	S	Yes	Yes	BAM	BAM	-	-	Yes	Yes	-	-	"OR" EKM	Yes
PC-ss	S	Yes	Yes	BAM	BAM	-	-	Yes	Yes	Yes	Yes	"OR" EKM	Yes
PT-cp or PTI-cp	RRE	-	R_2	-	R_2 BAM	-	-	-	R_2**	-	-	"AND" R_1	Yes
PT-ss or PTI-ss	RRE	-	R_2	-	R_2 BAM	-	-	-	R_2**	-	Yes	"AND" R_1	Yes
RP	S	-	Yes	-	Yes	-	Yes	-	-	-	-	-	Yes
SAM24	E	-	-	-	Yes 0	-	Yes 0	-	-	-	-	-	Yes
SAM31	E	-	-	-	Yes 1	-	Yes 0	-	-	-	-	-	Yes
SAM64	E	-	-	-	Yes 1	-	Yes 1	-	-	-	-	-	Yes
SVC2	RR	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-	-	-	-
TRAP2	E	Yes	Yes	Yes	Yes	Yes	-	Yes	-	-	-	-	Yes
TRAP4	S	Yes	Yes	Yes	Yes	Yes	-	Yes	-	-	-	-	Yes

Figure 5-3. Summary of Linkage Instructions without the Linkage Stack

Instruction	Format	Instruction Address PSW Bits 64-127		Basic Addr. Mode PSW Bit 32		Extended Addr. Mode PSW Bit 31		Problem State PSW Bit 15		PASN CR4 Bits 48-63		PSW-Key Mask Changed in CR3	Trace
		Save	Set	Save	Set	Save	Set	Save	Set	Save	Set		

Explanation:

- No
- * In the 24-bit addressing mode, the instruction-length code, condition code, and program mask are saved in bit positions 32-39 of the R_1 general register.
- ** A change from the supervisor to the problem state is allowed; a privileged-operation exception is recognized when a change from the problem to the supervisor state is specified.
- # Monitor-mask bits provide a means of disallowing linkage, or enabling linkage, for selected classes of events. When the enhanced-monitor facility is installed, the enhanced-monitor-mask bits are also used in controlling linkage (by means of a monitor-event program interruption).
- [1] The action takes place only if the associated R field in the instruction is nonzero.
- [2] MC and SVC, as part of the interruption, save the entire current PSW and load a new PSW.
- [3] The primary address-space-control element is set even though the PASN is not set.
- [4] The problem state is set
- [5] The PSW key also is set from general register R_1
- BAM The basic-addressing-mode bit is saved or set only in the 24-bit or 31-bit addressing mode.
- BAM31 The basic-addressing-mode bit is saved only in the 31-bit addressing mode.
- EAM64 The extended-addressing-mode bit is saved only in the 64-bit addressing mode.
- R_1 The field or bit is saved in general register R_1.
- R_2 The field or bit is set from general register R_2.

Figure 5-3. Summary of Linkage Instructions without the Linkage Stack

With respect to System/370 programs, it is assumed that old, unmodified programs operate in the 24-bit addressing mode and call, or directly communicate with, other programs operating in the 24-bit addressing mode only. Modified programs normally operate in the 24-bit addressing mode but may have called programs which operate in either the 24-bit or 31-bit addressing mode. They and also modified 370-XA, ESA/370, and ESA/390 programs now may call programs that operate in the 24-bit, 31-bit, or 64-bit addressing mode. New programs may be written to operate in any addressing mode, and, in some cases, a program may be written such that it can be invoked in any addressing mode.

BRANCH AND SAVE AND SET MODE (BASSM) is intended to be the principal calling instruction to subroutines outside of an assembler/linkage-editor control section (CSECT), for use by all new programs and particularly by programs that must change the addressing mode during the linkage. The calling sequence has normally been:

```
         L       15,ACON
         BASSM   14,15
         …
         EXTRN   SUB
ACON     DC      A(X'80000000'+SUB)
```

where ACON is an A-type address constant, and the X'80000000' should be present to give control in the 31-bit addressing mode or should be omitted to give control in the 24-bit addressing mode.

The return from such a routine normally is:

```
         BSM     0,14
```

It is assumed that the A-type address constant will be extended so it may be an eight-byte field containing a 64-bit entry-point address, with bit 63 of the address indicating, when one, that the entry is in the 64-bit addressing mode. This extended constant is shown here as “ACONE”. The calling sequence would normally be:

```
         LG      15,ACONE
         BASSM   14,15
         …
         EXTRN   SUB
ACONE    DC      AD(X'1'+SUB)
```

The return from such a routine would normally be:

```
         BSM     0,14
```

When a change of the addressing mode is not required, BRANCH AND LINK or BRANCH AND SAVE should be used instead of BASSM.

The BRANCH AND LINK (BAL, BALR) instruction is provided primarily for compatibility with System/370. It is defined to operate in the 31-bit and 64-bit addressing modes to increase the probability that an old, straightforward program can be modified to operate in those addressing modes with minimal or no change. It is recommended, however, that BRANCH AND SAVE (BAS and BASR) be used instead and that BRANCH AND LINK be avoided since it places nonzero information in bit positions 32-39 of the general register in the 24-bit addressing mode, which may lead to problems and may decrease performance.

BRANCH RELATIVE AND SAVE and BRANCH RELATIVE AND SAVE LONG may be used instead of BRANCH AND SAVE.

It is assumed that the normal return from a subroutine called in the 24-bit or 31-bit addressing mode by BRANCH AND LINK (BAL or BALR) will be:

```
BCR     15,14
```

However, the standard “return instruction”:

```
BSM     0,14
```

operates correctly for all cases except for a calling BAL executed in the 24-bit addressing mode. In the 24-bit addressing mode, BAL causes an ILC of 10 to be placed in bit positions 32 and 33 of the link register. Thus, a BSM would return in the 31-bit addressing mode. Note that an EXECUTE of BALR in the 24-bit addressing mode also causes the same ILC effect; an EXECUTE RELATIVE LONG of BALR in the 24-bit addressing mode causes a similar ILC effect (the ILC is 11 binary).

The BRANCH AND SAVE (BAS, BASR) instruction is provided to be used for subroutine linkage to any program either within the same CSECT or known to be in the same addressing mode. BASR with the R_2 field 0 is also useful for obtaining addressability to the instruction stream by getting a 31-bit address, uncluttered by leftmost fields, in the 24-bit addressing mode.

The instruction for returning from a routine called in the 24-bit or 31-bit addressing mode by BRANCH AND SAVE (BAS or BASR) may be either:

```
BCR     15,14
```

or:

```
BSM     0,14
```

The instruction for returning from a routine called in the 64-bit addressing mode by BAS or BASR must be BCR; BSM would set the 24-bit or 31-bit addressing mode, depending on bit 32 of the link register (an address bit), because bit 63 of the link register (the rightmost bit of an instruction address) is zero. BSM can always be used as the return instruction if BASSM is used as the calling instruction.

In some cases, it may be desirable to rewrite a program that is called by an old program which has not been rewritten. In such a case, the old program, which operates in the 24-bit or 31-bit addressing mode, will be given the address of an intermediate program that will set up the correct entry and return modes and then call the rewritten program. Such an intermediate program is sometimes referred to as a *glue module.* The instruction BRANCH AND SET MODE (BSM) with a nonzero R_1 field provides the function necessary to perform this operation efficiently. This is shown in Figure 5-4 on page 5-22 for a linkage from a 24-bit-mode program to a 31-bit-mode program.

Note that the “BSM 14,15” in the glue module causes either an indication of the 64-bit addressing mode to be saved in bit position 63 of general register 14 or an indication of one of the 24-bit and 31-bit addressing modes to be saved in bit position 32 of the register, and that the other bits of the register are unchanged. Thus, when “BSM 0,14” is executed in the new program, control passes directly back to the old program without passing through the glue module again. The glue module could give control to a program in the 64-bit addressing mode and possibly above the 2 G-byte boundary by loading an eight-

byte A-type address constant, with bit 63 set to one, instead of a four-byte A-type address constant.

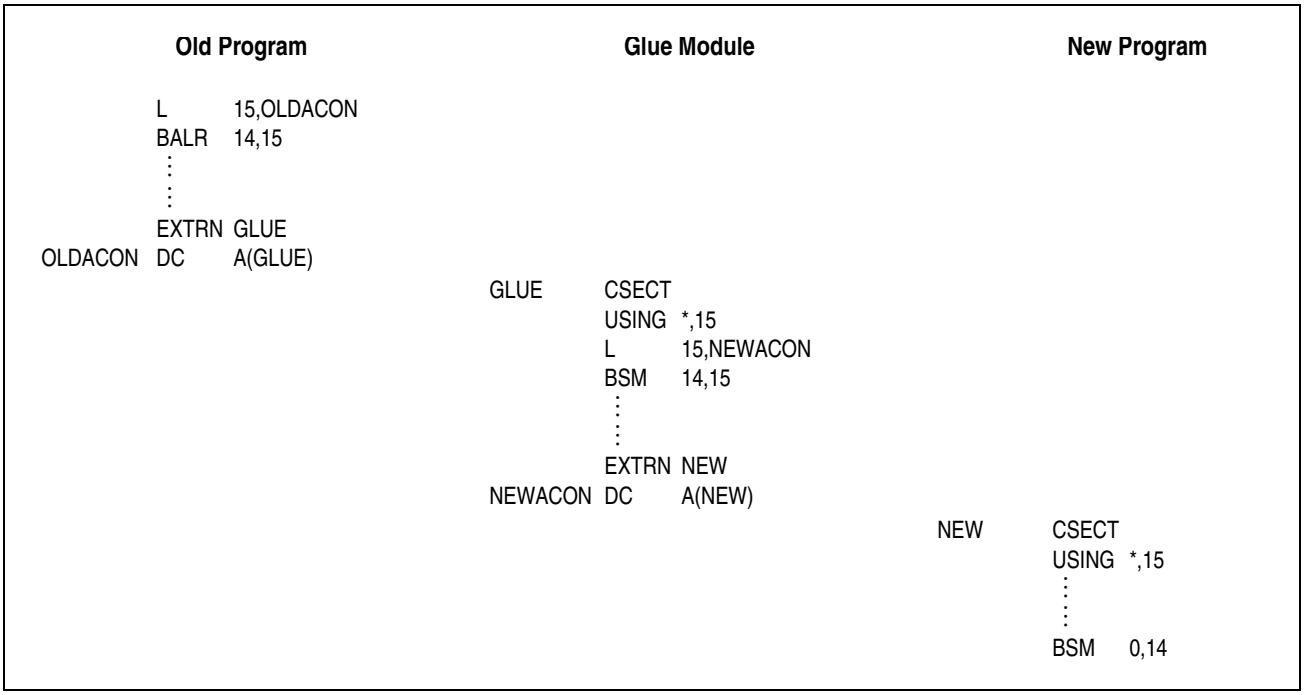


Figure 5-4. Glue Module for Linkage from the 24-Bit Mode to the 31-Bit Mode

Interruptions

Interruptions permit the CPU to change state as a result of conditions external to the system, in subchannels or input/output (I/O) devices, in other CPUs, or in the CPU itself. Details are to be found in Chapter 6, “interruptions”.

Six classes of interruption conditions are provided: external, I/O, machine check, program, restart, and supervisor call. Each class has two related PSWs, called old and new, in permanently assigned real storage locations. In all classes, an interruption involves storing information identifying the cause of the interruption, storing the current PSW at the old-PSW location, and fetching the PSW at the new-PSW location, which becomes the current PSW.

The old PSW contains CPU-status information necessary for resumption of the interrupted program. At the conclusion of the program invoked by the interruption, the instruction LOAD PSW EXTENDED may be used to restore the current PSW to the value of the old PSW.

Types of Instruction Ending

Instruction execution ends in one of five ways: completion, nullification, suppression, termination, and partial completion.

Partial completion of instruction execution occurs only for interruptible instructions; it is described in “Interruptible Instructions” on page 5-24.

Completion

Completion of instruction execution provides results as called for in the definition of the instruction. When an interruption occurs after the completion of the execution of an instruction, the instruction address in the old PSW designates the next sequential instruction.

Suppression

Except as noted below, suppression of instruction execution causes the instruction to be executed as if it specified “no operation.” The contents of any result fields, including the condition code, are not changed. The instruction address in the old PSW on an interruption after suppression designates the next sequential instruction.

When (a) the AFP-register (additional floating-point register) control bit, bit 45 of control register 0, is one, (b) either an IEEE-invalid-operation condition or an IEEE-division-by-zero condition is recognized, and (c) the respective IEEE mask bit in the floating-point-control (FPC) register is one, then the resulting data-exception program interruption is considered to be suppressing, even though the FPC is altered. See "Data-Exception Code (DXC)" on page 6-17 for details.

When (a) the additional floating-point register (AFP-register) control bit, bit 45 of control register 0, is one, (b) the vector enablement control, bit 46 of control register 0, is one, and (c) a vector-processing exception is recognized, then the resulting vector-processing exception program interruption is considered to be suppressing, even though the FPC is altered. See "Vector-Exception Code" on page 6-20 for details.

In the access-register mode, if an access exception defined to be suppressing is recognized for an operand of PERFORM LOCKED OPERATION whose address is in the parameter list, the ALET associated with the address is placed into the access register designated by the R_3 field of the instruction.

Nullification

Except as noted below, nullification of instruction execution has the same effect as suppression, except that when an interruption occurs after the execution of an instruction has been nullified, the instruction address in the old PSW designates the instruction whose execution was nullified (or an execute-type instruction, as appropriate) instead of the next sequential instruction.

In the access-register mode, if an access exception defined to be nullifying is recognized for an operand of PERFORM LOCKED OPERATION whose address is in the parameter list, the ALET associated with the address is placed into the access register designated by the R_3 field of the instruction.

Termination

Termination of instruction execution causes the contents of any fields due to be changed by the instruction to be unpredictable. The operation may replace all, part, or none of the contents of the designated result fields and may change the condition code if such change is called for by the instruction. Unless the interruption is caused by a machine-check condition, the validity of the instruction address in the PSW, the interruption code, and the ILC are not affected, and the state or the operation of the machine is not affected in any other way. The instruction address in the old PSW on an interruption after termination designates the next sequential instruction.

The terms completion, nullification, partial completion, suppression, and termination, and their corresponding effects, apply to the ending of individual instructions. When a transaction aborts, all transactional stores made by instructions while the CPU was in the transactional-execution mode are discarded, nontransactional stores are committed, and general registers designated by the general-register-save mask (GRSM) of the outermost TRANSACTION BEGIN instruction are restored to their contents prior to transactional execution. Any other general registers (not designated by the GRSM), access registers, vector registers, and floating-point registers (including the floating-point control register) that were altered while the CPU was in the transactional-execution mode retain their contents.

When a transaction is aborted, the program-status word is replaced with the transaction-abort PSW, except that the condition code is set to indicate the severity of the abort condition. In the case of a transaction that is aborted due to conditions that result in an interruption, this PSW becomes the old PSW stored as a result of the interruption. See "Transaction-Abort PSW (TAPSW)" on page 5-93 for further details.

If a transaction is aborted due to any cause except a program-interruption condition, the instruction address that is placed into the aborted-transaction-instruction-address field of the transaction diagnostic block is that of the instruction that would have been executed next in the absence of the abort. If a transaction is aborted due to a program-interruption condition, the instruction address that is placed into the aborted-transaction-instruction-address field is that of either the instruction at which the exception condition was detected or the instruction that would have been executed next in the absence of the abort, depending on whether the program-interruption condition was or was not nullifying, respectively. See "Aborted-Transaction Instruction Address (ATIA)" on page 5-95 for additional details.

Programming Note: Although the execution of an instruction is treated as a no-operation when suppression or nullification occurs, stores may be per-

formed as the result of the implicit tracing action associated with some instructions. See “Tracing” on page 4-12.

Interruptible Instructions

Point of Interruption

For most instructions, the entire execution of an instruction is one operation. An interruption is permitted between operations; that is, an interruption can occur after the performance of one operation and before the start of a subsequent operation.

For the following instructions, referred to as interruptible instructions, an interruption is permitted also after partial completion of the instruction:

- COMPARE AND FORM CODEWORD
- COMPARE LOGICAL LONG
- COMPARE UNTIL SUBSTRING EQUAL
- COMPRESSION CALL
- INVALIDATE PAGE TABLE ENTRY (when the IPTE-range facility is installed, and the R_3 field is nonzero)
- MOVE LONG
- PERFORM FRAME MANAGEMENT FUNCTION when the EDAT-1 facility is installed and the frame-size code designates a 1 M-byte frame, or when the EDAT-2 facility is installed and the frame-size code designates a 2 G-byte frame
- SET STORAGE KEY EXTENDED (when the enhanced-DAT facility is installed, and the multiple-block control is one)
- TEST BLOCK
- UPDATE TREE

Unit of Operation

Whenever points of interruption that include those occurring within the execution of an interruptible instruction are discussed, the term “unit of operation” is used. For a noninterruptible instruction, the entire execution consists, in effect, in the execution of one unit of operation.

The execution of an interruptible instruction is considered to consist in the execution of a number of units of operation, and an interruption is permitted between units of operation. The amount of data processed in a unit of operation depends on the particular instruction and may depend on the model and on the particular condition that causes the execution of the instruction to be interrupted.

When an instruction execution consists of a number of units of operation and an interruption occurs after some, but not all, units of operation have been completed, the instruction is said to be partially completed. In this case, the type of ending (completion, nullification, or suppression) is associated with the unit of operation. In the case of termination, the entire instruction is terminated, not just the unit of operation.

An exception may exist that causes the first unit of operation of an interruptible instruction not to be completed. In this case when the ending is nullification or suppression, all operand parameters and result locations remain unchanged, except that the condition code is unpredictable if the instruction is defined to set the condition code.

When a storage-alteration PER event is recognized, fewer than 4K additional bytes are stored before the event is indicated by an interruption. When a zero-address-detection PER event is recognized, the event is indicated by an interruption that may occur on completion of a unit of operation.

Execution of Interruptible Instructions

The execution of an interruptible instruction is completed when all units of operation associated with that instruction are completed. When an interruption occurs after completion, nullification, or suppression of a unit of operation, all preceding units of operation have been completed, and subsequent units of operation and instructions have not been started. The main difference between these types of ending is the handling of the current unit of operation and whether the instruction address stored in the old PSW identifies the current instruction or the next sequential instruction.

At the time of an interruption, changes to storage locations or register contents which are due to be made by instructions following the interrupted instruction have not yet been made.

Completion: On completion of the last unit of operation of an interruptible instruction, the instruction address in the old PSW designates the next sequential instruction. The result location for the current unit of operation has been updated. It depends on the particular instruction how the operand parameters are adjusted. On completion of a unit of operation other than the last one, the instruction address in the old PSW designates the interrupted instruction or an

execute-type instruction, as appropriate. The result location for the current unit of operation has been updated. The operand parameters are adjusted such that the execution of the interrupted instruction is resumed from the point of interruption when the old PSW stored during the interruption is made the current PSW.

Nullification: When a unit of operation is nullified, the instruction address in the old PSW designates the interrupted instruction or an execute-type instruction, as appropriate. The result location for the current unit of operation remains unchanged. The operand parameters are adjusted such that, if the instruction is reexecuted, execution of the interrupted instruction is resumed with the current unit of operation.

Suppression: When a unit of operation is suppressed, the instruction address in the old PSW designates the next sequential instruction. The operand parameters, however, are adjusted so as to indicate the extent to which instruction execution has been completed. If the instruction is reexecuted after the conditions causing the suppression have been removed, the execution is resumed with the current unit of operation.

Termination: When an exception which causes termination occurs as part of a unit of operation of an interruptible instruction, the entire operation is terminated, and the contents, in general, of any fields due to be changed by the instruction are unpredictable. On such an interruption, the instruction address in the old PSW designates the next sequential instruction.

The differences among the four types of ending for a unit of operation are summarized in Figure 5-5 on page 5-25.

If an instruction is defined to set the condition code, the execution of the instruction makes the condition code unpredictable except when the last unit of operation has been completed.

Condition-Code Alternative to Interruptibility

The following instructions are not interruptible instructions but instead may be completed after performing a CPU-determined subportion of the processing specified by the parameters of the instructions:

Unit of Operation Is	Instruction Address	Operand Parameters	Current Result Location
Completed			
Last unit of operation	Next instruction	Depends on the instruction	Changed
Any other unit of operation	Current instruction	Next unit of operation	Changed
Nullified	Current Instruction	Current unit of operation	Unchanged
Suppressed	Next Instruction	Current unit of operation	Unchanged
Terminated	Next instruction	Unpredictable	Unpredictable

Figure 5-5. Types of Ending for a Unit of Operation

- CHECKSUM
- CIPHER MESSAGE
- CIPHER MESSAGE WITH AUTHENTICATION
- CIPHER MESSAGE WITH CIPHER FEEDBACK
- CIPHER MESSAGE WITH CHAINING
- CIPHER MESSAGE WITH COUNTER
- CIPHER MESSAGE WITH OUTPUT FEEDBACK
- COMPARE LOGICAL LONG EXTENDED
- COMPARE LOGICAL LONG UNICODE
- COMPARE LOGICAL STRING
- COMPUTE INTERMEDIATE MESSAGE DIGEST
- COMPUTE LAST MESSAGE DIGEST
- COMPUTE MESSAGE AUTHENTICATION CODE
- CONVERT UTF-16 TO UTF-32
- CONVERT UTF-16 TO UTF-8
- CONVERT UTF-32 TO UTF-16
- CONVERT UTF-32 TO UTF-8
- CONVERT UTF-8 TO UTF-16
- CONVERT UTF-8 TO UTF-32
- MOVE LONG EXTENDED
- MOVE LONG UNICODE
- MOVE STRING
- PERFORM CRYPTOGRAPHIC COMPUTATION
- PERFORM RANDOM NUMBER OPERATION
- SEARCH STRING
- SEARCH STRING UNICODE
- TRANSLATE AND TEST EXTENDED
- TRANSLATE AND TEST REVERSE EXTENDED

- TRANSLATE EXTENDED
- TRANSLATE ONE TO ONE
- TRANSLATE ONE TO TWO
- TRANSLATE TWO TO ONE
- TRANSLATE TWO TO TWO

When any of the above instructions is completed after performing only a CPU-determined amount of processing instead of all specified processing, the instruction sets condition code 3. On such completion, the instruction address in the PSW designates the next sequential instruction, and the operand parameters of the instruction have been adjusted so that the processing of the instruction can be resumed simply by branching back to the instruction to execute it again. When the instruction has performed all specified processing, it sets a condition code other than 3.

The points at which any of the above instructions may set condition code 3 are comparable to the points of interruption of an interruptible instruction, and the amount of processing between adjacent points is comparable to a unit of operation of an interruptible instruction. However, since the instruction is not interruptible, each execution is considered the execution of one unit of operation.

Completion with the setting of condition code 3 permits interruptions to occur. Depending on the model and the instruction, condition code 3 may or may not be set when there is not a need for an interruption.

When a storage-alteration PER event is recognized, fewer than 4K additional bytes are stored before the instruction is completed (with condition code 3, if no other condition takes precedence), and the event is indicated by an interruption. When a zero-address-detection PER event is recognized, the instruction is completed (with condition code 3, if no other condition takes precedence), and the event is indicated by an interruption.

The COMPARE UNTIL SUBSTRING EQUAL and COMPRESSION CALL instructions both are interruptible instructions and ones that may set condition code 3 after performing a CPU-determined amount of processing.

Programming Notes:

1. Any interruption, other than supervisor call and some program interruptions, can occur after a partial execution of an interruptible instruction. In particular, interruptions for external, I/O, machine-check, restart, and program interruptions for access exceptions and PER events can occur between units of operation.

2. The amount of data processed in a unit of operation of an interruptible instruction depends on the model and may depend on the type of condition which causes the execution of the instruction to be interrupted or stopped. Thus, when an interruption occurs at the end of the current unit of operation, the length of the unit of operation may be different for different types of interruptions. Also, when the stop function is requested during the execution of an interruptible instruction, the CPU enters the stopped state at the completion of the execution of the current unit of operation. Similarly, in the instruction-step mode, only a single unit of operation is performed, but the unit of operation for the various cases of stopping may be different.

Exceptions to Nullification and Suppression

In certain unusual situations, the result fields of an instruction having a store-type operand are changed in spite of the occurrence of an exception which would normally result in nullification or suppression. These situations are exceptions to the general rule that the operation is treated as a no-operation when an exception requiring nullification or suppression is recognized. Each of these situations may result in the turning on of the change bit associated with the store-type operand, even though the final result in storage may appear unchanged. Depending on the particular situation, additional effects may be observable. The extent of these effects is described along with each of the situations.

All of these situations are limited to the extent that a store access does not occur and the change bit is not set when the store access is prohibited. For the CPU, a store access is prohibited whenever an access exception exists for that access, or whenever an exception exists which is of higher priority than the priority of an access exception for that access.

When, in these situations, an interruption for an exception requiring suppression occurs, the instruction address in the old PSW designates the next sequential instruction. When an interruption for an exception requiring nullification occurs, the instruc-

tion address in the old PSW designates the instruction causing the exception even though partial results may have been stored.

Programming Note: The term *translation-associated access exception* refers to any of the following exceptions: ALE-sequence, ALEN-translation, ALET-specification, ASCE-type, ASTE-sequence (during ART), ASTE-validity (during ART), addressing due to the contents of an ART- or DAT-table entry, extended-authority, page-translation, region-first-translation, region-second-translation, region-third-translation, segment-translation, translation-specification. In some models prior to z/Architecture, if a CPU made a store-type reference where part of the operand was in an accessible page, and the other part of the operand was in a page for which a translation-associated access exception existed, the I/O subsystem could observe the accessible portion of the operand changed to an intermediate value that was then restored to the original value. The visibility of an intermediate operand was limited to the I/O subsystem; other CPUs did not observe an intermediate value.

This behavior is not applicable to any machine that is capable of operating in the z/Architecture architectural mode.

Modification of DAT-Table Entries

When a valid and attached DAT-table entry is changed to a value which would cause an exception, and when, before the TLB is cleared of entries which qualify for substitution for that entry, an attempt is made to refer to storage by using a virtual address requiring that entry for translation, the contents of any fields due to be changed by the instruction are unpredictable. Results, if any, associated with the virtual address whose DAT-table entry was changed may be placed in those real locations originally associated with the address. Furthermore, it is unpredictable whether or not an interruption occurs for an access exception that was not initially applicable. On some machines, this situation may be reported by means of an instruction-processing-damage machine check with the delayed-access-exception bit also indicated.

Trial Execution for Editing Instructions and Translate Instruction

For the instructions EDIT, EDIT AND MARK, and TRANSLATE, the portions of the operands that are actually used in the operation may be established in a trial execution for operand accessibility that is performed before the execution of the instruction is started. This trial execution consists in an execution of the instruction in which results are not stored. If the first operand of TRANSLATE or either operand of EDIT or EDIT AND MARK is changed by another CPU or by a channel program, after the initial trial execution but before completion of execution, the contents of any fields due to be changed by the instruction are unpredictable.

Authorization Mechanisms

Most of the authorization mechanisms that are described in this section permit the control program to establish the degree of function provided to a particular semiprivileged program. These authorization mechanisms are intended for use by programs considered to be semiprivileged, that is, programs that are executed in the problem state but which may be authorized to use additional capabilities. With these authorization controls, a hierarchy of programs may be established, with programs at a higher level having a greater degree of privilege or authority than programs at a lower level. The range of functions available at each level, and the ability to transfer control from a lower to a higher level, are specified in tables which are managed by the control program. When the linkage stack is used, a nonhierarchical transfer of control also can be specified.

A semiprivileged instruction is one which can be executed in the problem state, but which is subject to the control of one or more of the authorization mechanisms described in this section. There are 25 semiprivileged instructions and also the privileged LOAD ADDRESS SPACE PARAMETERS instruction that are controlled by the authorization mechanisms. All of these semiprivileged and privileged instructions are described in Chapter 10, “Control Instructions”.

This section also describes, or refers to descriptions of, authorization mechanisms that are available when the ASN-and-LX-reuse facility is installed. These mechanisms do not apply to particular programs or provide for a hierarchy of programs.

The instructions controlled by the authorization mechanisms are listed in Figure 5-6 on page 5-31. The figure also shows additional authorization mechanisms that do not control specifically semiprivileged instructions; they control implicit access-register translation (access-register translation as part of an

instruction making a storage reference) and also access-register translation in the LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, STORE REAL ADDRESS, TEST ACCESS, and TEST PROTECTION instructions and a special form of access-register translation in the BRANCH IN SUBSPACE GROUP instruction. These additional mechanisms (the extended authorization index, ALE sequence number, and ASTE sequence number) index and ALE sequence number) are described in "Access-Register-Specified Address Spaces" on page 5-46, and the ASTE instance number is further described in "ASN-Second-Table-Entry Sequence Number" on page 3-24. The figure also shows the LSTE sequence number, which may be used in PC-number translation in the PROGRAM CALL instruction and is described in "ASN-and-LX-Reuse Control (R):" on page 5-34.

Mode Requirements

Most of the semiprivileged instructions can be executed only with DAT on. Basic PROGRAM CALL, PROGRAM TRANSFER, and PROGRAM TRANSFER WITH INSTANCE, are valid only in the primary-space mode. (Basic PROGRAM CALL is the PROGRAM CALL operation when the linkage stack is not used. When the linkage stack is used, the PROGRAM CALL operation is called stacking PROGRAM CALL). MOVE TO PRIMARY and MOVE TO SECONDARY are valid only in the primary-space and secondary-space modes. BRANCH AND STACK, stacking PROGRAM CALL, and PROGRAM RETURN are valid only in the primary-space and access-register modes. EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE are valid only in the primary-space, access-register, and home-space modes. When a semiprivileged instruction is executed in an invalid translation mode, a special-operation exception is recognized.

PROGRAM TRANSFER and PROGRAM TRANSFER WITH INSTANCE specify a new value for the problem-state bit in the PSW. If a program in the problem state attempts to execute PROGRAM TRANSFER or PROGRAM TRANSFER WITH INSTANCE and set the supervisor state, a privileged-operation exception is recognized. A privileged-operation exception is also recognized on an attempt to use RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST to set the home-space mode in the problem state.

Extraction-Authority Control

The extraction-authority-control bit is located in bit position 36 of control register 0. In the problem state, bit 36 must be one to allow completion of these instructions:

- EXTRACT PRIMARY ASN
- EXTRACT PRIMARY ASN AND INSTANCE
- EXTRACT SECONDARY ASN
- EXTRACT SECONDARY ASN AND INSTANCE
- INSERT ADDRESS SPACE CONTROL
- INSERT PSW KEY
- INSERT VIRTUAL STORAGE KEY

Otherwise, a privileged-operation exception is recognized. The extraction-authority control is not examined in the supervisor state.

PSW-Key Mask

The PSW-key mask consists of bits 32-47 in control register 3, with the bits corresponding to the values 0-15, respectively, of the PSW key. These bits are used in the problem state to control which keys and entry points are authorized for the program. The PSW-key mask is modified by PROGRAM TRANSFER, and PROGRAM TRANSFER WITH INSTANCE, is modified or loaded by BRANCH AND SET AUTHORITY and PROGRAM CALL, and is loaded by LOAD ADDRESS SPACE PARAMETERS and PROGRAM RETURN. The PSW-key mask is used in the problem state to control the following:

- The PSW-key values that can be set by means of the instruction SET PSW KEY FROM ADDRESS.
- The PSW-key values that are valid for the six move instructions that specify a second access key: MOVE PAGE, MOVE TO PRIMARY, MOVE TO SECONDARY, MOVE WITH DESTINATION KEY, MOVE WITH KEY, and MOVE WITH SOURCE KEY.
- The PSW-key values that are valid for the MOVE WITH OPTIONAL SPECIFICATIONS instruction when a key is explicitly specified for the first, second, or both operands.
- The entry points which can be called by means of PROGRAM CALL. In this case, the PSW-key mask is ANDed with the authorization key mask in the entry-table entry, and, if the result is zero, the program is not authorized.

When an instruction in the problem state attempts to use a key not authorized by the PSW-key mask, a privileged-operation exception is recognized. The same action is taken when an instruction in the problem state attempts to call an entry not authorized by the PSW-key mask. The PSW-key mask is not examined in the supervisor state, all keys and entry points being valid.

Secondary-Space Control

Bit 37 of control register 0 is the secondary-space-control bit. This bit provides a mechanism whereby the control program can indicate whether or not the secondary region-first table, region-second table, region-third table or segment table has been established. Bit 37 may be required to be one to allow completion of SET ADDRESS SPACE CONTROL FAST and MOVE WITH OPTIONAL SPECIFICATIONS, and it must be one to allow completion of these instructions:

- MOVE TO PRIMARY
- MOVE TO SECONDARY
- MOVE WITH OPTIONAL SPECIFICATIONS, when either operand-access control designates the secondary space
- SET ADDRESS SPACE CONTROL

Otherwise, a special-operation exception is recognized. The secondary-space control is examined in both the problem and supervisor states.

Subsystem-Linkage Control

Bit 192 of the primary ASN-second-table entry is the subsystem-linkage-control bit. The subsystem-linkage control must be one to allow completion of these instructions:

- PROGRAM CALL
- PROGRAM TRANSFER
- PROGRAM TRANSFER WITH INSTANCE

Otherwise, a special-operation exception is recognized. The subsystem-linkage control is examined in both the problem and supervisor states and controls both the space-switching and current-primary versions of the instructions.

ASN-Translation Control

Bit 44 of control register 14 is the ASN-translation-control bit. This bit provides a mechanism whereby the control program can indicate whether ASN translation may occur while a particular program is being executed. Bit 44 must be one to allow completion of these instructions:

- LOAD ADDRESS SPACE PARAMETERS
- SET SECONDARY ASN
- SET SECONDARY ASN WITH INSTANCE
- PROGRAM CALL with space switching
- PROGRAM RETURN with space switching and also when the restored secondary ASN is not equal to the restored primary ASN
- PROGRAM TRANSFER with space switching
- PROGRAM TRANSFER WITH INSTANCE with space switching

Otherwise, a special-operation exception is recognized. The ASN-translation control is examined in both the problem and supervisor states. The ASN-translation control is examined by PROGRAM CALL even though PROGRAM CALL obtains the address of the ASN-second-table entry directly from the entry-table entry instead of by performing ASN translation.

Authorization Index

The authorization index is contained in bit positions 32-47 of control register 4. The authorization index is associated with the primary address space and is loaded along with the PASN when PROGRAM CALL with space switching, PROGRAM RETURN with space switching, PROGRAM TRANSFER with space switching, PROGRAM TRANSFER WITH INSTANCE with space switching, or LOAD ADDRESS SPACE PARAMETERS is executed. The authorization index is used to determine whether a program is authorized to establish a particular address space. A program may be authorized to establish the address space as a secondary-address space, as a primary-address space, or both. The authorization index is examined in both the problem and supervisor states.

Associated with each address space is an authority table. The authorization index is used to select an entry in the authority table. Each entry contains two bits, which indicate whether the program with that authorization index is permitted to establish the address space as a primary address space, as a secondary address space, or both.

The instructions SET SECONDARY ASN with space switching, and SET SECONDARY ASN WITH INSTANCE with space switching, and the instruction PROGRAM RETURN when the restored secondary ASN is not equal to the restored primary ASN, use

the authorization index to test the secondary-authority bit in the authority-table entry to determine if the address space can be established as a secondary address space. The tested bit must be one; otherwise, a secondary-authority exception is recognized.

The instructions PROGRAM TRANSFER with space switching and PROGRAM TRANSFER WITH INSTANCE with space switching use the authorization index to test the primary-authority bit in the authority-table entry to determine if the address space can be established as a primary address space. The tested bit must be one; otherwise, a primary-authority exception is recognized.

The instruction PROGRAM CALL with space switching causes a new authorization index to be loaded from the ASN-second-table entry. This permits the program which is called to be given an authorization index which authorizes it to access more or different address spaces than those authorized for the calling program. The instructions PROGRAM RETURN with space switching, PROGRAM TRANSFER with space switching, and PROGRAM TRANSFER WITH INSTANCE with space switching restore the authorization index that is associated with the returned-to address space.

The secondary-authority bit in the authority-table entry may also be used, along with the extended authorization index, to determine if the program is authorized to use an access-list entry in access-register translation. This is described in “Access-Register-Specified Address Spaces” on page 5-46.

Instructions and Controls Related to ASN-and-LX Reuse

This section describes instructions and controls that are provided when the ASN-and-LX-reuse facility is installed. The controls apply in both the problem and the supervisor states.

Instructions: The ASN-and-LX-reuse facility provides the following instructions:

- EXTRACT PRIMARY ASN AND INSTANCE
- EXTRACT SECONDARY ASN AND INSTANCE
- PROGRAM TRANSFER WITH INSTANCE
- SET SECONDARY ASN WITH INSTANCE

PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE are similar to PROGRAM TRANSFER and SET SECONDARY ASN, respectively. The space-switching forms of PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE compare an ASTE instance number (ASTEIN) specified by the program in general register 1 to the ASTEIN in the ASN-second-table entry for the address space to which the space switch is to occur. An ASTE-instance exception is recognized if the two ASTEINs are not equal.

ASN-and-LX-Reuse Control: Bit 44 of control register 0 is the ASN-and-LX-reuse control. This bit is used to control both ASN reuse and LX reuse, as described below.

ASN Reuse: When the ASN-and-LX-reuse control is one, PROGRAM RETURN and LOAD ADDRESS SPACE PARAMETERS, when they perform ASN translation of either a primary ASN or a secondary ASN, compare a specified ASTEIN to the ASTEIN in the ASN-second-table entry located by the ASN translation. The two ASTEINs must be equal; otherwise, PROGRAM RETURN recognizes an ASTE-instance exception, and LOAD ADDRESS SPACE PARAMETERS sets condition code 1 or 2. For PROGRAM RETURN, the specified ASTEIN is in the linkage-stack program-call state entry being unstacked. For LOAD ADDRESS SPACE PARAMETERS, the specified ASTEIN is in the first operand of the instruction.

Bit 31 of word 1 of the ASN-second-table entry is the reusable-ASN bit. When the ASN-and-LX-reuse control is one, the space-switching forms of PROGRAM TRANSFER and SET SECONDARY ASN recognize a special-operation exception if the reusable-ASN bit is one in the ASN-second-table entry located by ASN translation. These instructions do not make a comparison to the ASTEIN in the ASN-second-table entry.

Bit 30 of word 1 of the ASN-second-table entry is the controlled-ASN bit. Regardless of the value of the ASN-and-LX-reuse control, PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE recognize a special-operation exception if the controlled-ASN bit is one in the ASN-second-table entry located by ASN translation and the CPU is in the problem state at the beginning of the operation.

A more detailed description of ASN reuse is given in “ASN-Second-Table-Entry Instance Number and ASN Reuse” on page 3-25.

LX Reuse: When the ASN-and-LX-reuse control is one, the linkage table is replaced by a linkage first table and, for each valid linkage-first-table entry, a linkage second table.

The second word of the linkage-second-table entry used contains an LSTE sequence number (LSTESN), and PROGRAM CALL recognizes an LSTE-sequence exception if this LSTESN is nonzero and not equal to an LSTESN specified by the program in general register 15.

A more detailed description of LX reuse is given in "ASN-and-LX-Reuse Control (R):" on page 5-34.

Function or Instruction	Mode Requirement		Authorization Mechanism											Space Switch Event Ctl. (1.57, 13.57)
	Pr. Op	Trans. Mode	Subs. Link. Ctl. [6]	Sec.-Space Ctl. (0.37)	ASN-Trans. Ctl. (14.44)	Extr. Auth. Ctl (0.36)	PSW-Key Mask (3.32-3.47)	Auth. Index (4.32-4.47)	Ext.-Auth. Index (8.32-8.47)	ALE Seq. No. [7]	ASTE Seq. No. [8]	ASTE Inst. No. [9]	LSTE Seq. No. [10]	
Implicit AR translation		A							EA	ALQ	ASQ			
BAKR		SO-PA												
BSA-ba							Q							
BSA-ra														
BSG		SO-PSAH									ASQ			
EPAR		SO-PSAH				Q								
EPAIR		SO-PSAH				Q								
EREG		SO-PAH												
EREGG		SO-PAH												
ESAR		SO-PSAH				Q								
ESAIR		SO-PSAH				Q								
ESTA		SO-PAH												
IAC		SO-PSAH				Q								
IPK						Q								
IVSK		SO-PSAH				Q								
LASP	P				SO			CC			CC	CC		CC
LPTEA	P								CCA	CCA	CCA			
LRA	P								CCA	CCA	CCA			
LRAG	P								CCA	CCA	CCA			
MSTA		SO-PAH												
MVCDK							Q							
MVCOS		SO-PSAH		SO-MS			Q							
MVCP		SO-PS		SO			Q							
MVCS		SO-PS		SO			Q							
MVCSK							Q							
bPC-cp		SO-P	SO				Q[1]						LSQ	
sPC-cp		SO-PA	SO				Q[1]						LSQ	
bPC-ss		SO-P	SO		SO		Q[1]				ASQ		LSQ	X1
sPC-ss		SO-PA	SO		SO		Q[1]				ASQ		LSQ	X1
PR-cp		SO-PA			SO[4]			SA[5]			ASQ	AIN		
PR-ss		SO-PA			SO			PASA[5]			ASQ	AIN		X1
PT-cp	Q[2]	SO-P	SO											
PT-ss[11]	Q[2]	SO-P	SO		SO			PA			ASQ			X1
PTI-cp	Q[2]	SO-P	SO											
PTI-ss[12]	Q[2]	SO-P	SO		SO			PA			ASQ	AIN		X1

Figure 5-6. Summary of Authorization Mechanisms (Part 1 of 2)

Function or Instruction	Mode Requirement		Authorization Mechanism											Space Switch Event Ctl. (1.57, 13.57)
	Pr. Op	Trans. Mode	Subs. Link. Ctl. [6]	Sec.-Space Ctl. (0.37)	ASN-Trans. Ctl. (14.44)	Extr. Auth. Ctl (0.36)	PSW-Key Mask (3.32-3.47)	Auth. Index (4.32-4.47)	Ext.-Auth. Index (8.32-8.47)	ALE Seq. No. [7]	ASTE Seq. No. [8]	ASTE Inst. No. [9]	LSTE Seq. No. [10]	
RP	Q[3]													X2
SAC	Q[3]	SO-PSAH		SO										X2
SACF	Q[3]	SO-PSAH		SO[13]										X2
SPKA							Q							
SSAR-cp		SO-PSAH			SO									
SSAR-ss[11]		SO-PSAH			SO			SA			ASQ			
SSAIR-cp		SO-PSAH			SO									
SSAIR-ss[12]		SO-PSAH			SO			SA			ASQ	AIN		
STRAG	P													
TAR									CC	CC	CC			
TPROT	P								CC	CC	CC			

Figure 5-6. Summary of Authorization Mechanisms (Part 2 of 2)

Explanation for Summary of Authorization Mechanisms:

1 The PSW-key mask is ANDed with the authorization key mask in the entry-table entry.

2 The exception is recognized on an attempt to set the supervisor state when in the problem state.

3 The exception is recognized on an attempt to set the home-space mode when in the problem state.

4 ASN translation is performed for the new SASN, and the exception may be recognized, only when the new SASN is not equal to the new PASN.

5 Secondary authority is checked for the new SASN, and the exception may be recognized, only when the new SASN is not equal to the new PASN.

6 Subsystem-linkage control is bit 92 of the primary ASN-second-table entry.

7 ALE sequence number is bits 8-15 of the access-list entry

8 ASTE sequence number is bits 160-191 of the ASN-second-table entry.

9 ASTE instance number is bits 352-383 of the ASN-second-table entry

10 LSTE sequence number is bits 32-63 of the linkage-second-table entry.

11 Special-operation exception is recognized if ASN-and-LX-reuse control, bit 44 of control register 0, is one and reusable-ASN bit, bit 31 of word 1 of the ASN-second-table entry, is one.

12 Special-operation exception is recognized if controlled-ASN bit, bit 30 of word 1 of the ASN-second-table entry, is one and the CPU is in the problem state at the beginning of the operation.

13 Whether the exception is recognized is unpredictable.

A Access-register translation occurs only in the access-register mode.

AIN ASTE-instance exception.

ALQ ALE-sequence exception.

ASQ ASTE-sequence exception

bPC Basic (nonstacking) PROGRAM CALL.

CC Test results in setting a condition code.

CCA Test results in setting a condition code. The test occurs only in the access-register mode.

CRx.y Control register x, bit position y.

EA Extended-authority exception.

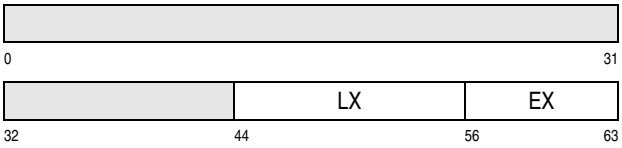
LSQ	LSTE-sequence exception.
P	Privileged-operation exception for privileged instruction.
PA	Primary-authority exception.
PASA	Primary-authority exception or secondary-authority exception.
Q	Privileged-operation exception for semi-privileged instruction. Authority checked only in the problem state.
SA	Secondary-authority exception.
SO	Special-operation exception.
SO-MS	The secondary-space control, bit 37 of control register 0, must be one when either operand of MOVE WITH OPTIONAL SPECIFICATIONS is accessed in the secondary-space mode; otherwise, a special-operation exception is recognized.
SO-P	CPU must be in the primary-space mode; special-operation exception if the CPU is in the secondary-space, access-register, home-space, or real mode.
SO-PA	CPU must be in the primary-space or access-register mode; special-operation exception if the CPU is in the secondary-space, home-space, or real mode.
SO-PAH	CPU must be in the primary-space, access-register, or home-space mode; special-operation exception if the CPU is in the secondary-space or real mode.
SO-PS	CPU must be in the primary-space or secondary-space mode; special-operation exception if the CPU is in the home-space, access-register, or real mode.
SO-PSAH	CPU must be in the primary-space, secondary-space, access-register, or home-space mode; special-operation exception if the CPU is in the real mode.
sPC	Stacking PROGRAM CALL.
X1	When bit 57 of control register 1 is one, a space-switch event is recognized. The operation is completed.
X2	When bit 57 of control register 1 or 13 is one and the instruction space is changed to or from the home address space, a space-switch event is recognized. The operation is completed.

PC-Number Translation

PC-number translation is the process of translating PC number to locate an entry-table entry as part of the execution of the PROGRAM CALL instruction. When the ASN-and-LX-reuse facility is not installed or is not enabled, the PC number is 20 bits in bit positions 44-63 of the effective address used by PROGRAM CALL. When the ASN-and-LX-reuse facility is installed and enabled, the PC number is 20 bits in bit positions 44-63 of the effective address if bit 44 of the address is zero, or it is 32 bits in bit positions 32-63 of the address if bit 44 of the address is one. The facility is enabled if the ASN-and-LX-reuse control, bit 44 of control register 0, is one. The case when the facility is installed and enabled is referred to simply by saying that ASN-and-LX reuse is enabled.

To perform the translation of a PC number to locate an entry-table entry, the PC number is divided into two fields: a linkage index (LX) and an entry index (EX). The entry index is always the rightmost eight bits of the PC number. When ASN-and-LX reuse is not enabled, the leftmost 12 bits of the 20-bit PC number are the linkage index. In this case, the effective address has the following format:

Effective Address when ASN-and-LX Reuse Is Not Enabled

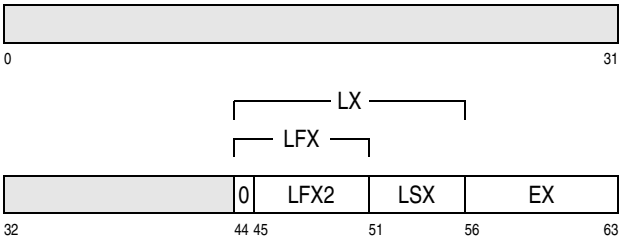


Bit 44 of the effective address has no special meaning and may be zero or one.

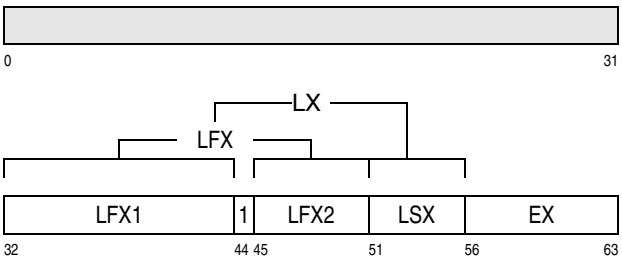
When ASN-and-LX reuse is enabled, the linkage index is further divided into a linkage first index (LFX) and a linkage second index (LSX). The linkage second index is always the five bits immediately on the left of the entry index. The size and format of the linkage first index depend on whether the PC number is 20 bits or 32 bits, which in turn depends on whether bit 44 of the effective address is zero or one, respec-

tively. In these cases, the effective address has the following formats:

Effective Address when ASN-and-LX Reuse Is Enabled and Bit 44 Is Zero



Effective Address when ASN-and-LX Reuse Is Enabled and Bit 44 Is One



When ASN-and-LX reuse is enabled and bit 44 of the effective address is zero, the linkage first index is bits 44-50, or bits 45-50 (LFX2) with a zero appended on the left. Thus, the linkage first index is seven bits of which the leftmost bit is always zero. When bit 44 is one, the linkage first index is bits 45-50 (LFX2) with bits 32-43 (LFX1) appended on the left, or 18 bits. However, a linkage first table can contain at most 16,384 entries, and, therefore, the leftmost four bits of the linkage first index, bits 32-35 of the effective address, must always be zeros; otherwise, an LFX-translation exception is recognized.

When ASN-and-LX reuse is not enabled, the translation of a PC number is performed by means of two tables: a linkage table and an entry table. When ASN-and-LX reuse is enabled, the translation is performed by means of three tables: a linkage first table, a linkage second table, and an entry table. All of these tables reside in real storage. The linkage-table designation or linkage-first-table designation resides in another area in storage, called the primary ASN-second-table entry (primary ASTE), whose origin is in control register 5. When there are two levels of tables, The entry table is designated by means of a linkage-table entry. When there are three levels of tables, the linkage second table is designated by means of a linkage-first-table entry, and the entry table is designated by means of a linkage-second-table entry.

Programming Notes:

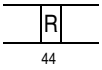
1. Bit 44 of a PROGRAM CALL effective address specifying a 32-bit PC number is not a numeric part of the PC number.

2. The effective address from which a PC number is derived is subject to the addressing mode in the current PSW. Therefore, bits 0-39 of the effective address in the 24-bit addressing mode, and bits 0-32 in the 31-bit addressing mode, are treated as containing zeros.

PC-Number Translation Control

PC-number translation is controlled by means of the ASN-and-LX-reuse control in control register 0 and a linkage-table designation or linkage-first-table designation in the primary ASN-second-table entry designated by the contents of control register 5.

Control Register 0



ASN-and-LX-Reuse Control (R): Bit 44 of control register 0 is the ASN-and-LX-reuse-control bit and is assigned if the ASN-and-LX-reuse facility is installed. When the bit is one, (1) the PC number is 20 bits if bit 44 of the effective address is zero or 32 bits if bit 44 is one, (2) the PC number is divided into a seven-bit (bit 44 is zero) or 18 bit (bit 44 is one) linkage first index, five-bit linkage second index, and eight-bit entry index, (3) the primary ASN-second-table entry contains a linkage-first-table designation instead of a linkage-table designation, and (4) the PC number is translated by means of a linkage first table, linkage second table, and entry table.

Also when the ASN-and-LX-reuse-control bit is one, if the linkage-second-table entry used during PC-number translation performed by PROGRAM CALL contains a nonzero linkage-second-table-entry sequence number (LSTESN), this LSTESN must be equal to an LSTESN specified in bit positions 0-31 of general register 15; otherwise, the PROGRAM CALL

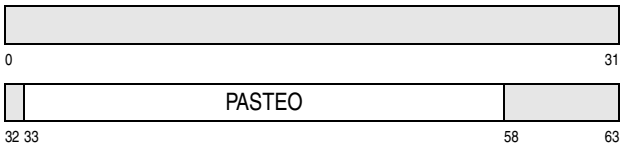
instruction cannot be completed, and an LSTE-sequence exception is recognized.

The ASN-and-LX-reuse control in control register 0 is examined in both the problem and the supervisor states. Other uses of this control, related to the ASN-second-table-entry instance number, are summarized in “ASN-Second-Table-Entry Instance Number and ASN Reuse” on page 3-25.

This use of the LSTESN allows a linkage index associated with a particular LSTESN to be made unusable when the linkage index is reassigned to specify a different (different origin) or conceptually different (different contents, or containing ASNs that designate conceptually different address spaces) entry table. The LX-and-LSTESN combination can be made unusable by changing the LSTESN in the linkage-second-table entry.

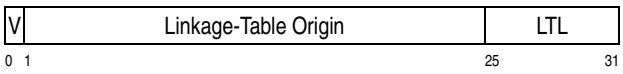
Control Register 5

Control register 5 specifies the location of the primary ASN-second-table entry. The register has the following format:



Primary-ASTE Origin (PASTEO): Bits 33-57 of control register 5, with six zeros appended on the right, form a 31-bit real address that designates the beginning of the primary ASTE. Bytes 24-27 of the primary ASTE contain the linkage-table designation if the ASN-and-LX-reuse facility is not installed or is not enabled by a one value of the ASN-and-LX-reuse control in control register 0, or they contain the linkage-first-table designation if the facility is installed and enabled.

The linkage-table designation has the following format:



Subsystem-Linkage Control (V): Bit 0 of the linkage-table designation is the subsystem-linkage-control bit. Bit 0 must be one to allow completion of these instructions:

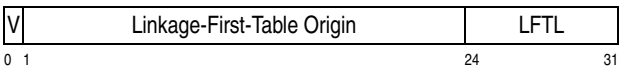
- PROGRAM CALL
- PROGRAM TRANSFER
- PROGRAM TRANSFER WITH INSTANCE

Otherwise, a special-operation exception is recognized. The subsystem-linkage control is examined in both the problem and the supervisor states and controls both the space-switching and current-primary versions of the instructions.

Linkage-Table Origin: Bits 1-24 of the linkage-table designation, with seven zeros appended on the right, form a 31-bit real address that designates the beginning of the linkage table.

Linkage-Table Length (LTL): Bits 25-31 of the linkage-table designation specify the length of the linkage table in units of 128 bytes, thus making the length of the linkage table variable in multiples of 32 four-byte entries. The length of the linkage table, in units of 128 bytes, is one more than the value in bit positions 25-31. The linkage-table length is compared against the leftmost seven bits of the linkage-index portion of the PC number to determine whether the linkage index designates an entry within the linkage table.

The linkage-first-table designation has the following format:



Subsystem-Linkage Control (V): Bit 0 of the linkage-first-table designation has the same definition as bit 0 of the linkage-table designation.

Linkage-First-Table Origin: Bits 1-23 of the linkage-first-table designation, with eight zeros appended on the right, form a 31-bit real address that designates the beginning of the linkage first table.

Linkage-First-Table Length (LFTL): Bits 24-31 of the linkage-first-table designation specify the length of the linkage first table in units of 256 bytes, thus making the length of the linkage first table variable in multiples of 64 four-byte entries. The length of the linkage first table, in units of 256 bytes, is one more than the value in bit positions 24-31. When bit 44 of the effective address specifying the PC number is one, the linkage-first-table length is compared against the leftmost 12 bits of the linkage-first-index portion of the PC number, bits 32-43 of the effective address, to determine whether the linkage first index

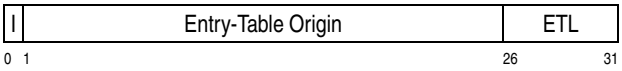
designates an entry within the linkage first table. When bit 44 of the effective address is zero, the linkage-first-table length is ignored.

PC-Number Translation Tables

If the ASN-and-LX-reuse facility is not installed or is not enabled by a one value of the ASN-and-LX-reuse control in control register 0, the PC-number translation process consists in a two-level lookup using two tables: a linkage table and an entry table. If the facility is installed and enabled, the PC-number translation process consists in a three-level lookup using three tables: a linkage first table, a linkage second table, and an entry table. All of these tables reside in real storage.

Linkage-Table Entries

The entry fetched from the linkage table has the following format:



The fields in the linkage-table entry are allocated as follows:

LX-Invalid Bit (I): Bit 0 controls whether the entry table associated with the linkage-table entry is available.

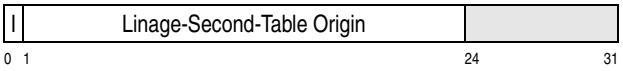
When the bit is zero, PC-number translation proceeds by using the linkage-table entry. When the bit is one, an LX-translation exception is recognized.

Entry-Table Origin: Bits 1-25, with six zeros appended on the right, form a 31-bit real address that designates the beginning of the entry table.

Entry-Table Length (ETL): Bits 26-31 specify the length of the entry table in units of 128 bytes, thus making the table variable in multiples of four 32-byte entries. The length of the entry table, in units of 128 bytes, is one more than the value in bit positions 26-31. The entry-table length is compared against the leftmost six bits of the entry index to determine whether the entry index designates an entry within the entry table.

Linkage-First-Table Entries

The entry fetched from the linkage first table has the following format:



The fields in the linkage-first-table entry are allocated as follows:

LFX-Invalid Bit (I): Bit 0 controls whether the linkage second table associated with the linkage-first-table entry is available.

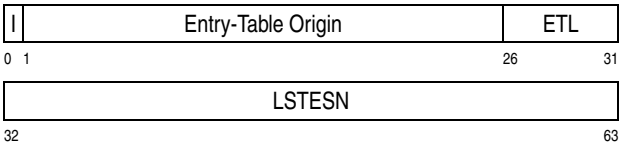
When the bit is zero, PC-number translation proceeds by using the linkage-first-table entry. When the bit is one, an LFX-translation exception is recognized.

Linkage-Second-Table Origin: Bits 1-23, with eight zeros appended on the right, form a 31-bit real address that designates the beginning of the linkage second table.

Programming Note: The unused field in the linkage-first-table entry, bits 24-31, should be set to zeros. This field is reserved for future extensions, and programs which place nonzero values in this field may not operate compatibly on future machines.

Linkage-Second-Table Entries

The entry fetched from the linkage second table has the following format:



The fields in the linkage-second-table entry are allocated as follows:

LSX-Invalid Bit (I): Bit 0 controls whether the entry table associated with the linkage-second-table entry is available.

When the bit is zero, PC-number translation proceeds by using the linkage-second-table entry. When the bit is one, an LSX-translation exception is recognized.

Entry-Table Origin: Bits 1-25, with six zeros appended on the right, form a 31-bit real address that designates the beginning of the entry table.

Entry-Table Length (ETL): Bits 26-31 specify the length of the entry table in units of 128 bytes, thus making the table variable in multiples of four 32-byte entries. The length of the entry table, in units of 128 bytes, is one more than the value in bit positions 26-31. The entry-table length is compared against the leftmost six bits of the entry index to determine whether the entry index designates an entry within the entry table.

Linkage-Second-Table-Entry Sequence Number (LSTESN): During PC-number translation in PROGRAM CALL when bits 32-63 are not all zeros, bits 32-63 are compared to an LSTESN specified in bit positions 0-31 of general register 15. The comparison must give an equal result; otherwise, an LSTE-sequence exception is recognized. When bits 32-63 are all zeros, bits 0-31 of general register 15 are ignored.

Programming Note: The format of bits 0-31 of the linkage-second-table entry is the same as the format of the linkage-table entry.

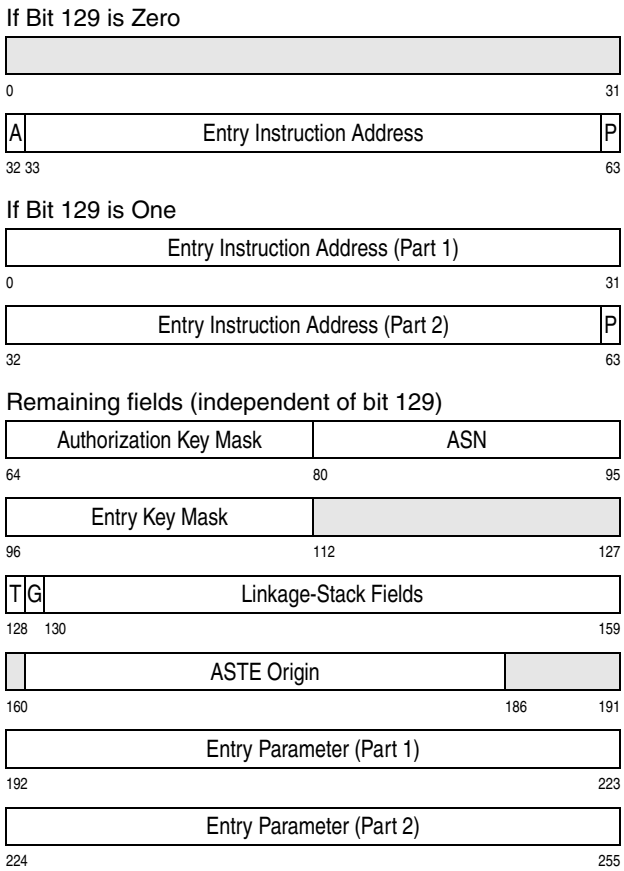
Entry-Table Entries

The format of bits 0-63 of the entry-table entry depends on whether the addressing-mode in effect after the PROGRAM CALL operation is the extended (64-bit) addressing mode or a basic (24-bit or 31-bit) addressing mode. This in turn depends on bits 128 and 129 of the entry-table entry.

Bit 128 of the entry-table entry (T) is the PC-type bit. When bit 128 is zero, PROGRAM CALL is to perform the basic (nonstacking) operation. When bit 128 is one, PROGRAM CALL is to perform the stacking operation.

Bit 129 of the entry-table entry (G) is the entry-extended-addressing-mode bit. In the basic PROGRAM CALL operation, bit 31 of the current PSW, the extended-addressing-mode bit, must equal bit 129; otherwise, a special-operation exception is recognized. In the stacking operation when bit 129 is zero, bit 31 of the current PSW is set to zero, and bit 32 of the PSW, the basic-addressing-mode bit, is set with the value of bit 32 of the entry-table entry (A), the entry-basic-addressing-mode bit. In the stacking operation when bit 129 is one, bits 31 and 32 of the current PSW both are set to one. Thus, the basic PROGRAM CALL operation does not switch between the extended and a basic addressing mode but can switch between the 24-bit and 31-bit modes, and the stacking operation can set any addressing mode.

The 32-byte entry-table entry has the following format:



The fields in the entry-table entry are allocated as follows:

Entry Basic Addressing Mode (A): When bit 129 is zero, bit 32 replaces the basic-addressing-mode bit, bit 32 of the current PSW, as part of the PROGRAM CALL operation. In this case if bit 32 is zero, bits 33-39 must also be zeros; otherwise, a PC-translation-specification exception is recognized. When bit 129 is one, bit 32 is a bit of the entry instruction address, and bit 32 of the PSW remains or is set to one.

Entry Instruction Address: When bit 129 is zero, bits 33-62, with 33 zeros appended on the left and a zero appended on the right, form the instruction

address which replaces the instruction address in the PSW as part of the PROGRAM CALL operation. When bit 129 is one, bits 0-62, with a zero appended on the right, form the instruction address.

Entry Problem State (P): Bit 63 replaces the problem-state bit, bit 15 of the current PSW, as part of the PROGRAM CALL operation.

Authorization Key Mask: Bits 64-79 are used to verify whether the program issuing the PROGRAM CALL instruction, when in the problem state, is authorized to call this entry point. The authorization key mask and the current PSW-key mask in control register 3 are ANDed, and the result is checked for all zeros. If the result is all zeros, a privileged-operation exception is recognized. The test is not performed in the supervisor state.

ASN: Bits 80-95 specify whether a space-switching (PC-ss) operation or a to-current-primary (PC-cp) operation is to occur. When bits 80-95 are zeros, PC-cp is specified. When bits 80-95 are not all zeros, PC-ss is specified, and the bits are the ASN that replaces the primary ASN.

Entry Key Mask: Bits 96-111 may be ORed into or may replace the PSW-key mask in control register 3 as part of the PROGRAM CALL operation, as determined by a bit in bit positions 130-159.

PC-Type Bit (T): Bit 128 specifies the basic PROGRAM CALL operation when the bit is zero or the stacking PROGRAM CALL operation when the bit is one.

Entry Extended Addressing Mode (G): In the basic PROGRAM CALL operation, bit 129 must match the extended-addressing-mode bit, bit 31 of the current PSW; otherwise, a special-operation exception is recognized. In the stacking operation, bit 129 replaces bit 31 of the PSW.

ASTE Origin: When bits 80-95 are not all zeros, bits 161-185, with six zeros appended on the right, form the 31-bit real ASN-second-table-entry address that should result from applying the ASN-translation process to bits 80-95.

Entry Parameter: When bit 129 is zero, bits 224-255 are placed in bit positions 32-63 of general register 4, and bits 0-31 of the register remain unchanged, as part of the PC operation. When bit 129 is one, bits 192-255 are placed in general register 4 as part of the PC operation.

Bits 130-159 are used in connection with the linkage stack and are described in “Linkage-Stack Entry-Table Entries” on page 5-75.

Bits 112-127, 160, and 186-191 are reserved for possible future extensions and should be zeros.

Programming Note: The entry parameter is intended to provide the called program with an address which can be depended upon and used as the basis of addressability in locating necessary information which may be environment dependent. The parameter may be appropriately changed for each environment by setting up different entry tables. The alternative — obtaining this information from the calling program — may require extensive validity checking or may present an integrity exposure.

Table Summary

Figure 5-7 on page 5-39 presents a summary of information about the tables used in PC-number translation, except for the primary ASN second table.

PC-Number-Translation Process

When the ASN-and-LX-reuse facility is not installed or is not enabled, the translation of the PC number is performed by means of a linkage table and entry table. When the ASN-and-LX-reuse facility is installed and enabled, referred to by saying that ASN-and-LX reuse is enabled, the translation of the PC number is performed by means of a linkage first table, linkage second table and entry table. All of these tables reside in real storage. The translation also requires the use of the primary ASN-second-table entry, which also resides in real storage.

For the purposes of PC-number translation, the PC number is divided into two parts: the leftmost part is called the linkage index (LX), and the rightmost eight bits are called the entry index (EX). When ASN-and-LX reuse is not enabled, the PC number is 20 bits, and the linkage index is 12 bits. When ASN-and-LX reuse is enabled, the linkage index also is divided into two parts: the leftmost part is called the linkage first index (LFX), and the rightmost five bits are called the linkage second index (LSX). When ASN-and LX reuse is enabled and bit 44 of the effective address specifying the PC number is zero, the PC number is 20 bits, and the linkage first index is seven bits, or,

when bit 44 is one, the PC number is 32 bits, and the linkage first index is 18 bits.

Table	Boundary (Bytes)	Entry Size (Bytes)	Unit Size		Maximum Number of Units	Maximum Size	
			Bytes	Entries		Bytes	Entries
Linkage Table	12	4	128	32	128	16384	4096
Linkage First Table	256	4	256	64	256[1]	65536	16384
Linkage Second Table	256	8	256	32	1	256	32
Entry Table	64	32	128	4	64	8192	256

Explanation:

[1] A 20-bit PC number can specify an entry only within the first unit. Bits 0-3 of a 32-bit PC number specify entries beyond the end of the largest possible table.

Figure 5-7. Summary of PC-Number-Translation Tables

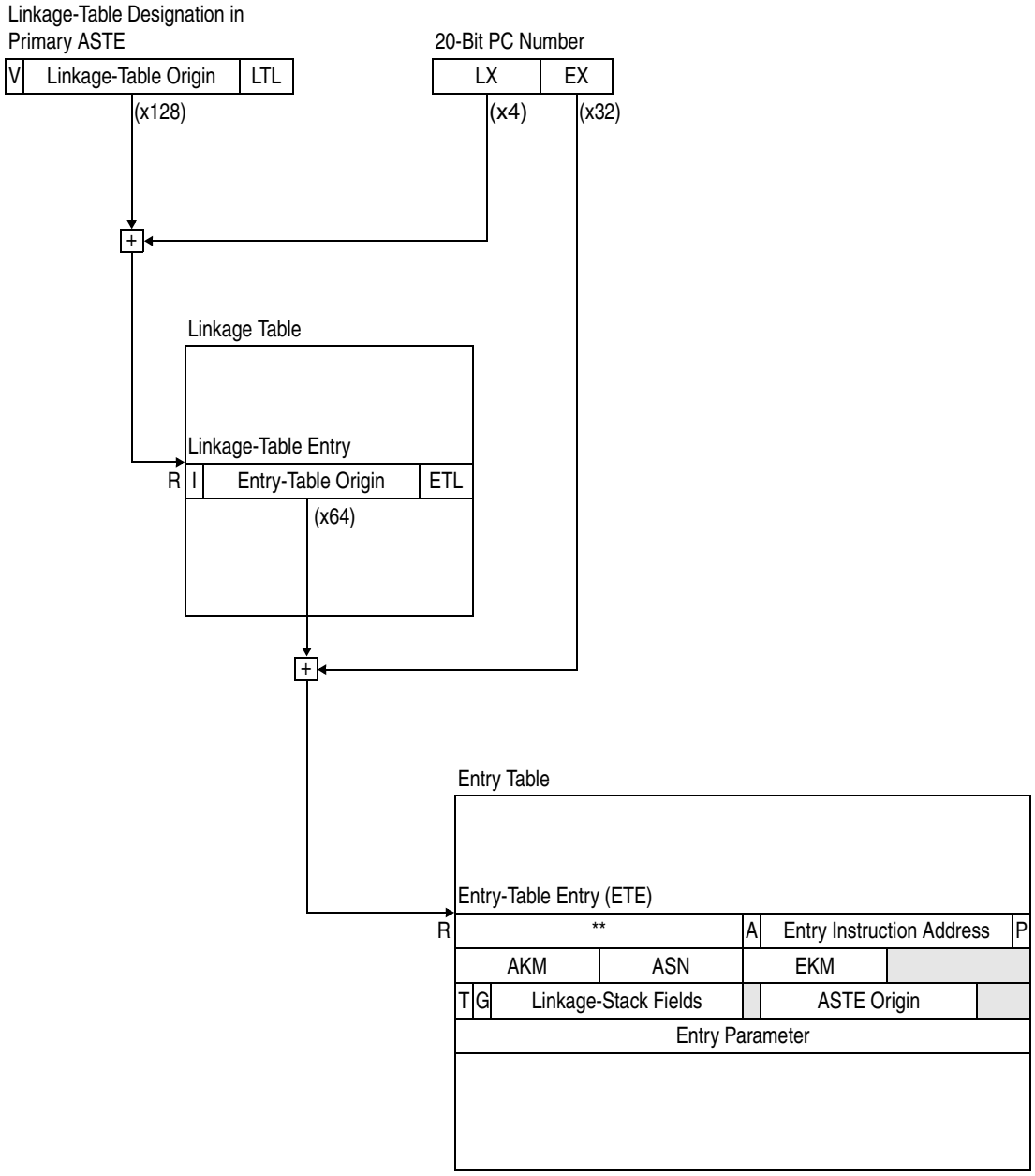
When ASN-and-LX reuse is not enabled, the LX is used to select an entry from the linkage table, the starting address and length of which are specified by the linkage-table designation in the primary ASTE. This entry designates the entry table to be used. The EX field of the PC number is used to select an entry from the entry table.

When ASN-and-LX reuse is enabled, the LFX is used to select an entry from the linkage first table, the starting address and length of which are specified by the linkage-first-table designation in the primary ASTE. This entry designates the linkage second table to be used. The LSX is then used to select an entry from the linkage second table, which entry designates the entry table to be used. The EX field of the PC number is then used to select an entry from the entry table.

When ASN-and-LX reuse is enabled, a linkage-second-table-entry sequence number (LSTESN) in general register 15 is compared to the LSTESN in the linkage-second-table entry if the LSTESN in the entry is nonzero.

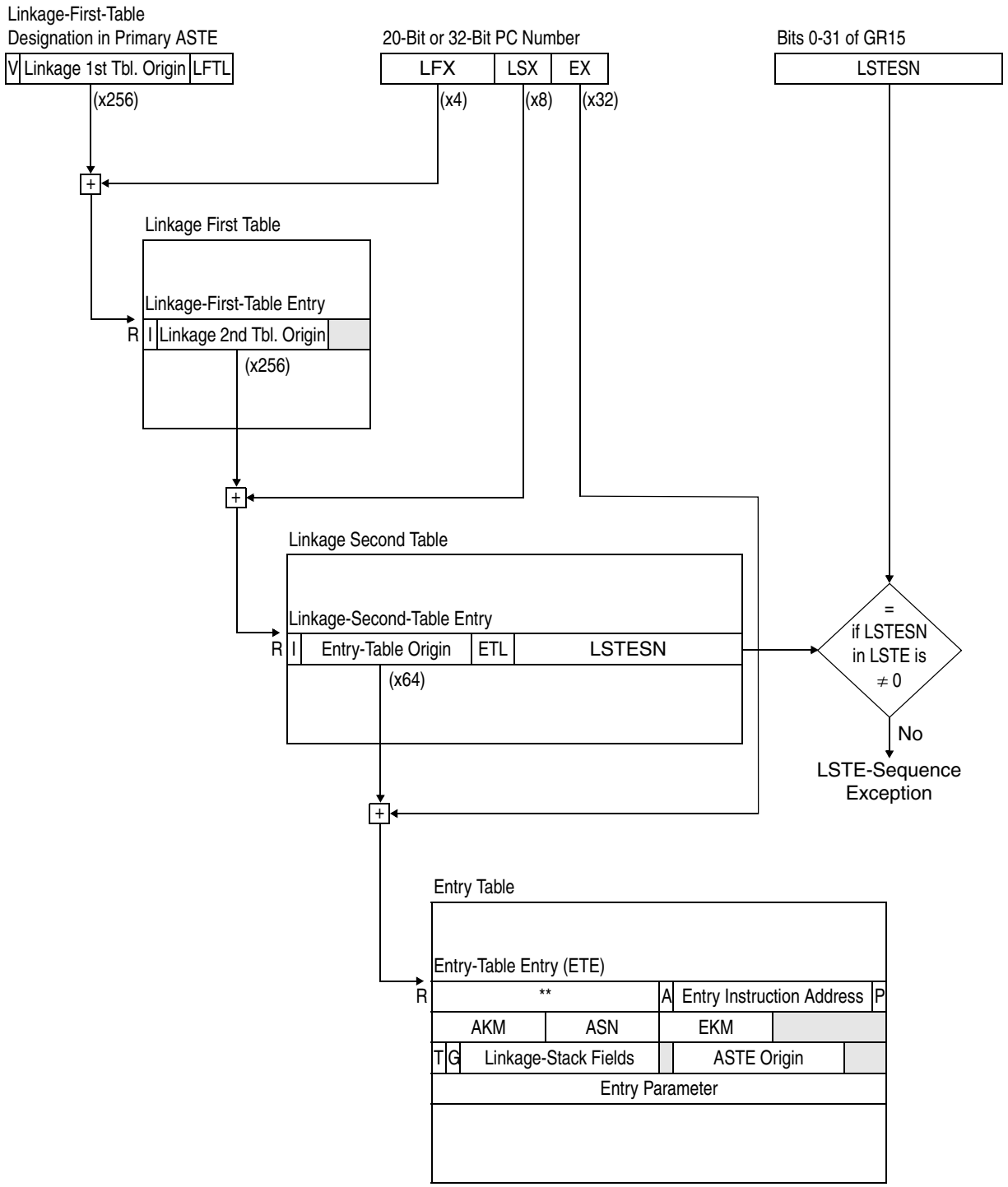
When, for the purposes of PC-number translation, accesses are made to main storage to fetch entries from the primary ASTE, linkage table, linkage first table, linkage second table, and entry table, key-controlled protection does not apply.

The PC-number-translation process is shown in Figure 5-8 on page 5-40 for when ASN-and-LX-reuse is not enabled and in Figure 5-9 on page 5-41 for when it is enabled.



R: Address is real.
**: First word and A of ETE are bits 0-32 of entry-instruction address (EIA) if G is one.

Figure 5-8. PC-Number Translation when ASN-and-LX Reuse Is Not Enabled



R: Address is real.
*: PC Number is 32 bits if bit 44 of the effective address is one. Bit 44 is not shown in this case.
**: First word and A of ETE are bits 0-32 of entry-instruction address (EIA) if G is one.

Figure 5-9. PC-Number Translation when ASN-and-LX Reuse Is Enabled

Obtaining the Linkage-Table or Linkage-First-Table Designation

The linkage-table or linkage-first-table designation is obtained from bytes 24-27 of the primary ASN-second-table entry, the starting address of which is specified by the contents of control register 5.

The 31-bit real address of the linkage-table or linkage-first-table designation is obtained by appending six zeros on the right to the primary-ASTE origin, bits 33-57 of control register 5, and adding 24. The addition cannot cause a carry into bit position 0. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

All four bytes of the linkage-table or linkage-first-table designation appear to be fetched concurrently from the primary ASTE as observed by other CPUs. The fetch access is not subject to protection. When the storage address which is generated for fetching the linkage-table or linkage-first-table designation designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed. Besides the linkage-table or linkage-first-table designation, no other field in the primary ASTE is examined.

Linkage-Table Lookup

When the ASN-and-LX-reuse facility is not installed or the ASN-and-LX-reuse-control bit, bit 44 of control register 0, is zero, the field fetched from the primary ASTE is a linkage-table designation. The linkage-index (LX) portion of the PC number, in conjunction with the linkage-table origin, is used to select an entry from the linkage table.

The 31-bit real address of the linkage-table entry is obtained by appending seven zeros on the right to the contents of bit positions 1-24 of the linkage-table designation and adding the linkage index, with two rightmost and 17 leftmost zeros appended. When a carry into bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from 2^{31} - 1 to 0. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

As part of the linkage-table-lookup process, the leftmost seven bits of the linkage index are compared against the linkage-table length, bits 25-31 of the linkage-table designation, to establish whether the addressed entry is within the linkage table. If the value in the linkage-table-length field is less than the value of the seven leftmost bits of the linkage index, an LX-translation exception is recognized.

All four bytes of the linkage-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address which is generated for fetching the linkage-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Bit 0 of the linkage-table entry specifies whether the entry table corresponding to the linkage index is available. This bit is inspected, and, if it is one, an LX-translation exception is recognized.

When no exceptions are recognized in the process of linkage-table lookup, the entry fetched from the linkage table designates the origin and length of the corresponding entry table.

Linkage-First-Table Lookup

When the ASN-and-LX-reuse facility is installed and the ASN-and-LX-reuse-control bit, bit 44 of control register 0, is one, the field fetched from the primary ASTE is a linkage-first-table designation. The linkage-first-index (LFX) portion of the PC number, in conjunction with the linkage-first-table origin, is used to select an entry from the linkage first table.

When bit 44 of the effective address of PROGRAM CALL is zero, the PC number is 20 bits, the linkage first index is bits 44-50 of the effective address, and the 31-bit real address of the linkage-first-table entry is obtained by appending eight zeros on the right to the contents of bit positions 1-23 of the linkage-first-table designation and adding the linkage first index, with two rightmost and 22 leftmost zeros appended.

When bit 44 of the effective address of PROGRAM CALL is one, the PC number is 32 bits, the linkage first index is bits 32-43 of the effective address appended on the left of bits 45-50 of the effective address, and the 31-bit real address of the linkage-first-table entry is obtained by appending eight zeros on the right to the contents of bit positions 1-23 of the linkage-first-table designation and adding the linkage first index, with two rightmost and 11 leftmost zeros appended.

When bit 44 of the effective address is one and a carry into bit position 0 occurs during the addition to form the address of a linkage-first-table entry, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from 2^{31} - 1 to 0.

When bit 44 of the effective address is either zero or one, the 31-bit address of the linkage-first-table entry is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

As part of the linkage-first-table-lookup process when bit 44 of the effective address is one, the leftmost 12 bits of the linkage first index are compared against the linkage-first-table length, bits 24-31 of the linkage-first-table designation, to establish whether the addressed entry is within the linkage first table. For this comparison, the linkage-first-table length is extended with four zero bits on the left. If the value of the extended linkage-first-table length is less than the value of the 12 leftmost bits of the linkage first index, an LFX-translation exception is recognized.

All four bytes of the linkage-first-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address which is generated for fetching the linkage-first-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Bit 0 of the linkage-first-table entry specifies whether the linkage second table corresponding to the linkage first index is available. This bit is inspected, and, if it is one, an LFX-translation exception is recognized.

When no exceptions are recognized in the process of linkage-first-table lookup, the entry fetched from the linkage first table designates the origin of the corresponding linkage second table.

Linkage-Second-Table Lookup

When the ASN-and-LX-reuse facility is installed and the ASN-and-LX-reuse-control bit, bit 44 of control register 0, is one, a linkage-second-table lookup is performed after a linkage-first-table entry has been fetched. The linkage-second-index (LSX) portion of the PC number, in conjunction with the linkage-second-table origin, is used to select an entry from the linkage second table.

The 31-bit real address of the linkage-second-table entry is obtained by appending eight zeros on the right to the linkage-second-table origin, bits 1-23 of the linkage-first-table entry, and adding the linkage second index, with three rightmost and 23 leftmost zeros appended. A carry into bit position 0 cannot occur during this addition. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

All eight bytes of the linkage-second-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address which is generated for fetching the linkage-second-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Bit 0 of the linkage-second-table entry specifies whether the entry table corresponding to the linkage second index is available. This bit is inspected, and, if it is one, an LSX-translation exception is recognized.

When no exceptions are recognized in the process of linkage-second-table lookup, the entry fetched from the linkage second table designates the origin of the corresponding entry table. The linkage-second-table entry contains a linkage-second-table-entry sequence number that may be used to test the correctness of the use of the linkage index.

Linkage-Second-Table-Entry-Sequence-Number Comparison

The linkage-second-table entry contains a linkage-second-table-entry sequence number (LSTESN) in bit positions 32-63. If this LSTESN is nonzero, it is compared to an LSTESN in bit positions 0-31 of general register 15, and an LSTE-sequence exception is recognized if the two LSTESNs are not equal.

Entry-Table Lookup

When the ASN-and-LX-reuse facility is not installed or is not enabled, the entry-table entry is located from a linkage-table entry. When the ASN-and-LX-reuse facility is installed and enabled, the entry-table entry is located from a linkage-second-table entry.

The entry-index (EX) portion of the PC number, in conjunction with the entry-table origin contained in the linkage-table or linkage-second-table entry, is used to select an entry from the entry table.

The 31-bit real address of the entry-table entry is obtained by appending six zeros on the right to the entry-table origin bits 1-25 of the linkage-table or linkage-second-table entry, and adding the entry index, with five rightmost and 18 leftmost zeros appended. When a carry into bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from 2^{31} - 1 to 0. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

As part of the entry-table-lookup process, the six leftmost bits of the entry index are compared against the entry-table length, bits 26-31 of the linkage-table or linkage-second-table entry, to establish whether the addressed entry is within the table. If the value in the entry-table length field is less than the value of the six leftmost bits of the entry index, an EX-translation exception is recognized.

The 32-byte entry-table entry is fetched by using the real address. The fetch of the entry appears to be word concurrent, as observed by other CPUs, with the leftmost word fetched first. The order in which the remaining seven words are fetched is unpredictable. The fetch access is not subject to protection. When the storage address which is generated for fetching the entry-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

The use that is made of the information fetched from the entry-table entry is described in the definition of the PROGRAM CALL instruction.

Recognition of Exceptions during PC-Number Translation

The exceptions which can be encountered during the PC-number-translation process and their priority are described in the definition of the PROGRAM CALL instruction.

Programming Note: The linkage-table or linkage-first-table designation is fetched successfully from the primary ASN-second-table entry regardless of the value of bit 0, the ASX-invalid bit, in the primary ASTE. A one value of this bit may cause an exception to be recognized in other circumstances.

Home Address Space

Facilities are provided which a privileged program, such as the control program, can use to obtain control in and access the home address space of a dispatchable unit (for example, a task).

Each dispatchable unit normally has an address space associated with it in which the control program keeps the principal control blocks that represent the dispatchable unit. This address space is called the home address space of the dispatchable unit. Different dispatchable units may have the same or different home address spaces. When the control program initiates a dispatchable unit, it may set the primary and secondary address spaces equal to the home address space of the dispatchable unit. Thereafter, because of the dispatchable unit's possible use of the PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, or SET SECONDARY ASN instruction, the control program normally cannot depend on either the primary address space or the secondary address space being the home address space when the home address space must be accessed, for example, during the processing by the control program of an interruption. Therefore, the control program normally must take some special action to ensure that the home address space is addressed when it must be accessed. The home-address-space facilities provide an efficient means to take this action.

The home-address-space facilities include:

- The home address-space-control element (HASCE) in control register 13. The HASCE is used by DAT in the same way as the primary address-space-control element (PASCE) in control register 1 and the secondary address-space-control element (SASCE) in control register 7.
- Home-space mode, which results when DAT is on and the address-space control, PSW bits 16 and 17, has the value 11 binary. When the CPU is in the home-space mode, instruction and logical addresses are home virtual addresses and are translated by DAT by means of the HASCE.
- The ability of the RESUME PROGRAM, SET ADDRESS SPACE CONTROL, and SET ADDRESS SPACE CONTROL FAST instructions to set the home-space mode in the supervisor state, and the ability of the INSERT ADDRESS

SPACE CONTROL instruction to return an indication of the home-space mode.

- The home space-switch-event control, bit 57 of control register 13.
- Recognition of a space-switch event upon completion of a RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST instruction if the CPU was in the home-space mode before or after the operation but not both before and after the operation, if any of the following is true: (1) the primary space-switch-event control, bit 57 of control register 1, is one, (2) the home space-switch-event control is one, or (3) a PER event is to be indicated.

The space-switch event can be used to enable or disable PER or tracing when fetching of instructions begins or ends in particular address spaces.

Access-Register Introduction

Many of the functions related to access registers are described in this section and in “Subroutine Linkage without the Linkage Stack” on page 5-14, “Access-Register Translation” on page 5-53, and “Sequence of Storage References” on page 5-113. Additionally, translation modes and access-list-controlled protection are described in Chapter 3, “Storage”; the PER means of restricting storage-alteration events to designated address spaces and the handling of access registers during resets and during the store-status operation are described in Chapter 4, “Control”; interruptions are described in Chapter 6, “Interruptions”; instructions are described in Chapter 7, “General Instructions”, and Chapter 10, “Control Instructions”; the handling of access registers during a machine-check interruption and the programmed validation of the access registers are described in Chapter 11, “Machine-Check Handling”; and the alter-and-display controls for access registers are described in Chapter 12, “Operator Facilities.”

Summary

These major functions are provided:

- A maximum of 16 address spaces, including the instruction space, for immediate and simultaneous use by a semiprivileged program; the address spaces are specified by 16 registers called access registers.
- Instructions for examining and changing the contents of the access registers.

In addition, control and authority mechanisms are incorporated to control these functions.

Access registers allow a sequence of instructions, or even a single instruction such as MOVE (MVC) or MOVE LONG (MVCL), to operate on storage operands in multiple address spaces, without the requirement of changing either the translation mode or other control information. Thus, a program residing in one address space can use the complete instruction set to operate on data in that address space and in up to 15 other address spaces, and it can move data between any and all pairs of these address spaces. Furthermore, the program can change the contents of the access registers in order to access still other address spaces.

The instructions for examining and changing access-register contents are unprivileged and are described in Chapter 7, “General Instructions.” They are:

- COPY ACCESS
- EXTRACT ACCESS
- LOAD ACCESS MULTIPLE
- LOAD ADDRESS EXTENDED
- SET ACCESS
- STORE ACCESS MULTIPLE

The privileged PURGE ALB instruction and COMPARE AND SWAP AND PURGE instruction are used in connection with access registers and are described in Chapter 10, “Control Instructions.”

Access registers specify address spaces when the CPU is in the access-register mode. The SET ADDRESS SPACE CONTROL and SET ADDRESS SPACE CONTROL FAST instructions allow setting of the access-register mode, and the INSERT ADDRESS SPACE CONTROL instruction provides an indication of the access-register mode. The stacking PROGRAM CALL, PROGRAM RETURN, and RESUME PROGRAM instructions also allow setting of the access-register mode. All of these instructions are described in Chapter 10, “Control Instructions.”

Access registers are used in a special way by the BRANCH IN SUBSPACE GROUP instruction. The use of access registers by that instruction is

described in detail only in the definition of the instruction in Chapter 10, “Control Instructions.” However, “Subspace-Group Tables” on page 5-66 describes the use of the dispatchable-unit control table and the extended ASN-second-table entry by BRANCH IN SUBSPACE GROUP.

Access-Register Functions

Access-Register-Specified Address Spaces

The CPU includes sixteen 32-bit access registers numbered 0-15. In the access-register mode, which results when DAT is on and PSW bits 16 and 17 are 01 binary, an instruction B or R field that is used to specify the logical address of a storage operand designates not only a general register but also an access register. The designated general register is used in the ordinary way to form the logical address of the storage operand. The designated access register is used to specify the address space to which the logical address is relative. The access register specifies the address space by specifying an address-space-control element for the address space, and this address-space-control element is used by DAT to translate the logical address. An access register specifies an address-space-control element in an indirect way, not by containing the address-space-control element.

An access register may specify the primary or secondary address-space-control element in control register 1 or 7, respectively, or it may specify an address-space-control element contained in an ASN-second-table entry. In the latter case, the access register designates an entry in a table called an access list, and the designated access-list entry in turn designates the ASN-second-table entry.

The process of using the contents of an access register to obtain an address-space-control element for use by DAT is called access-register translation (ART). This is depicted in Figure 5-10.

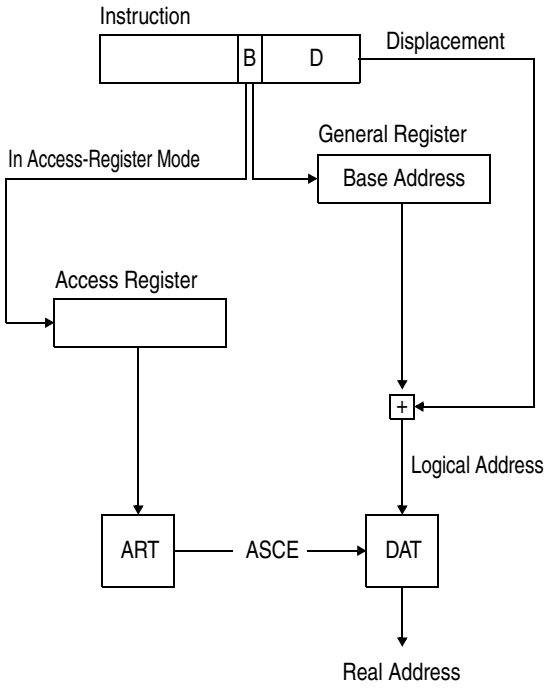


Figure 5-10. Use of Access Registers

An access register is said to specify an AR-specified address space by means of an AR-specified address-space-control element. The virtual addresses in an AR-specified address space are called AR-specified virtual addresses.

In the access-register mode, whereas all storage-operand addresses are AR-specified virtual, instruction addresses are primary virtual.

Designating Access Registers: In the access-register mode, an instruction B or R field designates an access register, for use in access-register translation, under the following conditions:

- The field is a B field which designates a general register containing a base address. The base address is used, along with a displacement (D) and possibly an index (X), to form the logical address of a storage operand.
- The field is an R field which designates a general register containing the logical address of a storage operand.

For example, consider the following instruction:

```
MVC 0(L,1),0(2)
```

The second operand, of length L, is to be moved to the first-operand location. The logical address of the second operand is in general register 2, and that of the first-operand location in general register 1. The address space containing the second operand is specified by access register 2, and that containing the first-operand location by access register 1. These two address spaces may be different address spaces, and each may be different from the current instruction address space (the primary address space).

When PSW bits 16 and 17 are 01, the B_2 field of the LOAD REAL ADDRESS and STORE REAL ADDRESS instructions designates an access register, for use in access-register translation, regardless of whether DAT is on or off. When the M_4 field of the LOAD PAGE-TABLE-ENTRY ADDRESS instruction is 0001 binary, or when the M_4 field is 0100 binary and bits 16-17 of the PSW are 01 binary, then the R_2 field designates an access register, for use in access-register translation, regardless of whether DAT is on or off. The B_1 field of TEST ACCESS designates an access register regardless of whether DAT is on or off and regardless of PSW bits 16-17.

The COMPARE AND FORM CODEWORD and UPDATE TREE instructions specify storage operands by means of implicitly designated general registers and access registers.

The MOVE TO PRIMARY and MOVE TO SECONDARY instructions specify storage operands by means of primary virtual and secondary virtual addresses, and access registers do not apply to these instructions. An exception is recognized when either of these instructions is executed in the access-register mode. The MOVE WITH KEY instruction can be used in place of MOVE TO PRIMARY and MOVE TO SECONDARY in the access-register mode. The MOVE WITH SOURCE KEY and MOVE WITH DESTINATION KEY instructions also can be used. The MOVE WITH OPTIONAL SPECIFICATIONS instruction can be used in place of MOVE TO PRIMARY, MOVE TO SECONDARY, MOVE WITH DESTINATION KEY, MOVE WITH KEY, or MOVE WITH SOURCE KEY in the access-register mode.

An instruction R field may designate an access register for other than the purpose of access-register translation.

The fields which may designate access registers, whether or not for access-register translation, are indicated in the summary figure at the beginning of each instruction chapter.

Obtaining the Address-Space-Control Element: This section and the following ones introduce the access-register-translation process and present the concepts related to access lists.

The address-space-control element specified by an access register is obtained by access-register translation as follows:

- If the access register contains 00000000 hex, the specified address-space-control element is the primary address-space-control element (PASCE), obtained from control register 1.
- If the access register contains 00000001 hex, the specified address-space-control element is the secondary address-space-control element (SASCE), obtained from control register 7.
- If the access register contains any other value, the specified address-space-control element is obtained from an ASN-second-table entry. The contents of the access register designate an access-list entry that contains the real origin of the ASN-second-table entry.

Access register 0 is treated in a special way by access-register translation; it is treated as containing 00000000 hex, and its actual contents are not examined. Thus, a logical address specified by means of a zero B or R field in the access-register mode is always relative to the primary address space, regardless of the contents of access register 0. However, there is one exception to how access register 0 is treated: the TEST ACCESS instruction uses the actual contents of access register 0, instead of treating access register 0 as containing 00000000 hex.

The treatment of an access register containing the value 00000000 hex as designating the current primary address space allows that address space to be addressed, in the access-register mode, without

requiring the use of an access-list entry. This is useful when the primary address space is changed by a space-switching PROGRAM CALL (PC-ss), PROGRAM RETURN (PR-ss), or PROGRAM TRANSFER (PT-ss) instruction. Similarly, the treatment of an access register containing the value 00000001 hex as designating the secondary address space allows that space to be addressed after a space-switching operation, again without requiring the use of an access-list entry.

The contents of the access registers are not changed by the PROGRAM CALL and PROGRAM TRANSFER instructions. Therefore, an access register containing 00000000 or 00000001 hex may specify a different address space after the execution of PROGRAM CALL or PROGRAM TRANSFER than before the execution. For example, if a space-switching PROGRAM CALL instruction is executed, an access register containing 00000000 hex specifies the old primary address space before the execution and the new primary address space after the execution.

Access Lists: The access-list entry that is designated by the contents of an access register can be located in either one of two access lists, the dispatchable-unit access list or the primary-space access list. A bit in the access register specifies which of the two access lists contains the designated entry. Both of the access lists reside in real or absolute storage. The locations of the access lists are specified by means of control registers 2 and 5.

Control register 2 contains the origin of a real-storage area called the dispatchable-unit control table. The dispatchable-unit control table contains the designation — the real or absolute origin, and length — of the dispatchable-unit access list.

Control register 5 contains the origin of a real-storage area called the primary ASN-second-table entry. The primary ASN-second-table entry contains the designation of the primary-space access list.

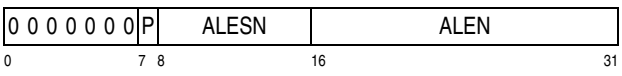
An access list, either the dispatchable-unit access list or the primary-space access list, contains some multiple of eight 16-byte entries, up to a maximum of 1,024 entries.

Programs and Dispatchable Units: When discussing access lists, it is necessary to distinguish between the terms “program” and “dispatchable unit.” A program is a sequence of instructions and may be referred to as a program module. A program may be a sequence of calling and called programs. A dispatchable unit, which is sometimes called a process or a task, is a unit of work that is performed through the execution of a program by one CPU at a time. The dispatchable-unit access list is intended to be associated with a dispatchable unit; that is, it is intended that a dispatchable unit have the same dispatchable-unit access list regardless of which program is currently being executed to perform the dispatchable unit. There is no mechanism, except for the LOAD CONTROL instruction, that changes the dispatchable-unit-control-table origin in control register 2.

The primary-space access list is associated with the primary address space that is specified by the primary ASN in control register 4 and the primary address-space-control element in control register 1. The primary-space access list that is available for use by a dispatchable unit changes as the primary address space of the dispatchable unit changes, that is, whenever a program in a different primary address space begins to be executed to perform the dispatchable unit. Whenever a LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL, PROGRAM RETURN, or PROGRAM TRANSFER instruction replaces the primary ASN in control register 4 and the primary address-space-control element in control register 1, it also replaces the primary-ASN-second-table-entry origin in control register 5.

Thus, for a dispatchable unit, the dispatchable-unit access list is intended to be constant (although its entries may be changed, as will be described), and the primary-space access list is a function of which program is being executed, through being a function of the primary address space of the program. Also, all dispatchable units and programs in the same primary address space have the same primary-space access list.

Access-List-Entry Token: The contents of an access register are called an access-list-entry token (ALET) since, in the general case, they designate an entry in an access list. An ALET has the following format:



The ALET contains a primary-list bit (P) that specifies which access list contains the designated access-list entry: the dispatchable-unit access list if the bit is

zero, or the primary-space access list if the bit is one. The specified access list is called the effective access list.

The ALET also contains an access-list-entry number (ALEN) which, when multiplied by 16, is the number of bytes from the beginning of the effective access list to the designated access-list entry. During access-register translation, an exception is recognized if the ALEN designates an entry that is outside the effective access list or if the leftmost seven bits in the ALET are not all zeros.

The access-list-entry sequence number (ALESN) in the ALET is described in the next section.

The above format of the ALET does not apply when the ALET is 00000000 or 00000001 hex.

An ALET can exist in an access register, in a general register, or in storage, and it has no special protection from manipulation by the problem program. Any program can transfer ALETs back and forth among access registers, general registers, and storage. A called program can save the contents of the access registers in any storage area available to it, load and use the access registers for its own purposes, and then restore the original contents of the access registers before returning to its caller.

Allocating and Invalidating Access-List Entries: It is intended that access lists be provided by the control program and that they be protected from direct manipulation by any problem program. This protection may be obtained by means of key-controlled protection or by placing the access lists in real storage not accessible by any problem program by means of DAT.

As determined by a bit in the entry, an access-list entry is either valid or invalid. A valid access-list entry specifies an address space and can be used by a suitably authorized program to access that space. An invalid access-list entry is available for allocation as a valid entry. It is intended that the control program provide services that allocate valid access-list entries and that invalidate previously allocated entries.

Allocation of an access-list entry may consist in the following steps. A problem program passes some kind of identification of an address space to the control program, and it passes a specification of either the dispatchable-unit access list or the primary-space access list. The control program checks, by some means, the authority of the problem program to access the address space. If the problem program is authorized, the control program selects an invalid entry in the specified access list, changes it to a valid entry specifying the subject address space, and returns to the problem program an access-list-entry token (ALET) that designates the allocated entry. The problem program can subsequently place the ALET in an access register in order to access the address space. Later, through the use of the invalidation service of the control program, the access-list entry that was allocated may be made invalid. An exception is recognized during access-register translation if an ALET is used that designates an invalid access-list entry.

It may be that a particular access-list entry is allocated, then invalidated, and then allocated again, this time specifying a different address space than the first time. To guard against erroneous use of an ALET that designates a conceptually wrong address space, an access-list-entry sequence number (ALESN) is provided in both the ALET and the access-list entry. When the control program allocates an access-list entry, it should place the same ALESN in the entry and in the designating ALET that it returns to the problem program. When the control program reallocates an access-list entry, it should change the value of the ALESN. An exception is recognized during access-register translation if the ALESN in the ALET used is not equal to the ALESN in the designated access-list entry.

The ALESN check is a reliability mechanism, not an authority mechanism, because the ALET is not protected from the problem program, and the problem program can change the ALESN in the ALET to any value. Also, this is not a fail-proof reliability mechanism because the ALESN is one byte and its value wraps around after 256 reallocations, assuming that the value is incremented by one for each reallocation.

Authorizing the Use of Access-List Entries: Although an access list is intended to be associated with either a dispatchable unit or a primary address space, the valid entries in the list are intended to be associated with the different programs that are executed, in some order, to perform the work of the dispatchable unit. It is intended that each program be able to have a particular authority that permits the use of only those access-list entries that are associated with the program. The authority being referred to here is represented by a 16-bit extended authorization index (EAX) in control register 8.

Other elements used in the related authorization mechanism are: (1) a private bit in the access-list entry, (2) an access-list-entry authorization index (ALEAX) in the access-list entry, and (3) the authority table.

A program is authorized to use an access-list entry, in access-register translation, if any of the following conditions is met:

1. The private bit in the access-list entry is zero. This condition provides a high-performance means to authorize any and all programs that are executed to perform the dispatchable unit.
2. The ALEAX in the access-list entry is equal to the EAX in control register 8. This condition provides a high-performance means to authorize only particular programs.
3. The EAX selects a secondary bit that is one in the authority table associated with the address space that is specified by the access-list entry. The authority table is locatable in that the access-list entry contains the real origin of the ASN-second-table entry (ASTE) for the address space, and the ASTE contains the real origin of the authority table. This condition provides another means, less well-performing than condition 2, for authorizing only particular programs. However, providing for condition 3 to be met instead of condition 2 can be advantageous because it permits several programs, each executed with a different EAX, all to use a single access-list entry to access a particular address space.

Access-register translation tests for the three conditions in the order indicated by their numbers, and a higher-numbered condition is not tested for if a lower-numbered condition is met. An exception is recognized if none of the conditions is met.

Figure 5-11 shows an example of how the authorization mechanism can be used. In the figure, “PB=0” means that the private bit is zero, and “PB=1” means that the private bit is one.

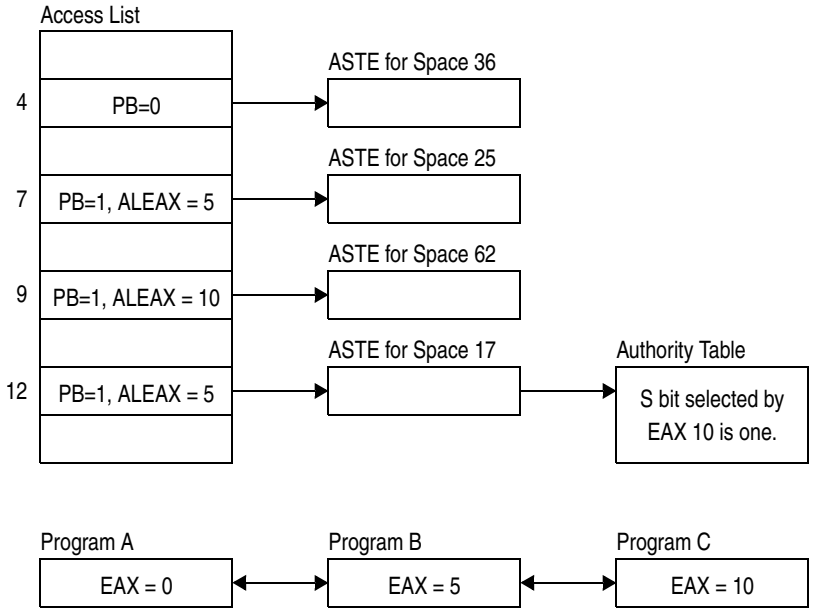


Figure 5-11. Example of Authorizing the Use of Access-List Entries

The figure shows an access list — assume it is a dispatchable-unit access list — in which the entries of interest are entries 4, 7, 9, and 12. Each access-list entry contains a private bit, an ALEAX, and the real origin of the ASTE for an address space. The private bit in entry 4 is zero, and, therefore, the value of the ALEAX in entry 4 is immaterial and is not shown. The private bits in entries 7, 9, and 12 are ones, and the ALEAX values in these entries are as shown. The numbers used to identify the address spaces (36, 25, 62, and 17) are arbitrary. They may be the ASNs of the address spaces; however, ASNs are in no way

used in access-register translation. Only the authority table for address space 17 is shown. In it, the secondary bit selected by EAX 10 is one. Assume that no secondary bits are ones in the authority tables for the other spaces.

The figure also shows a sequence of three programs, named A, B, and C, that is executed to perform the work of the dispatchable unit associated with the access list. These programs may be in the same or different address spaces. The EAX in control register 8 when each of these programs is executed is 0, 5, and 10, respectively.

Each of programs A, B, and C can use access-list entry (ALE) 4 to access address space 36 since the private bit in ALE 4 is zero. Program B can use ALE 7 to access space 25 because the ALEAX in the ALE equals the EAX for the program, and no other program can use this ALE. Similarly, only program C can use ALE 9. Program B can use ALE 12 because the ALEAX and EAX are equal, and program C can use it because A's EAX selects a secondary bit that is one in the authority table for space 17.

The example would be the same if programs A, B, and C were all in the same address space and the access list were the primary-space access list for that space.

An ALE in which the private bit is zero may be called public because the ALE can be used by any program, regardless of the value of the current EAX. An ALE in which the private bit is one may be called private because the ability of a program to use the ALE depends on the current EAX.

Notes on the Authorization Mechanism: An access list is a kind of capability list, in the sense in which the word "capability" is used in computer science. It is up to the control program to formulate the policies that are used to allocate entries in an access list, and the programmed authorization checking required during allocation may be very complex and lengthy. After a valid entry has been made in an access list, the access-register-translation process enforces the control-program policies in a well-performing way by means of the authorization mechanism described above.

Using access lists has an advantage over using only ASNs and authority tables. For example, assume that an access register could contain an ASN and that access-register translation would do ASN translation of the ASN and then use the EAX to test the authority table. This would make the EAX relevant to all existing address spaces, and, therefore, it would make the management of EAXs and their assignment to programs more difficult. With the actual definitions of the ALET and access-register translation, an EAX is relevant to only the address spaces that are represented in the current dispatchable-unit and primary-space access lists. Also, since ASN translation is not done as a part of access-register translation, the number of concurrently existing address spaces, as represented by ASN-second-table entries, can be greater than the number of available ASNs (64K).

The entry-table entry and linkage stack can be used to assign EAXs to programs and to change the EAX in control register 8 during program linkages. These components are introduced in "Linkage-Stack Introduction" on page 5-70. The privileged EXTRACT AND SET EXTENDED AUTHORITY instruction also is available for saving and changing the EAX in control register 8.

The SET SECONDARY ASN instruction and the authorization index (AX), bits 32-47 of control register 4, can play a role in the use of access registers. The space-switching form of SET SECONDARY ASN (SSAR-ss) establishes a new secondary address space if the secondary bit selected by the AX is one in the authority table associated with the new secondary space. The secondary space can be addressed by means of an ALET having the value 00000001 hex.

Revoking Accessing Capability: Another mechanism, which is a combined authority and integrity mechanism, is part of access-register translation, and it is described in this section.

An access-list entry (ALE) contains an ASN-second-table-entry sequence number (ASTESN), and so does the ASTE designated by the ALE. During access-register translation, the ASTESN in the ALE must equal the ASTESN in the designated ASTE; otherwise, an exception is recognized.

When the control program allocates an ALE, it should copy the ASTESN from the designated ASTE into the ALE. Subsequently, the control program can, in effect, revoke the addressing capability represented by the ALE by changing the ASTESN in the ASTE. Changing the ASTESN in the ASTE makes all previously usable ALEs that designate the ASTE unusable.

Making an ALE unusable may be required in either of two cases:

1. Some element of the control-program policy for determining the authority of a program to have access to the address space specified by the ASTE has changed. This may mean that some or all of the programs that were authorized to the address space, and for which ALEs have been allocated, are no longer authorized. Changing the ASTESN in the ASTE ends the usability of all ALEs that designate the ASTE. If this revocation of capability is to be selective, then, when an exception is recognized because of unequal ASTESNs, the control program can reapply its programmed procedures for determining authorization, and an ALE which should have remained usable can be made usable again by copying the new ASTESN into it. When the usability of an ALE is restored, the control program normally should cause reexecution of the instruction that encountered the exception.
2. The ASTE has been reassigned to specify a conceptually different address space, and ALEs which specified the old address space must not be allowed to specify the new one. (Bit 0 of the ASTE, the ASX-invalid bit, can be set to one to delete the assignment of the ASTE to an address space, and this prevents the use of the ASTE in access-register translation. But after reassignment, bit 0 normally is set back to zero.)

The ASTESN mechanism may be regarded as an authority mechanism in the first case above and as an integrity mechanism in the second.

The ASTESN mechanism is especially valuable because it avoids the need of the control program to keep track of the access lists that contain the ALEs that designate each ASTE. Furthermore, it avoids the need of searching through these access lists in order to find the ALEs and set them invalid, to prevent the use of the ALEs in access-register translation. The latter activity could be particularly time-consuming, or could present a particularly difficult management problem, because the access lists could be in auxiliary storage, such as a direct-access storage device, when the need arises to invalidate the ALEs.

The ASTESN is a four-byte field. Assuming a reasonable frequency of authorization-policy changes or address-space reassignments, the approximately four billion possible values of the ASTESN provide a fail-proof authority or integrity mechanism over the lifetime of the system.

Programming Note: Refer to “Modification of ART Tables” on page 5-66 for a discussion of ALB maintenance required when modifying any of the tables used in access-register translation.

Preventing Store References: The access-list entry contains a fetch-only bit which, when one, specifies that the access-list entry cannot be used to perform storage-operand store references. The principal description of the effect of the fetch-only bit is in “Access-List-Controlled Protection” on page 3-13.

Improving Translation Performance: Access-register translation (ART) conceptually occurs each time a logical address is used to reference a storage operand in the access-register mode. To improve performance, ART normally is implemented such that some or all of the information contained in the ART tables (access-list-designation sources, access lists, ASN second tables, and authority tables) is maintained in a special buffer referred to as the ART-lookaside buffer (ALB). The CPU necessarily refers to an ART-table entry in real storage only for the initial access to that entry. The information in the entry may be placed in the ALB, and subsequent translations may be performed using the information in the ALB.

The PURGE ALB instruction and the COMPARE AND SWAP AND PURGE instruction can be used to clear all information from the ALB after a change has been made to an ART-table entry in real storage.

Access-Register Instructions

The following instructions are provided for examining and changing the contents of access registers:

- COPY ACCESS
- EXTRACT ACCESS
- LOAD ACCESS MULTIPLE
- LOAD ADDRESS EXTENDED
- SET ACCESS
- STORE ACCESS MULTIPLE

The SET ACCESS instruction replaces the contents of a specified access register with the contents of bit positions 32-63 of the specified general register. Conversely, the EXTRACT ACCESS instruction moves the contents of an access register to bit positions 32-63 of a general register. The COPY

ACCESS instruction moves the contents of one access register to another.

The LOAD ACCESS MULTIPLE instruction loads a specified set of consecutively numbered access registers from a specified storage location whose length in words equals the number of access registers loaded. Conversely, the STORE ACCESS MULTIPLE instruction function stores the contents of a set of access registers at a storage location.

The LOAD ADDRESS EXTENDED instruction is similar to the LOAD ADDRESS instruction in that it loads a specified general register with an effective address specified by means of the B, X, and D fields of the instruction. In addition, LOAD ADDRESS EXTENDED operates on the access register having the same number as the general register loaded. When the address-space control, PSW bits 16 and 17, is 00, 10, or 11 binary, LOAD ADDRESS EXTENDED loads the access register with 00000000, 00000001, or 00000002 hex, respectively. When the address space control is 01 binary, LOAD ADDRESS EXTENDED loads the target access register with a value that depends on the B field of the instruction. If the B field is zero, LOAD ADDRESS EXTENDED loads the target access register with 00000000 hex. If the B field is nonzero, LOAD ADDRESS EXTENDED loads the target access register with the contents of the access register designated by the B field. However, in the last case when bits 0-6 of the access register designated by the B field are not all zeros, the results in the target general register and access register are unpredictable.

The address-space-control values 00, 01, 10, and 11 binary specify primary-space, access-register, secondary-space, and home-space mode, respectively, when DAT is on. LOAD ADDRESS EXTENDED functions the same regardless of whether DAT is on or off.

When used in access-register translation, the access-register values 00000000 and 00000001 hex specify the primary and secondary address spaces, respectively, and the value 00000002 hex designates entry 2 in the dispatchable-unit access list. Loading the target access register with 00000002 hex when the address-space control is 11 binary is intended to support assignment, by the control program, of entry 2 in the dispatchable-unit access list as specifying the home address space.

Access-Register Translation

Access-register translation is introduced in “Access-Register-Specified Address Spaces” on page 5-46.

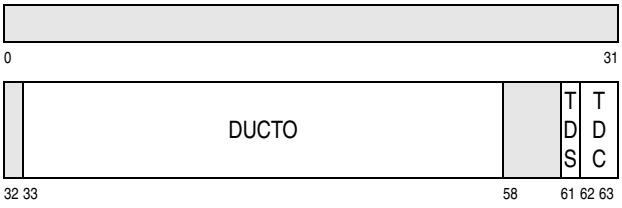
Access-Register-Translation Control

Access-register translation is controlled by an address-space control and by controls in control registers 2, 5, and 8. The address-space control, PSW bits 16 and 17, is described in “Translation Modes” on page 3-40. The other controls are described below.

Additional controls are located in the access-register-translation tables.

Control Register 2

The location of the dispatchable-unit control table is specified in control register 2. The register has the following format:



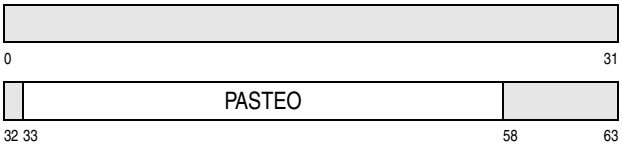
Dispatchable-Unit-Control-Table Origin (DUCTO): Bits 33-57 of control register 2, with six zeros appended on the right, form a 31-bit real address that designates the beginning of the dispatchable-unit control table. Access-register translation may obtain the dispatchable-unit access-list designation from the dispatchable-unit control table.

Transaction Diagnostic Scope (TDS): Bit 61 is described in “Transaction Diagnostic Scope (TDS)” on page 5-92.

Transaction Diagnostic Control (TDC): Bits 62-63 are described in “Transaction Diagnostic Control (TDC)” on page 5-92.

Control Register 5

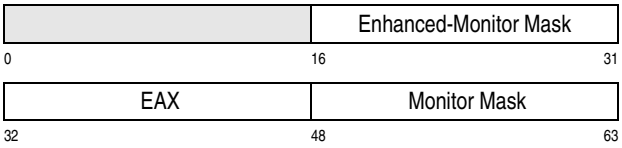
The location of the primary ASN-second-table entry is specified in control register 5. The register has the following format:



Primary-ASTE Origin (PASTEO): Bits 33-57 of control register 5, with six zeros appended on the right, form a 31-bit real address that designates the beginning of the primary ASN-second-table entry. Access-register translation may obtain the primary-space access-list designation from the primary ASTE. The primary-ASTE origin is implicitly set by LOAD ADDRESS SPACE PARAMETERS when it performs PASN translation and by the space-switching forms of PROGRAM CALL, PROGRAM RETURN, and PROGRAM TRANSFER. When any of these instructions places the primary-ASTE origin in control register 5, it also places zeros in bit positions 32 and 58-63 of the register and leaves bits 0-31 of the register unchanged. Bits 0-32 and 58-63 of control register 5 are subject to possible future assignment, and they should not be depended upon to be zeros.

Control Register 8

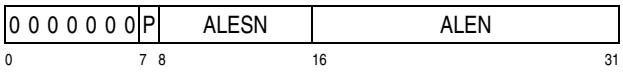
The extended authorization index is in control register 8. The register has the following format:



Extended Authorization Index (EAX): Bits 32-47 of control register 8 are the extended authorization index. During access-register translation, the EAX may be compared against the access-list-entry authorization index (ALEAX) in an access-list entry, and it may be used as an index to locate a secondary bit in an authority table. The EAX may be set by a stacking PROGRAM CALL operation, and it is restored by PROGRAM RETURN. The EAX can also be saved and set by the privileged instruction EXTRACT AND SET EXTENDED AUTHORITY.

Access Registers

There are sixteen 32-bit access registers numbered 0-15. The contents of an access register are called an access-list-entry token (ALET). An ALET has the following format:



The fields in the ALET are allocated as follows:

Primary-List Bit (P): When the ALET is not 00000000 or 00000001 hex, bit 7 specifies the access list to be used by access-register translation. When bit 7 is zero, the dispatchable-unit access list is used; this is specified by the dispatchable-unit access-list designation in the dispatchable-unit control table designated by the contents of control register 2. When bit 7 is one, the primary-space access list is used; this is specified by the primary-space access-list designation in the primary ASTE designated by the contents of control register 5.

Access-List-Entry Sequence Number (ALESN): Bits 8-15 may be used as a check on whether the access-list entry designated by the ALET has been invalidated and reallocated since the ALET was obtained. During access-register translation when the ALET is not 00000000 or 00000001 hex, bits 8-15 of the ALET are compared against the access-list-entry sequence number (ALESN) in the designated access-list entry.

Access-List-Entry Number (ALEN): When the ALET is not 00000000 or 00000001 hex, bits 16-31 of the ALET designate an entry in either the dispatchable-unit access list or the primary-space access list, as determined by bit 7. The access-list designation that is used is called the effective access-list designation; it consists of the effective access-list origin and the effective access-list length.

During access-register translation, the ALEN, with four zeros appended on the right, is added to the 31-bit real or absolute address specified by the effective access-list origin, and the result is the real or absolute address of the designated access-list entry. The ALEN is compared against the effective access-list length to determine whether the designated access-list entry is within the list, and an ALEN-translation exception is recognized if the entry is outside the list. Although the largest possible value of the

ALEN is 65,535, an access list can contain at most 1,024 entries.

Bits 0-6 must be zeros during access-register translation; otherwise, an ALET-specification exception is recognized.

When the ALET is 00000000 or 00000001 hex, it specifies the primary or secondary address space, respectively, and the above format does not apply.

Access register 0 usually is treated in access-register translation as containing 00000000 hex, and its actual contents are not examined; the access-register translation done as part of TEST ACCESS is the only exception. Access register 0 is also treated as containing 00000000 hex when it is designated by the B field of LOAD ADDRESS EXTENDED when PSW bits 16 and 17 are 01 binary. When access register 0 is specified for TEST ACCESS or as a source for COPY ACCESS, EXTRACT ACCESS, or STORE ACCESS MULTIPLE, the actual contents of the access register are used. Access register 0, like any other access register, can be loaded by COPY ACCESS, LOAD ACCESS MULTIPLE, LOAD ADDRESS EXTENDED, and SET ACCESS.

Another definition of ALETs 00000000 and 00000001 hex is given in “BRANCH IN SUBSPACE GROUP” on page 10-13.

Access-Register-Translation Tables

When the ALET being translated is not 00000000 or 00000001 hex, access-register translation performs a two-level lookup to locate first the effective access-list designation and then an entry in the effective access list. The effective access-list designation resides in real storage. The effective access list resides in real or absolute storage.

Access-register translation uses an origin in the access-list entry to locate an ASN-second-table entry, and it may perform a one-level lookup to locate an entry in an authority table. The ASN-second-table entry resides in real storage. The authority table resides in real or absolute storage.

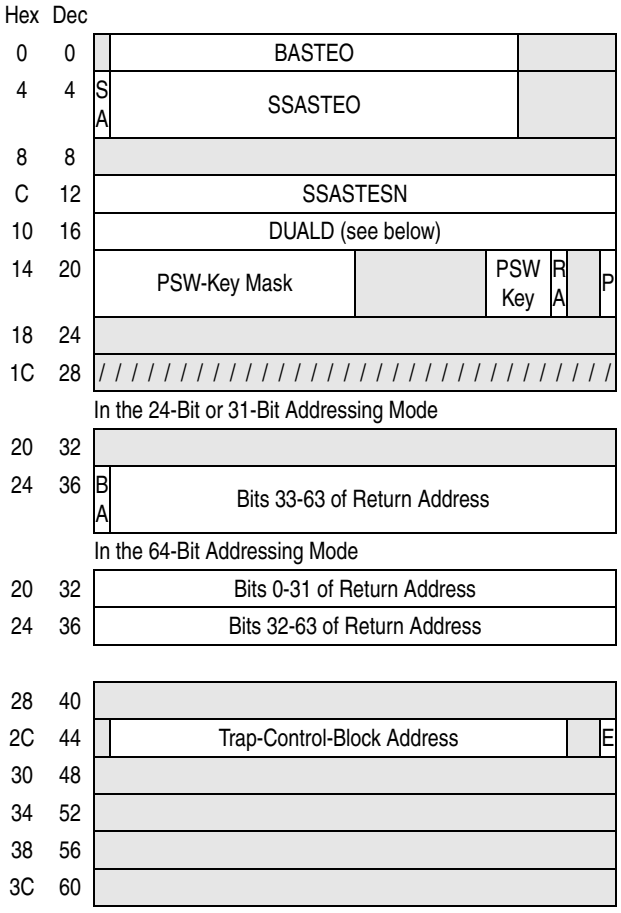
Authority-table entries are described in “Authority-Table Entries” on page 3-36. Access-list designations, access-list entries, and ASN-second-table entries are described in the following sections.

Dispatchable-Unit Control Table and Access-List Designations

When the ALET being translated is not 00000000 or 00000001 hex, access-register translation obtains the dispatchable-unit access-list designation if bit 7 of the ALET is zero, or it obtains the primary-space access-list designation if bit 7 is one. The obtained access-list designation is called the effective access-list designation.

The dispatchable-unit access-list designation (DUALD) is located in bytes 16-19 of a 64-byte area called the dispatchable-unit control table (DUCT). The DUCT resides in real storage, and its location is specified by the DUCT origin in control register 2.

The dispatchable-unit control table has the following format:



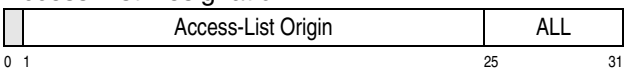
Bytes 0-7 (BASTEO, SA, and SSASTEO) and 12-15 (SSASTESN) of the DUCT are described in “Sub-

space-Group Dispatchable-Unit Control Table" on page 5-66. Bytes 20-23 (PSW key mask, PSW key, RA, and P) and 32-39 (BA and return address) are described in "BRANCH AND SET AUTHORITY" on page 10-7. Bytes 44-47 (trap-control-block address and E) are described in "TRAP" on page 10-174. Bytes 8-11, 24-27, 40-43, and 48-63 are reserved for possible future extensions and should contain all zeros. Bytes 28-31 are available for use by programming.

The primary-space access-list designation (PSALD) is located in bytes 16-19 of a 64-byte area called the primary ASN-second-table entry. The primary ASTE resides in real storage, and its location is specified by the primary-ASTE origin in control register 5. The format of the primary ASTE is described in "ASN-Second-Table Entries" on page 5-57.

The dispatchable-unit and primary-space access-list designations both have the same format, which is as follows:

Access-List Designation



The fields in the access-list designation are allocated as follows:

Access-List Origin: Bits 1-24 of the access-list designation, with seven zeros appended on the right, form a 31-bit address that designates the beginning of the access list. This address is treated unpredictably as either a real address or an absolute address.

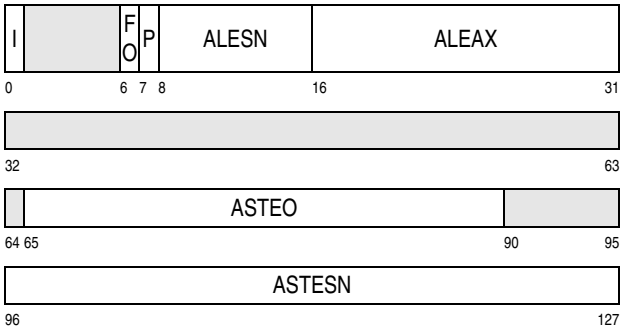
Access-List Length (ALL): Bits 25-31 of the access-list designation specify the length of the access list in units of 128 bytes, thus making the length of the access list variable in multiples of eight 16-byte entries. The length of the access list, in units of 128 bytes, is one more than the value in bit positions 25-31. The access-list length, with six zeros appended on the left, is compared against bits 0-12 of an access-list-entry number (bits 16-28 of an access-list-entry token) to determine whether the access-list-entry number designates an entry in the access list.

Bit 0 is reserved for a possible future extension and should be zero.

Programming Note: The maximum number of access-list entries allowed by an access-list designation is 1,024. There are two access lists available for use at any time. Therefore, a maximum of 2,048 sixteen-exabyte (16E) address spaces can be addressable without control-program intervention, which is a total of 2^{75} bytes.

Access-List Entries

The effective access list is the dispatchable-unit access list if bit 7 of the ALET being translated is zero, or it is the primary-space access list if bit 7 is one. The entry fetched from the effective access list is 16 bytes in length and has the following format:



The fields in the access-list entry are allocated as follows:

ALEN-Invalid Bit (I): Bit 0, when zero, indicates that the access-list entry specifies an address space. When bit 0 is one during access-register translation, an ALEN-translation exception is recognized.

Fetch-Only Bit (FO): Bit 6 controls which types of operand references are permitted to the address space specified by the access-list entry. When bit 6 is zero, both fetch-type and store-type references are permitted. When bit 6 is one, only fetch-type references are permitted, and an attempt to store causes a protection exception for access-list-controlled protection to be recognized and the operation to be suppressed.

Private Bit (P): Bit 7, when zero, specifies that any program is authorized to use the access-list entry in access-register translation. When bit 7 is one, authorization is determined as described for bits 16-31.

Access-List-Entry Sequence Number (ALESN): Bits 8-15 are compared against the ALESN in the ALET during access-register translation. Inequality causes an ALE-sequence exception to be recog-

nized. It is intended that the control program change bits 8-15 each time it reallocates the access-list entry.

Access-List-Entry Authorization Index (ALEAX): Bits 16-31 may be used to determine whether the program for which access-register translation is being performed is authorized to use the access-list entry. The program is authorized if any of the following conditions is met:

1. Bit 7 is zero.
2. Bits 16-31 are equal to the extended authorization index (EAX) in control register 8.
3. The EAX selects a secondary bit that is one in the authority table for the specified address space.

ASN-Second-Table-Entry Origin (ASTEO): Bits 65-89, with six zeros appended on the right, form the 31-bit real address of the ASTE for the specified address space. Access-register translation obtains the address-space-control element for the address space from the ASTE.

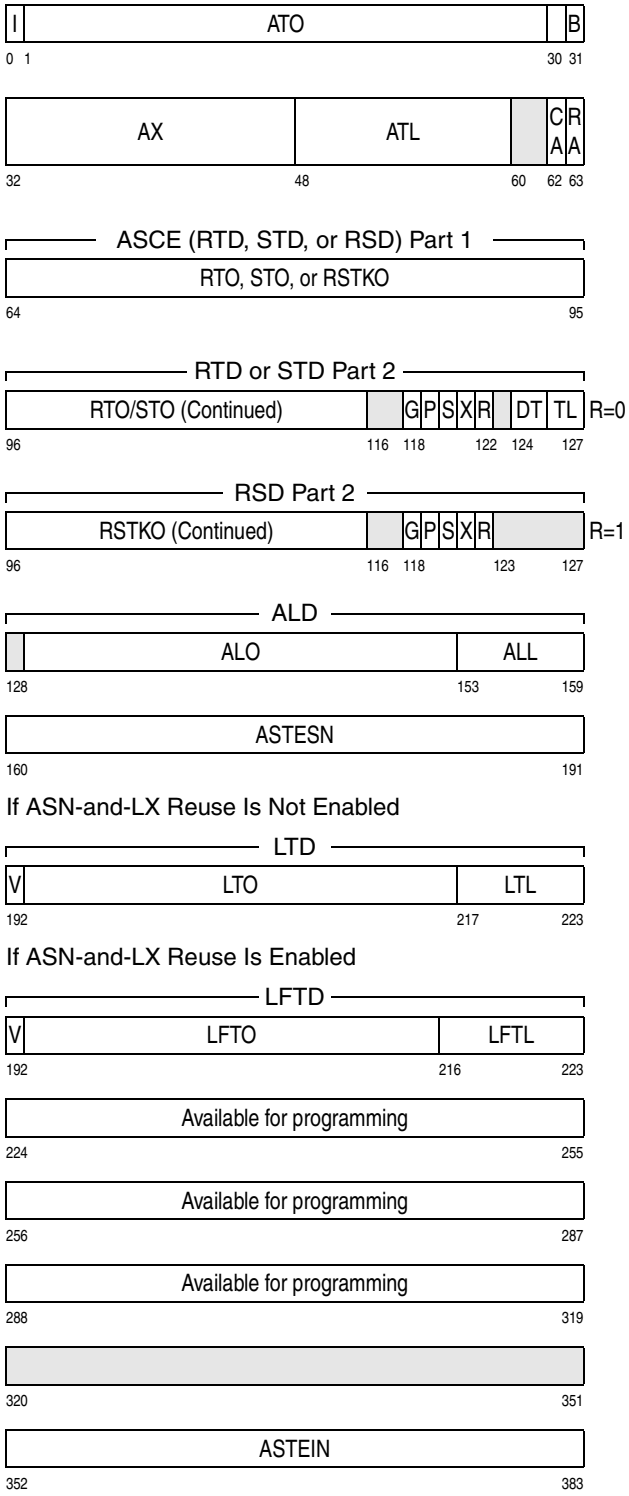
ASTE Sequence Number (ASTESN): Bits 96-127 may be used to revoke the addressing capability represented by the access-list entry. Bits 96-127 are compared against an ASTE sequence number (ASTESN) in the designated ASTE during access-register translation.

Bits 1-5, 32-64, and 90-95 are reserved for possible future extensions and should be zeros.

In both the dispatchable-unit access list and the primary-space access list, access-list entries 0 and 1 are not used in access-register translation. Bits 1-127 of access-list entry 0 and bits 1-63 of access-list entry 1 are reserved for possible future extensions and should be zeros. Bit 0 of access-list entries 0 and 1, and bits 64-127 of access-list entry 1, are available for use by programming. The control program should set bit 0 of access-list entries 0 and 1 to one in order to prevent the use of these entries by means of ALETs in which the ALEN is 0 or 1.

ASN-Second-Table Entries

The first 48 bytes of the 64-byte ASN-second-table entry have the following format:



The fields in bytes 0-47 of the ASN-second-table entry (ASTE) are defined with respect to certain

mechanisms and instructions in "ASN-Second-Table Entries" on page 3-31. The fields in the ASTE are defined with respect to the BRANCH IN SUBSPACE GROUP instruction in "Subspace-Group ASN-Second-Table Entries" on page 5-68. With respect to access-register translation only, and only for an instruction other than BRANCH IN SUBSPACE GROUP, the fields in the ASTE are allocated as follows:

ASX-Invalid Bit (I): Bit 0 controls whether the address space associated with the ASTE is available. When bit 0 is zero, access-register translation proceeds. When the bit is one, an ASTE-validity exception is recognized. When the primary-space ALD is fetched, the ASX-invalid bit in the primary ASTE is ignored.

Authority-Table Origin (ATO): Bits 1-29, with two zeros appended on the right, form a 31-bit address that designates the beginning of the authority table. This address is treated unpredictably as either a real address or an absolute address, although it is treated as a real address for ASN authorization. The authority table is accessed in access-register translation only if the private bit in the access-list entry is one and the access-list-entry authorization index (ALEAX) in the access-list entry is not equal to the extended authorization index (EAX) in control register 8.

Base-Space Bit (B): Bit 31 is ignored during access-register translation. Bit 31 is further described in "Subspace-Group ASN-Second-Table Entries" on page 5-68.

Authorization Index (AX): Bits 32-47 are not used in access-register translation.

Authority-Table Length (ATL): Bits 48-59 specify the length of the authority table in units of four bytes, thus making the authority table variable in multiples of 16 entries. The length of the authority table, in units of four bytes, is one more than the ATL value. The contents of the ATL field are used to establish whether the entry designated by a particular EAX is within the authority table. An extended-authority exception is recognized if the entry is not within the table.

Controlled-ASN Bit (CA): Bit 62 is not used in access-register translation.

Reusable-ASN Bit (RA): Bit 63 is not used in access-register translation.

Address-Space-Control Element (ASCE): Bits 64-127 are an eight-byte address-space-control element (ASCE) that may be a segment-table designation (STD), a region-table designation (RTD), or a real-space designation (RSD). (The term region-table designation is used to mean a region-first-table designation, region-second-table designation, or region-third-table designation.) The ASCE field is obtained as the result of access-register translation and is used by DAT to translate the logical address for the storage-operand reference being made. Bit 121, the space-switch-event control, is not used in or as a result of access-register translation. The other fields in the ASCE (RTO, STO, RSTKO, G, P, S, R, DT, and TL) are described in "Control Register 1" on page 3-42.

Access-List Designation (ALD): When this ASTE is designated by the primary-ASTE origin in control register 5, bits 128-159 are the primary-space access-list designation (PSALD). See the description of the access-list designation in "Dispatchable-Unit Control Table and Access-List Designations" on page 5-55. During access-register translation when the primary-list bit, bit 7, in the ALET being translated is one, the PSALD is the effective access-list designation.

ASN-Second-Table-Entry Sequence Number (ASTESN): Bits 160-191 are used to control revocation of the accessing capability represented by access-list entries that designate the ASTE. During access-register translation, bits 160-191 are com pared against the ASTESN in the access-list entry, and inequality causes an ASTE-sequence exception to be recognized. It is intended that the control program change the value of bits 160-191 when the authorization policies for the address space specified by the ASTE change or when the ASTE is reassigned to specify another address space.

Linkage-Table Designation (LTD) or Linkage-First-Table Designation (LFTD): Bits 192-223 are not used in access-register translation.

ASN-Second-Table-Entry Instance Number (ASTEIN): Bits 352-383 are not used in access-register translation.

Bits 224-319 in the ASTE are available for use by programming.

Programming Note: All unused fields in the ASTE, including the unused fields in bytes 0-31, bytes 40-43, and bytes 48-63, should be set to zeros. These fields are reserved for future extensions, and programs which place nonzero values in these fields may not operate compatibly on future machines.

Access-Register-Translation Process

This section describes the access-register-translation process as it is performed during a storage-operand reference in the access-register mode. The following instructions perform access-register translation as described in this section, but with the exceptions noted below.

- LOAD PAGE-TABLE-ENTRY ADDRESS when the M_4 field is 0001 binary, or when the M_4 field is 0100 binary and bits 16-17 of the PSW are 01 binary
- LOAD REAL ADDRESS and STORE REAL ADDRESS when PSW bits 16 and 17 are 01 binary
- TEST ACCESS in any translation mode
- TEST PROTECTION in the access-register mode

For LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, TEST ACCESS, and TEST PROTECTION, the following exceptions cause a setting of the condition code instead of being treated as program-interruption conditions:

- ALET specification
- ALEN translation
- ALE sequence
- ASTE validity
- ASTE sequence
- Extended authority

BRANCH IN SUBSPACE GROUP performs access-register translation as described in “BRANCH IN SUBSPACE GROUP” on page 10-13.

Access-register translation operates on the access register designated in a storage-operand reference in order to obtain an address-space-control element for use by DAT. When one of access-registers 1-15 is designated, the access-list-entry token (ALET) that is in the access register is used to obtain the address-space-control element. When access register 0 is designated, an ALET having the value 00000000 hex is used, except that TEST ACCESS uses the actual contents of access register 0.

When the ALET is 00000000 or 00000001 hex, the primary or secondary address-space-control element, respectively, is obtained.

When the ALET is other than 00000000 or 00000001 hex, the leftmost seven bits of the ALET are checked for zeros, the primary-list bit in the ALET and the contents of control register 2 or 5 are used to obtain the effective access-list designation, and the access-list entry number (ALEN) in the ALET is used to select an entry in the effective access list.

The access-list entry is checked for validity and for containing the correct access-list-entry sequence number (ALESN).

The ASN-second-table entry (ASTE) addressed by the access-list entry is checked for validity and for containing the correct ASN-second-table-entry sequence number (ASTESN).

Whether the program is authorized to use the access-list entry is determined through the use of one or more of: (1) the private bit and access-list-entry authorization index (ALEAX) in the access-list entry, (2) the extended authorization index (EAX) in control register 8, and (3) an entry in the authority table addressed by the ASN-second-table entry.

If a store-type reference is to be performed, the fetch-only bit in the access-list entry is checked for being zero.

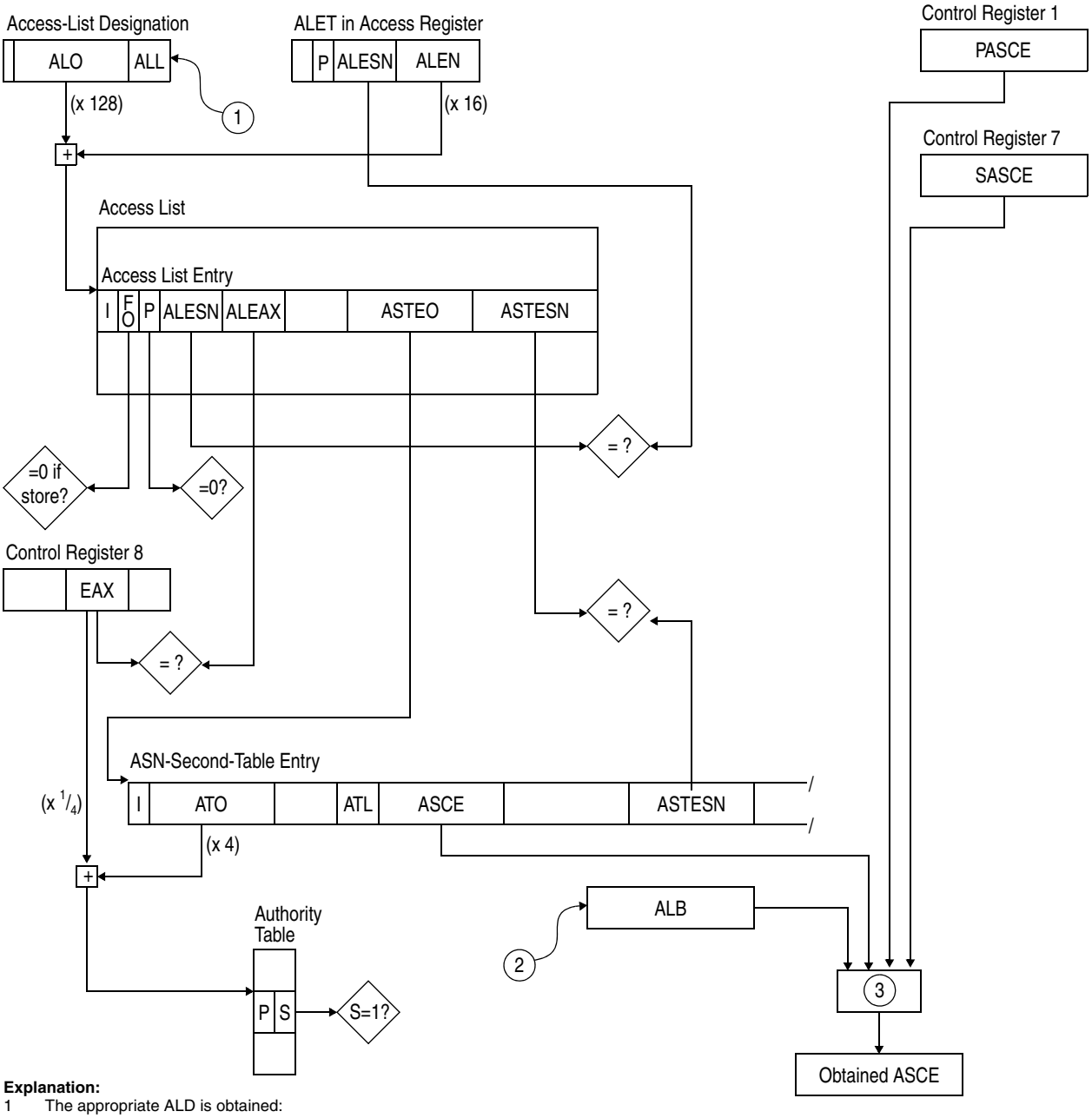
When no exceptions are recognized, the address-space-control element in the ASN-second-table entry is obtained.

In order to avoid the delay associated with references to real or absolute storage, the information fetched from real or absolute storage normally may also be placed in a special buffer, the ART-lookaside buffer (ALB), and subsequent translations involving the same information may be performed by using the contents of the ALB. The operation of the ALB is described in “ART-Lookaside Buffer” on page 5-64.

Whenever access to real or absolute storage is made during access-register translation for the purpose of fetching an entry from an access-list-designation source (DUCT or PASTE), access list, ASN second

table, or authority table, key-controlled protection does not apply.

The principal features of access-register translation, including the effect of the ALB, are shown in Figure 5-12 on page 5-61.



Explanation:

1 The appropriate ALD is obtained:
When P in the ALET is zero (and the ALET is not zero or one), the DUALD in the DUCT is obtained.
When P in the ALET is one, the PSALD in the primary ASTE is obtained.

2 Information, which may include the ALD-source origin, ALET, ALO, and EAX, is used to search the ALB. This information, along with information from the ALE, ASTE, and ATE, may be placed in the ALB.

3 The appropriate ASCE is obtained:
When the AR number is zero (except for TAR) or when the ALET is zero, the PASCE in CR 1 is obtained.
When the ALET is one, the SASCE in CR 7 is obtained.
When the ALET is larger than one:
- If a match exists, the ASCE from the ALB is used.
- If no match exists, tables from real or absolute storage are fetched. The resulting ASCE from the ASTE is obtained, and entries may be formed in the ALB.

Figure 5-12. Access-Register Translation

Selecting the Access-List-Entry Token

When one of access registers 1-15 is designated, or for the access register designated by the R_1. field of TEST ACCESS, access-register translation uses the access-list-entry token (ALET) that is in the access register. When access register 0 is designated, except for TEST ACCESS, an ALET having the value 00000000 hex is used, and the contents of access register 0 are not examined.

Obtaining the Primary or Secondary Address-Space-Control Element

When the ALET being translated is 00000000 hex, the primary address-space-control element in control register 1 is obtained. When the ALET is 00000001 hex, the secondary address-space-control element in control register 7 is obtained. In each of these two cases, access-register translation is completed.

Checking the First Byte of the ALET

When the ALET being translated is other than 00000000 or 00000001 hex, bits 0-6 of the ALET are checked for being all zeros. If bits 0-6 are not all zeros, an ALET-specification exception is recognized, and the operation is suppressed.

Obtaining the Effective Access-List Designation

The primary-list bit, bit 7, in the ALET is used to perform a lookup to obtain the effective access-list designation. When bit 7 is zero, the effective ALD is the dispatchable-unit ALD located in bytes 16-19 of the dispatchable-unit control table (DUCT). When bit 7 is one, the effective ALD is the primary-space ALD located in bytes 16-19 of the primary ASN-second-table entry (primary ASTE).

When bit 7 is zero, the 31-bit real address of the dispatchable-unit ALD is obtained by appending six zeros on the right to the DUCT origin, bits 33-57 of control register 2, and adding 16. The addition cannot cause a carry into bit position 0.

When bit 7 is one, the 31-bit real address of the primary-space ALD is obtained by appending six zeros on the right to the primary-ASTE origin, bits 33-57 of control register 5, and adding 16. The addition cannot cause a carry into bit position 0.

The obtained 31-bit real address is used to fetch the effective ALD — either the dispatchable-unit ALD or the primary-space ALD, depending on bit 7 of the ALET. The fetch of the effective ALD appears to be word concurrent, as observed by other CPUs, and is not subject to protection. When the storage address that is generated for fetching the effective ALD refers to a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed. When the primary-space ALD is fetched, bit 0, the ASX-invalid bit, in the primary ASTE is ignored.

Access-List Lookup

A lookup in the effective access list is performed. The effective access list is the dispatchable-unit access list if bit 7 of the ALET is zero, or it is the primary-space access list if bit 7 is one. The effective access list is treated unpredictably as being in either real or absolute storage.

The access-list-entry-number (ALEN) portion of the ALET is used to select an entry in the effective access list. The 31-bit real or absolute address of the access-list entry is obtained by appending seven zeros on the right to bits 1-24 of the effective ALD and adding the ALEN, with four rightmost and 11 leftmost zeros appended. When a carry into bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the access list to wrap from 2^{31} - 1 to 0. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

As part of the access-list-lookup process, the leftmost 13 bits of the ALEN are compared against the effective access-list length, bits 25-31 of the effective ALD, to establish whether the addressed entry is within the access list. For this comparison, the access-list length is extended with six leftmost zeros. If the value formed from the access-list length is less than the value in the 13 leftmost bits of the ALEN, an ALEN-translation exception is recognized, and the operation is nullified.

The 16-byte access-list entry is fetched by using the real or absolute address. The fetch of the entry appears to be word concurrent as observed by other CPUs, with the leftmost word fetched first. The order in which the remaining three words are fetched is unpredictable. The fetch access is not subject to protection. When the storage address that is generated for fetching the access-list entry refers to a location which is not available in the configuration, an

addressing exception is recognized, and the operation is suppressed.

Bit 0 of the access-list entry indicates whether the access-list entry specifies an address space by designating an ASN-second-table entry. This bit is inspected, and, if it is one, an ALEN-translation exception is recognized, and the operation is nullified.

When bit 0 is zero, the access-list-entry sequence number (ALESN) in bit positions 8-15 of the access-list entry is compared against the ALESN in the ALET to determine whether the ALET designates the conceptually correct access-list entry. Inequality causes an ALE-sequence exception to be recognized and the operation to be nullified.

Locating the ASN-Second-Table Entry

The ASN-second-table-entry (ASTE) origin in the access-list entry is used to locate the ASTE. Bits 65-89 of the access-list entry, with six zeros appended on the right, form the 31-bit real address of the ASTE.

The 64-byte ASTE is fetched by using the real address. The fetch of the entry appears to be word concurrent as observed by other CPUs, with the leftmost word fetched first, except that the fetch of the address-space-control element in the entry appears to be doubleword concurrent as observed by other CPUs. After the first word is fetched, the order in which the ASCE and the remaining words are fetched is unpredictable. The fetch access is not subject to protection. When the storage address that is generated for fetching the ASTE refers to a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Bit 0 of the ASTE indicates whether the ASTE specifies an address space. This bit is inspected, and, if it is one, an ASTE-validity exception is recognized, and the operation is nullified.

When bit 0 is zero, the ASTE sequence number (ASTESN) in bit positions 160-191 of the ASTE is compared against the ASTESN in bit positions 96-127 of the access-list entry to determine whether the addressing capability represented by the access-list entry has been revoked. Inequality causes an ASTE-sequence exception to be recognized and the operation to be nullified.

Authorizing the Use of the Access-List Entry

The private bit, bit 7, in the access-list entry is used to determine whether the program is authorized to use the access-list entry. The access-list-entry authorization index (ALEAX) in bit positions 16-31 of the access-list entry, the extended authorization index (EAX) in bit positions 32-47 of control register 8, and the authority table designated by the ASTE may also be used.

When the private bit is zero, the program is authorized, and the authorization step of access-register translation is completed.

When the private bit is one but the ALEAX is equal to the EAX, the program is authorized, and the authorization step of access-register translation is completed.

When the private bit is one and the ALEAX is not equal to the EAX, a process called the extended-authorization process is performed. Extended authorization uses the EAX to select an entry in the authority table designated by the ASTE, and it tests the secondary-authority bit in the selected entry for being one. The program is authorized if the tested bit is one.

Extended authorization is the same as the secondary-ASN-authorization process described in “ASN Authorization” on page 3-35, except as follows:

- The authority-table origin is treated as a real or absolute address instead of as a real address.
- The EAX in control register 8 is used instead of the authorization index (AX) in control register 4.
- When the value in bit positions 0-11 of the EAX is greater than the authority-table length (ATL) in the ASTE, an extended-authority exception is recognized instead of a secondary-authority exception. The operation is nullified if the extended-authority exception is recognized.

When the private bit is one, the ALEAX is not equal to the EAX, and the secondary bit in the authority-table entry selected by the EAX is not one, an extended-authority exception is recognized, and the operation is nullified.

Checking for Access-List-Controlled Protection

If a store-type reference is to be performed and the fetch-only bit, bit 6, in the access-list entry is one, a protection exception is recognized, and the operation is suppressed.

Obtaining the Address-Space-Control Element from the ASN-Second-Table Entry

When the ALET being translated is other than 00000000 or 00000001 hex and no exception is recognized in the steps described above, access-register translation obtains the address-space-control element from bit positions 64-127 of the ASTE.

Recognition of Exceptions during Access-Register Translation

The exceptions which can be encountered during the access-register-translation process and their priority are shown in the section "Access Exceptions" in Chapter 6, "Interruptions."

Programming Note: When updating an access-list entry or ASN-second-table entry, the program should change the entry from invalid to valid (set bit 0 of the entry to zero) as the last step of the updating. This ensures, because the leftmost word is fetched first, that words of a partially updated entry will not be fetched.

ART-Lookaside Buffer

To enhance performance, the access-register-translation (ART) mechanism normally is implemented such that access-list designations and information specified in access lists, ASN second tables, and authority tables are maintained in a special buffer, referred to as the ART-lookaside buffer (ALB). Access-list designations, access-list entries, ASN-second-table entries, and authority-table entries are collectively referred to as ART-table entries. The CPU necessarily refers to an ART-table entry in real or absolute storage only for the initial access to that entry. The information in the entry may be placed in the ALB, and subsequent ART operations may be performed using the information in the ALB. The presence of the ALB affects the ART process to the extent that (1) a modification of an ART-table entry in real or absolute storage does not necessarily have an immediate effect, if any, on the translation, (2) the comparison against the access-list length in an access-list designation that is in storage and used in a translation may be omitted if an ALB access-list entry is used, and (3) the comparison against the authority-table length in an ASN-second-table entry that is in storage and used in a translation may be omitted if an ALB authority-table entry is used. In a multiple-CPU configuration, each CPU has its own ALB.

Entries within the ALB are not explicitly addressable by the program.

Information is not necessarily retained in the ALB under all conditions for which such retention is possible. Furthermore, information in the ALB may be cleared under conditions additional to those for which clearing is mandatory.

ALB Structure

The description of the logical structure of the ALB covers the implementation by all systems operating as defined by z/Architecture. The ALB entries are considered as being of four types: ALB access-list designations (ALB ALDs), ALB access-list entries (ALB ALEs), ALB ASN-second-table entries (ALB ASTEs), and ALB authority-table entries (ALB ATEs). An ALB entry is considered as containing within it both the information obtained from the ART-table entry in real or absolute storage and the attributes used to fetch the ART-table entry from real or absolute storage.

There is not an indication in an ALB ALD of whether the ALD-source origin used to select the ALD in real storage was the dispatchable-unit-control-table origin or the primary-ASTE origin.

Note: The following sections describe the conditions under which information may be placed in the ALB, the conditions under which information from the ALB may be used for access-register translation, and how changes to the tables affect the ART process.

Formation of ALB Entries

The formation of ALB entries and the effect of any manipulation of the contents of an ART-table entry in real or absolute storage by the program depend on whether the entry is attached to a particular CPU and on whether the entry is valid.

The *attached* state of an ART-table entry denotes that the CPU to which the entry is attached can attempt to use the entry for access-register translation. The ART-table entry may be attached to more than one CPU at a time.

An access-list entry or ASN-second-table entry is *valid* when the invalid bit associated with the entry is zero. Access-list designations and authority-table entries have no invalid bit and are always valid. The primary-space access-list designation is valid regardless of the value of the invalid bit in the primary ASTE.

An ART-table entry may be placed in the ALB whenever the entry is attached and valid.

An access-list designation is attached to a CPU when the designation is within the dispatchable-unit control table designated by the dispatchable-unit-control-table origin in control register 2 or is within the primary ASTE designated by the primary-ASTE origin in control register 5.

An access-list entry is attached to a CPU when the entry is within the access list specified by either an attached access-list designation (ALD) or a usable ALB ALD. A usable ALB ALD is explained in the next section.

An ASN-second-table entry is attached to a CPU when it is designated by the ASTE origin in either an attached and valid access-list entry (ALE) or a usable ALB ALE. A usable ALB ALE is explained in the next section.

An authority-table entry is attached to a CPU when it is within the authority table designated by either an attached and valid ASN-second-table entry (ASTE) or a usable ALB ASTE. A usable ALB ASTE is explained in the next section.

Subject to the attached and valid constraints defined above, the CPU may form ALB entries in anticipation of future storage references or as a result of the speculative execution of instructions. Such ALB entries may be formed independent of the content of the access registers. See “Overlapped Operation of Instruction Execution” on page 5-114 for additional details.

Use of ALB Entries

The *usable* state of an ALB entry denotes that the CPU can attempt to use the ALB entry for access-register translation. A usable ALB entry attaches the next-lower-level table, if any, and may be usable for a particular instance of access-register translation.

An ALB ALD is in the usable state when the ALDSO field in the ALB ALD matches the current dispatchable-unit-control-table origin or the current primary-ASTE origin.

An ALB ALD may be used for a particular instance of access-register translation when either of the following conditions is met:

1. The primary-list bit in the ALET to be translated is zero, and the ALDSO field in the ALB ALD matches the current dispatchable-unit-control-table origin.
2. The primary-list bit in the ALET to be translated is one, and the ALDSO field in the ALB ALD matches the current primary-ASTE origin.

An ALB ALE is in the usable state when the ALO field in the ALB ALE matches the ALO field in an attached ALD or a usable ALB ALD.

An ALB ALE may be used for a particular instance of access-register translation when all of the following conditions are met:

1. The ALET to be translated has a value larger than 1. (If the ALET is 0 or 1, the contents of CR 1 or CR 7 are used.)
2. The ALO field in the ALB ALE matches the ALO field in the ALD or ALB ALD being used in the translation.
3. The ALEN field in the ALB ALE matches the ALEN field in the ALET to be translated.

An ALB ASTE is in the usable state when the ASTEO field in the ALB ASTE matches the ASTEO field in an attached and valid ALE or a usable ALB ALE.

An ALB ASTE may be used for a particular instance of access-register translation when the ASTEO field in the ALB ASTE matches the ASTEO field in the ALE or ALB ALE being used in the translation.

An ALB ATE may be used for a particular instance of access-register translation when both of the following conditions are met:

1. The ATO field in the ALB ATE matches the ATO field in the ASTE or ALB ASTE being used in the translation.
2. The EAX field in the ALB ATE matches the current EAX.

Modification of ART Tables

When an attached but invalid ART-table entry is made valid, or when an unattached but valid ART-table entry is made attached, and no entry formed from the ART-table entry is already in the ALB, the change takes effect no later than the end of the current instruction.

When an attached and valid ART-table entry is changed, and when, before the ALB is cleared of copies of that entry, an attempt is made to perform ART requiring that entry, unpredictable results may occur, to the following extent. The use of the new value may begin between instructions or during the execution of an instruction, including the instruction that caused the change. Moreover, until the ALB is cleared of copies of the entry, the ALB may contain both the old and the new values, and it is unpredictable whether the old or new value is selected for a particular ART operation. If the old and new values are used as representations of effective space designations, failure to recognize that the effective space designations are the same may occur, with the result that operand overlap may not be recognized. Effective space designations and operand overlap are discussed in"Interlocks within a Single Instruction" on page 5-116.

When LOAD ACCESS MULTIPLE or LOAD CONTROL changes the parameters associated with ART, the values of these parameters at the start of the operation are in effect for the duration of the operation.

All entries are cleared from the ALB by the execution of PURGE ALB, a COMPARE AND SWAP AND PURGE instruction that purges the ALB, and SET PREFIX, and by CPU reset.

Subspace Groups

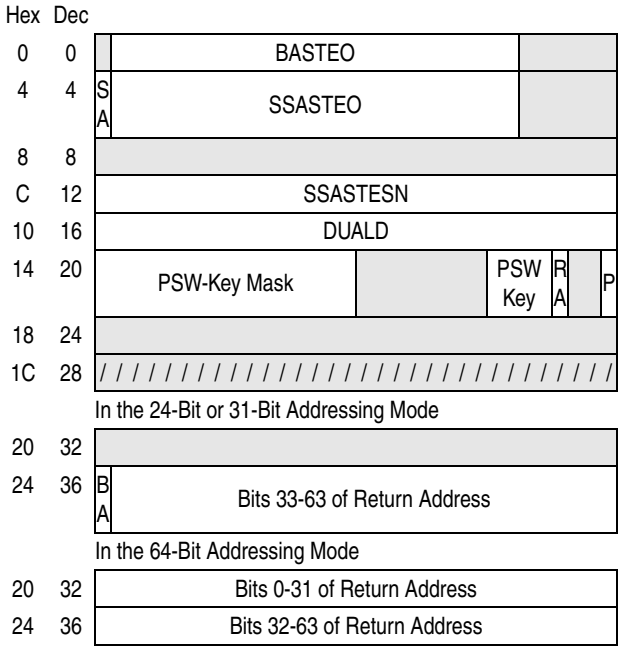
The subspace-group facility includes the BRANCH IN SUBSPACE GROUP instruction, allocations of fields in the address-space-control element, dispatchable-unit control table, and ASN-second-table entry, and subspace-replacement operations of the PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, and LOAD ADDRESS SPACE PARAMETERS instructions. BRANCH IN SUBSPACE GROUP is introduced in "Subroutine Linkage without the Linkage Stack" on page 5-14 and described in detail in "BRANCH IN SUBSPACE GROUP" on page 10-13.

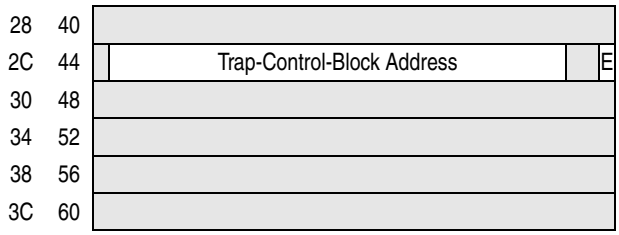
Subspace-Group Tables

This section describes the use of the dispatchable-unit control table and ASN-second-table entry by the subspace-group facility.

Subspace-Group Dispatchable-Unit Control Table

The dispatchable-unit control table has the following format:





The fields in the dispatchable-unit control table that are used by the subspace-group facility are allocated as follows:

Base-ASTE Origin (BASTEO): Bits 1-25 of bytes 0-3, with six zeros appended on the right, form a 31-bit real address that designates the beginning of the ASN-second-table entry that specifies the base space of a subspace group associated with the dispatchable unit. A comparison of bits 1-25 of bytes 0-3 to the primary-ASTE origin (PASTEO) in bit positions 33-57 of control register 5 is made by BRANCH IN SUBSPACE GROUP to determine whether the current primary address space is in the subspace group for the current dispatchable unit. For this comparison, either bits 1-25 may be compared to the PASTEO or the entire contents of bytes 0-3 may be compared to the contents of bit positions 33-63 of control register 5. A comparison of bits 1-25 of bytes 0-3 to the destination-ASTE origin (DASTEO) obtained from an access-list entry by access-register translation of an ALET other than ALETs 0 and 1 is made by BRANCH IN SUBSPACE GROUP to determine if the destination ASTE is the base-space ASTE. For this comparison, either bits 1-25 may be compared to the DASTEO or the entire contents of bytes 0-3 may be compared to the DASTEO with one leftmost and six rightmost zeros appended. A comparison of bits 1-25 of bytes 0-3 to an ASTE origin (ASTEO) obtained by ASN translation may be made by PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, and LOAD ADDRESS SPACE PARAMETERS. For this comparison, either bits 1-25 may be compared to the ASTEO or the entire contents of bytes 0-3 may be compared to the ASTEO with one leftmost and six rightmost zeros appended. When BRANCH IN SUBSPACE GROUP uses ALET 0, bits 1-25 of bytes 0-3, with six zeros appended on the right, designate the destination ASTE.

Subspace-Active Bit (SA): Bit 0 of bytes 4-7 indicates, when one, that the last BRANCH IN SUBSPACE GROUP instruction executed for the dispatchable unit transferred control to a subspace of the subspace group associated with the dispatchable unit. Bit 0 being zero indicates any one of the following: the last BRANCH IN SUBSPACE GROUP instruction executed for the dispatchable unit transferred control to the base space of the subspace group, BRANCH IN SUBSPACE GROUP has not yet been executed for the dispatchable unit, or the dispatchable unit is not associated with a subspace group. BRANCH IN SUBSPACE GROUP sets bit 0 of bytes 4-7 to one when it transfers control to a subspace of the subspace group associated with the dispatchable unit, and it sets bit 0 to zero when it transfers control to the base space of the subspace group.

Subspace-ASTE Origin (SSASTEO): Bits 1-25 of bytes 4-7, with six zeros appended on the right, form a 31-bit real address that designates the beginning of the ASN-second-table entry that specifies the subspace last given control by a BRANCH IN SUBSPACE GROUP instruction executed for the dispatchable unit. When BRANCH IN SUBSPACE GROUP transfers control to a subspace by means of an ALET other than ALET 1, it places the ASTEO for the subspace (the destination ASTEO) in bit positions 1-25 of bytes 4-7, places zeros in bit positions 26-31 of bytes 4-7, and sets the subspace-active bit, bit 0 of bytes 4-7, to one. When BRANCH IN SUBSPACE GROUP uses ALET 1 to transfer control to a subspace, bits 1-25 of bytes 4-7, with six zeros appended on the right, designate the destination ASTE, and BRANCH IN SUBSPACE GROUP sets the subspace-active bit to one and either sets bits 26-31 of bytes 4-7 to zeros or leaves those bits unchanged. However, if bits 1-25 are all zeros, a special-operation exception is recognized. When BRANCH IN SUBSPACE GROUP transfers control to the base space of the subspace group, it sets the subspace-active bit to zero, and bits 1-31 of bytes 4-7 remain unchanged. Bits 1-25 of bytes 4-7 may be used by PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, and LOAD ADDRESS SPACE PARAMETERS to set bits 0-55 and 57-63 of the primary ASCE in control register 1 or the secondary ASCE in control register 7 from the same bits of the ASCE in the subspace ASTE.

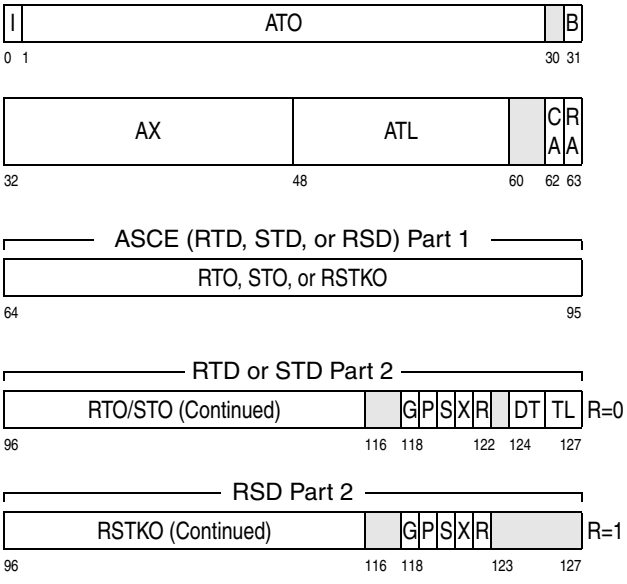
Subspace-ASTE Sequence Number (SS-ASTESN): Bytes 12-15 may be used to revoke the linkage capability represented by the SSASTEO, bits 1-25 of bytes 4-7, in the DUCT. When BRANCH

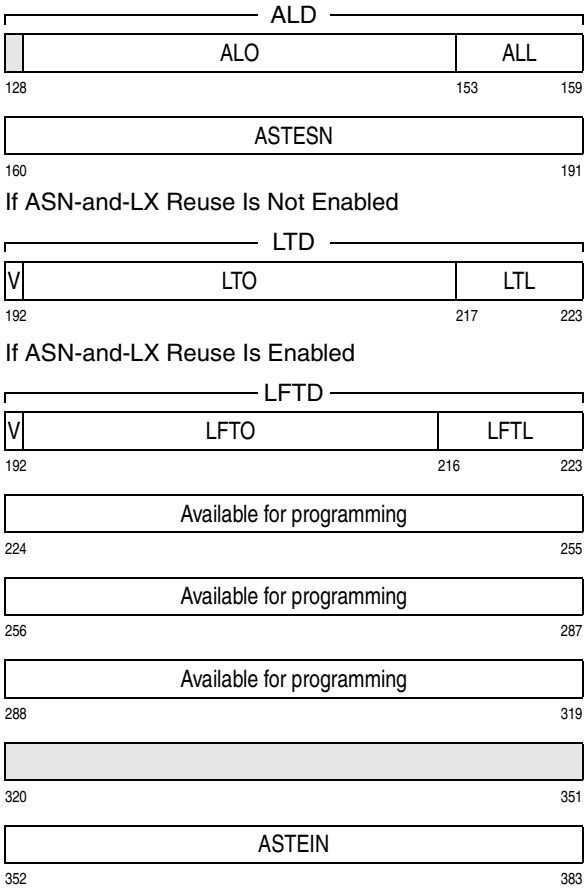
IN SUBSPACE GROUP transfers control to a subspace by means of an ALET other than ALET 1, it obtains the ASTESN in the subspace ASTE and places it in bytes 12-15. When BRANCH IN SUBSPACE GROUP uses ALET 1 to transfer control to a subspace, it compares bytes 12-15 to the ASTESN in the subspace ASTE, and it recognizes an ASTE-sequence exception if they are unequal. When the SSASTEO is used by PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, and LOAD ADDRESS SPACE PARAMETERS to set bits 0-55 and 57-63 of the primary ASCE in control register 1 or the secondary ASCE in control register 7 from the same bits of the ASCE in the subspace ASTE, those instructions first compare bytes 12-15 to the ASTESN in the subspace ASTE, and they recognize an ASTE-sequence exception if the two fields are unequal.

Bytes 16-19 are described in "Dispatchable-Unit Control Table and Access-List Designations" on page 5-55. Bytes 20-23 are described in "BRANCH AND SET AUTHORITY" on page 10-7. Bytes 32-39 and 44-47 are described in "TRAP" on page 10-174. Bytes 24-27, 40-43, and 48-63 are reserved for possible future extensions and should contain all zeros. Bytes 28-31 are available for use by programming.

Subspace-Group ASN-Second-Table Entries

The first 48 bytes of the 64-byte ASN-second-table entry have the following format:





For BRANCH IN SUBSPACE GROUP, the fields in bytes 0-47 of the ASTE are allocated as follows:

ASX-Invalid Bit (I): Bit 0 controls whether the address space associated with the ASTE is available. When bit 0 is zero during access-register translation of ALET 1 or an ALET other than 0 and 1 for BRANCH IN SUBSPACE GROUP, the translation proceeds. When the bit is one, an ASTE-validity exception is recognized. The bit is ignored during access-register translation of ALET 0. When the ASTE is designated by a subspace-ASTE origin (SSASTEO) in a dispatchable-unit control table, bit 0 is also used as described in the definition of bits 160-191 (ASTESN).

Authority-Table Origin (ATO): Bits 1-29 are not used by BRANCH IN SUBSPACE GROUP.

Base-Space Bit (B): Bit 31 specifies, when one, that the address space associated with the ASTE is the base space of a subspace group. When BRANCH IN SUBSPACE GROUP uses an ALET

other than ALETs 0 and 1 to locate a destination ASTE, it recognizes a special-operation exception if the destination-ASTE origin does not equal the base-ASTE origin in the dispatchable-unit control table and bit 31 is one in the destination ASTE.

Authorization Index (AX): Bits 32-47 are not used by BRANCH IN SUBSPACE GROUP.

Authority-Table Length (ATL): Bits 48-59 are not used by BRANCH IN SUBSPACE GROUP.

Controlled-ASN Bit (CA): Bit 62 is not used by BRANCH IN SUBSPACE GROUP.

Reusable-ASN Bit (RA): Bit 63 is not used by BRANCH IN SUBSPACE GROUP.

Address-Space-Control Element (ASCE): Bits 64-127 are an eight-byte address-space-control element (ASCE) that may be a segment-table designation (STD), a region-table designation (RTD), or a real-space designation (RSD). (The term "region-table designation" is used to mean a region-first-table designation, region-second-table designation, or region-third-table designation.) The ASCE field is obtained as the result of access-register translation done for BRANCH IN SUBSPACE GROUP. When BRANCH IN SUBSPACE GROUP uses an ALET other than ALETs 0 and 1 to locate a destination ASTE, it recognizes a special-operation exception if the destination-ASTE origin does not equal the base-ASTE origin in the dispatchable-unit control table and the subspace-group-control bit, bit 118 (G), in the destination ASTE is zero. When BRANCH IN SUBSPACE GROUP transfers control to the base space of a subspace group associated with the current dispatchable unit, it places bits 64-127 in control register 1; otherwise, when BRANCH IN SUBSPACE GROUP transfers control to a subspace of the subspace group, it places bits 64-119 and 121-127 in bit positions 0-55 and 57-63, respectively, of control register 1. Bits 64-127 are used after ASN translation by PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, SET SECONDARY ASN, and LOAD ADDRESS SPACE PARAMETERS as described in "ASN-Second-Table Entries" on page 3-31.

Access-List Designation (ALD): When this ASTE is designated by the primary-ASTE origin in control register 5, bits 128-159 are the primary-space access-list designation (PSALD). During access-register translation when the primary-list bit, bit 7, in the ALET being translated is one, the PSALD is the effective access-list designation.

ASN-Second-Table-Entry Sequence Number (ASTESN): Bits 160-191 are used to control revocation of the accessing capability represented by access-list entries that designate the ASTE. During access-register translation, bits 160-191 are compared against the ASTESN in the access-list entry, and inequality causes an ASTE-sequence exception to be recognized.

Linkage-Table Designation (LTD) or Linkage-First-Table Designation (LFTD): Bits 192-223 are not used by BRANCH IN SUBSPACE GROUP.

ASN-Second-Table-Entry Instance Number (ASTEIN): Bits 352-383 are not used by BRANCH IN SUBSPACE GROUP.

Bits 224-319 in the ASTE are available for use by programming.

When the ASTE is designated by a subspace-ASTE origin (SSASTEO) in a dispatchable-unit control table, bits 160-191 are also used to control revocation of the linkage capability represented by that SSASTEO. When BRANCH IN SUBSPACE GROUP uses ALET 1 to transfer control to the subspace specified by the SSASTEO, or when PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, SET SECONDARY ASN, or LOAD ADDRESS SPACE PARAMETERS uses the SSASTEO to set bits 0-55 and 57-63 of the primary ASCE in control register 1 or the secondary ASCE in control register 7 from the same bits of the ASCE in the subspace ASTE, those instructions first test bit 0 of the subspace ASTE for being zero and recognize an ASTE-validity exception if it is not, and they then compare bits 160-191 to the subspace-ASTE sequence number (SSASTESN) in the dispatchable-unit control table and recognize an ASTE-sequence exception if there is an inequality. However, when either of the two named exception conditions exists for LOAD ADDRESS SPACE PARAMETERS, the instruction sets condition code 1 or 2 instead of recognizing the exception.

Programming Note: All unused fields in the ASTE, including the unused fields in bytes 0-31 and all of bytes 32-63, should be set to zeros. These fields are reserved for future extensions, and programs which place nonzero values in these fields may not operate compatibly on future machines.

Subspace-Replacement Operations

The subspace-group facility includes subspace-replacement operations of PROGRAM CALL, PROGRAM TRANSFER, PROGRAM RETURN, SET SECONDARY ASN, and LOAD ADDRESS SPACE PARAMETERS. The operations apply when the dispatchable unit for which any of the five named instructions is executed is in a state called subspace active. A dispatchable unit is subspace active if it has used BRANCH IN SUBSPACE GROUP to transfer control to a subspace of its subspace group and has not subsequently used BRANCH IN SUBSPACE GROUP to return control to the base space of the group.

The definitions of the subspace-replacement operations are included in the definitions of the five named instructions in Chapter 10, "Control Instructions." The operations are described in a general way as follows. Whenever (1) an address space is established as the primary or secondary address space as a result of ASN translation or (2) PROGRAM CALL obtains the origin of the ASN-second-table entry specifying a new primary address space from the entry-table entry used, then, if that address space is in a subspace group, as indicated by the subspace-group-control bit, bit 54 (G), being one in the address-space-control element (ASCE) for the address space (the new PASCE in control register 1 or SASCE in control register 7), and if the dispatchable unit is subspace-active, as indicated by the subspace-active bit, bit 0 (SA) of word 1, in the dispatchable-unit control table (DUCT) being one, the ASN-second-table-entry (ASTE) origin (ASTEO) for the address space, which was obtained by ASN translation or from the entry-table entry, is compared to the base-ASTE origin (BASTEO), bits 1-25 of word 0, in the DUCT. If that ASTEO and the BASTEO are equal, the following occurs. An ASTE-validity exception is recognized if bit 0 in the ASTE for the last subspace entered by the dispatchable unit, which ASTE is designated by the subspace-ASTE origin (SSASTEO) in the DUCT, is one. An ASTE-sequence exception is recognized if the ASTE-sequence number (ASTESN) in word 5 of the subspace ASTE does not equal the subspace ASTESN (SSASTESN) in word 3 of the DUCT. However, LOAD ADDRESS SPACE PARAMETERS sets a nonzero condition code instead of recognizing the ASTE-validity or ASTE-sequence exception. If no exception exists, bits 0-55 and 57-63 of the ASCE for the address space (the PASCE in control register 1 or SASCE in control register 7) are replaced by the same bits of the ASCE in word 2 of the subspace ASTE.

If an addressing exception is recognized when attempting to access the DUCT or subspace ASTE, the instruction execution is suppressed. If an ASTE-validity or ASTE-sequence exception is recognized, the instruction execution is nullified. Such nullification or suppression causes all control register contents to remain unchanged from what they were at the beginning of the instruction execution.

Key-controlled protection does not apply to any accesses to the DUCT or subspace ASTE.

For comparing the ASTEO obtained by ASN translation to the BASTEO, either the ASTEO may be compared to the BASTEO or the ASTEO, with one leftmost and six rightmost zeros appended, may be compared to the entire contents of word 0 of the DUCT.

When the SSASTEO in the DUCT is used to access the subspace ASTE, no check is made for whether the SSASTEO is all zeros.

The references to the DUCT and subspace ASTE are single-access references and appear to be word concurrent as observed by other CPUs. The words of the DUCT are accessed in no particular order. The words of the subspace ASTE are accessed in no particular order except that word 0 is accessed first.

The exceptions that can be recognized during a subspace-replacement operation are referred to collectively as the subspace-replacement exceptions and are listed in priority order in "Subspace-Replacement Exceptions" on page 6-56.

Linkage-Stack Introduction

Many of the functions related to the linkage stack are described in this section and in "Linkage-Stack Operations" on page 5-76. Additionally, tracing of the stacking PROGRAM CALL instruction and of the PROGRAM RETURN instruction is described in Chapter 5, "Program Execution"; interruptions in Chapter 6, "Interruptions"; and the instructions are described in Chapter 10, "Control Instructions."

Summary

These major functions are provided:

1. A table-based subroutine-linkage mechanism that provides PSW and control-register status changing and which saves and restores this status and the contents of general registers and access registers through the use of an entry in a linkage stack.
2. A branch-type linkage mechanism that uses the linkage stack.
3. Instructions for placing an additional two words of status in the current linkage-stack entry and for retrieving all of the status and the general-register and access-register contents that are in the entry.
4. An instruction for determining whether a program is authorized to use a particular access-list-entry token.
5. Aids for program-problem analysis.

In addition, control and authority mechanisms are incorporated to control these functions.

It is intended that a separate linkage stack be associated with and used by each dispatchable unit. The linkage stack for a dispatchable unit resides in the home address space of the dispatchable unit.

It is intended that a dispatchable unit's linkage stack be protected from the dispatchable unit by means of key-controlled protection. Key-controlled protection does not apply to the linkage-stack instructions that place information in or retrieve information from the linkage stack.

The linkage-stack functions are for use by programs considered to be semi privileged, that is, programs which are executed in the problem state but which are authorized to use additional functions. With these authorization controls, a nonhierarchical organization of programs may be established, with each program in a sequence of calling and called programs having a degree of authority that is arbitrarily different from those of programs before or after it in the sequence. The range of functions available to each program, and the ability to transfer control from one program to another, are prescribed in tables that are managed by the control program.

The linkage-stack instructions, which are semiprivileged, are described in Chapter 10, "Control Instructions." They are:

- BRANCH AND STACK
- EXTRACT STACKED REGISTERS
- EXTRACT STACKED STATE
- MODIFY STACKED STATE
- PROGRAM RETURN
- TEST ACCESS

In addition, the PROGRAM CALL instruction optionally forms an entry in the linkage stack. A PROGRAM CALL instruction that operates on the linkage stack is called stacking PROGRAM CALL. Recognition of PROGRAM CALL as stacking PROGRAM CALL is under the control of a bit in the entry-table entry.

Linkage-Stack Functions

Transferring Program Control

The use of the linkage stack permits programs operating at arbitrarily different levels of authority to be linked directly without the intervention of the control program. The degree of authority of each program in a sequence of calling and called programs may be arbitrarily different, thus allowing a nonhierarchical organization of programs to be established. Modular authorization control can be obtained principally by associating an extended authorization index with each program module. This allows program modules with different authorities to coexist in the same address space. On the other hand, the extended authorization index in effect during the execution of a called program module can be the one that is associated with the calling program module, thus allowing the called module to be executed with different authorities on behalf of different dispatchable units. Options concerning the PSW-key mask and the secondary ASN are other means of associating different authorities with different programs or with the same called program. The authority of each program is prescribed in tables that are managed by the control program. By setting up the tables so that the same program can be called by means of different PC numbers, the program can be assigned different authorities depending on which PC number is used to call it. The tables also allow control over which PC numbers can be used by a program to call other programs.

The stacking PROGRAM CALL and PROGRAM RETURN linkage operations can link programs residing in different address spaces and having different

levels of authority. The execution state and the contents of the general registers and access registers are saved during the execution of stacking PROGRAM CALL and are partially restored during the execution of PROGRAM RETURN. A linkage stack provides an efficient means of saving and restoring both the execution state and the contents of registers during linkage operations.

During the execution of a PROGRAM CALL instruction, the PC-number-translation process is performed to locate a 32-byte entry-table entry. When the PC-type bit in the entry-table entry is one, the stacking PROGRAM CALL operation is specified; otherwise, the basic PROGRAM CALL operation is specified.

In addition to the information applying to both basic PROGRAM CALL and stacking PROGRAM CALL (described in "PC-Number Translation" on page 5-33 and consisting of an authorization key mask and specifications of the new ASN, addressing mode, instruction address, problem state, PSW-key mask, primary-ASTE address, and entry parameter), the entry-table entry contains information that specifies options concerning the address-space control and PSW key in the PSW, and the PSW-key mask, extended authorization index, and secondary ASN in the control registers.

During the stacking PROGRAM CALL operation and by means of the additional information in the entry-table entry, the address-space control in the PSW can be set to specify either the primary-space mode or the access-register mode. The PSW key can be either left unchanged or replaced from the entry-table entry. The PSW-key mask in control register 3 can be either ORed to or replaced from the entry-table entry. The extended authorization index in control register 8 can be either left unchanged or replaced from the entry-table entry. The secondary ASN in control register 3 can be set equal to the primary ASN of either the calling program or the called program; thus, the ability of the called program to have access to the primary address space of the calling program can be controlled.

The stacking PROGRAM CALL operation always forms an entry, called a state entry, in the linkage stack to save the execution state and the contents of general registers 0-15 and access registers 0-15. The saved execution state includes a called-space identification, the numeric part of the PC number used, the updated PSW before any changes are made due to the entry-table entry, the extended authorization index, PSW-key mask, primary ASN, and secondary ASN existing before the operation, and the extended-addressing-mode bit existing after the operation. However, the value of the PER mask in the saved updated PSW is unpredictable. The linkage-stack state entry also contains an entry-type code that identifies the entry as one that was formed by PROGRAM CALL. If the ASN-and-LX-reuse facility is installed and enabled, the state entry also contains the primary ASN-second-table-entry instance number (primary ASTEIN) and secondary ASTEIN existing before the operation.

A space-switching operation occurs when the address-space number (ASN) specified in the entry-table entry is nonzero. When space switching occurs, the operation is called PROGRAM CALL with space switching (PC-ss), and the ASN in the entry-table entry is placed in control register 4 as a new primary ASN. When no space switching occurs, the operation is called PROGRAM CALL to current primary (PC-cp), and there is no change to the primary ASN in control register 4.

PROGRAM CALL with space switching obtains the new ASN from the entry-table entry and places it in control register 4 as the primary ASN. It obtains a new primary-ASTE origin from the entry-table entry and new primary address-space-control element from the new primary ASTE, and it places them in control registers 5 and 1, respectively. It sets the secondary address-space-control element in control register 7 equal to either the old primary address-space-control element, or the new one, depending on whether it sets the secondary ASN in control register 3 equal to the old primary ASN or the new one, respectively. PROGRAM CALL to current primary sets the secondary ASN equal to the primary ASN and the secondary address-space-control element equal to the primary address-space-control element.

If the ASN-and-LX-reuse facility is installed and enabled, PROGRAM CALL with space switching obtains the ASN-second-table-entry instance number (ASTEIN) from the new primary ASTE and places it in control register 4 as the new primary ASTEIN. It sets the secondary ASTEIN in control register 3 equal to either the old primary ASTEIN or the new one, depending on whether it set the secondary ASN equal to the old primary ASN or the new one, respectively. PROGRAM CALL to current primary sets the secondary ASTEIN equal to the primary ASTEIN.

The instruction PROGRAM RETURN restores most of the information saved in the linkage stack by the stacking PROGRAM CALL operation. It restores the PSW, extended authorization index, PSW-key mask, primary ASN, secondary ASN, and the contents of general registers 2-14 and access-registers 2-14. However, the PER mask in the current PSW remains unchanged. If the ASN-and-LX-reuse facility is installed and enabled, PROGRAM RETURN restores the primary ASTEIN and secondary ASTEIN. The operation of PROGRAM RETURN is referred to by saying that PROGRAM RETURN unstacks a state entry.

For PROGRAM RETURN, a space-switching operation occurs when the restored primary ASN is not equal to the primary ASN existing before the operation. When space switching occurs, the operation is called PROGRAM RETURN with space switching (PR-ss). When no space switching occurs, the operation is called PROGRAM RETURN to current primary (PR-cp).

PROGRAM RETURN with space switching performs ASN translation of the restored primary ASN to obtain a new primary-ASTE origin and a new primary address-space-control element, which it places in control registers 5 and 1, respectively. For PROGRAM RETURN with space switching or to current primary, (1) if the restored secondary ASN is the same as the restored primary ASN, the secondary address-space-control element in control register 7 is set equal to the new primary address-space-control element in control register 1, or (2) if the restored secondary ASN is not the same as the restored primary ASN, ASN translation and ASN authorization of the restored secondary ASN are performed to obtain a new secondary address-space-control element, which is placed in control register 7.

If the ASN-and-LX-reuse facility is installed and enabled, PROGRAM RETURN with space switching requires that the ASN-second-table-entry instance number (ASTEIN) in the new primary ASTE be equal to the primary ASTEIN saved in the state entry being unstacked. PROGRAM RETURN with space switching or to current primary, if the restored secondary ASN is not the same as the restored primary ASN, requires that the ASTEIN in the new secondary ASTE be equal to the saved ASTEIN. An exception is recognized if either of these requirements is not met.

The stacking PROGRAM CALL operation and the PROGRAM RETURN operation each can be performed successfully only in the primary-space mode or access-register mode. An exception is recognized when the CPU is in the real mode, secondary-space mode, or home-space mode.

A bit, named the unstack-suppression bit, can be set to one in a linkage-stack state entry to cause an exception if an attempt is made by PROGRAM RETURN to unstack the entry. When the bit is one, the entry still can be operated on by the instructions that add information to or retrieve information from the entry. The unstack-suppression bit is intended to allow the control program to gain control when an attempt is made to unstack a state entry in which the bit is one.

Branching Using the Linkage Stack

The execution state and the contents of the general registers and access registers can also be saved in the linkage stack by means of the instruction BRANCH AND STACK. BRANCH AND STACK uses a branch address as do the other branching instructions, instead of using a PC number. BRANCH AND STACK, along with PROGRAM RETURN, can link programs residing in the same address space and having the same level of authority; that is, BRANCH AND STACK does not change the execution state except for the instruction address.

BRANCH AND STACK forms a linkage-stack state entry that is almost the same as one formed by PROGRAM CALL. When it is necessary to distinguish between these two types of state entry, an entry formed by PROGRAM CALL is called a program-call state entry, and one formed by BRANCH AND STACK is called a branch state entry. A branch state entry differs from a program-call state entry in two ways: (1) it contains a different entry-type code, which identifies it as a branch state entry, and (2) it contains the basic-addressing-mode bit and instruction address existing after the operation instead of a called-space identification and the numeric part of the PC number used. These new values of PSW bits 32 and 64-127 are in addition to the complete PSW that is saved in the state entry.

For BRANCH AND STACK, the basic- and extended addressing mode bits and the instruction address that are part of the complete PSW saved in the state entry can be the current (at the beginning of the operation) addressing-mode bits and the updated instruction address (the address of the next sequential instruction), or they can be specified in a register.

This register can be one that had link information placed in it by a BRANCH AND LINK (BALR only), BRANCH AND SAVE, BRANCH AND SAVE AND SET MODE, or BRANCH AND SET MODE instruction. Thus, BRANCH AND STACK can be used either in a calling program or at (or near) the entry point of a called program, and, in either case, a PROGRAM RETURN instruction located at the end of the called program will return correctly to the calling program. The ability to use BRANCH AND STACK at an entry point allows the linkage stack to be used without changing old calling programs.

When the R_2 field of BRANCH AND STACK is zero, the instruction is executed without causing branching.

When PROGRAM RETURN unstacks a branch state entry, it ignores the extended authorization index, PSW-key mask, primary ASN, secondary ASN, primary ASTEIN, and secondary ASTEIN in the entry. The PROGRAM RETURN instruction restores the PSW and the contents of general registers 2-14 and access registers 2-14 that were saved in the entry. However, the PER mask in the current PSW remains unchanged.

BRANCH AND STACK can be executed successfully only in the primary-space mode or access-register mode. An exception is recognized when the CPU is in the real mode, secondary-space mode, or home-space mode. The unstack-suppression bit has the same effect in a branch state entry as it does in a program-call state entry.

Adding and Retrieving Information

The instruction MODIFY STACKED STATE can be used by a program to place two words of information, contained in a designated general-register pair, in an area, called the modifiable area, of the current linkage-stack state entry (a branch state entry or a program-call state entry). This is intended to allow a called program to establish a recovery routine that will be given control by the control program, if necessary.

The instructions EXTRACT STACKED REGISTERS and EXTRACT STACKED STATE can be used by a program to obtain any of the information saved in the current state entry by BRANCH AND STACK or PROGRAM CALL or placed there by MODIFY STACKED STATE. EXTRACT STACKED REGISTERS (EREGG) places the contents of a specified range of general registers and access registers back in the registers from which the contents were saved. EXTRACT STACKED REGISTERS (EREG) does the same except that it restores only bits 32-63 of the general registers and leaves bits 0-31 unchanged. EXTRACT STACKED STATE obtains pairs of words of the nonregister information saved or placed in a state entry and places them in bit positions 32-63 of a designated general-register pair. Alternatively, EXTRACT STACKED STATE obtains two doublewords containing a PSW saved in the state entry and places them in bit positions 0-63 of a designated general-register pair. If the ASN-and-LX-reuse facility is installed, EXTRACT STACKED STATE can place the contents of two words of the state entry in bit positions 0-31 of a designated general-register pair. These contents are the saved secondary ASTEIN and primary ASTEIN if the ASN-and-LX-reuse facility is enabled. EXTRACT STACKED STATE sets the condition code to indicate whether the current state entry is a branch state entry or a program-call state entry.

Testing Authorization

The instruction TEST ACCESS has as operands an access-list-entry token (ALET) in a designated access register and an extended authorization index (EAX) in a designated general register. TEST ACCESS applies the access-register-translation process, which uses the specified EAX instead of the current EAX in control register 8, to the ALET, and it sets the condition code to indicate the result. The condition code may indicate: (1) the ALET is 00000000 hex, (2) the ALET designates an entry in the dispatchable-unit access list and can be translated without exceptions in access-register translation, (3) the ALET designates an entry in the primary-space access list and can be translated without exceptions in access-register translation, or (4) the ALET is 00000001 hex or causes exceptions in access-register translation.

The principal purpose of TEST ACCESS is to allow a called program to determine whether an ALET passed to it by the calling program is authorized for use by the calling program by means of the calling program's EAX. This is in support of a possible programming convention in which a called program will not operate on an AR-specified address space by means of its own EAX unless the calling program is authorized to operate on that space by means of the calling program's EAX. The called program can obtain the calling program's EAX, for use by TEST

ACCESS, from the current linkage-stack state entry by means of the EXTRACT STACKED STATE instruction.

Another purpose of TEST ACCESS is to indicate the special cases in which the ALET is 00000000 hex, designating the primary address space, or 00000001 hex, designating the secondary address space. Because PROGRAM CALL may change the primary and secondary address spaces, ALETs 00000000 hex and 00000001 hex may designate different address spaces when used by the called program than when used by the calling program.

Still another purpose of TEST ACCESS is to indicate whether the ALET designates an entry in the primary-space access list since such a designation after the primary address space was changed by a space-switching program-linkage operation may be an error.

Program-Problem Analysis

To aid program-problem analysis, the option is provided of having a trace entry made implicitly for three additional linkage operations when the linkage stack is used. When branch tracing is on, a trace entry is made each time a BRANCH AND STACK instruction is executed and causes branching. When ASN tracing is on, a trace entry is made each time the stacking PROGRAM CALL operation is performed and each time PROGRAM RETURN unstacks a linkage-stack state entry formed by PROGRAM CALL. When mode tracing is on, a trace entry is made each time the stacking PROGRAM CALL operation or PROGRAM RETURN operation is performed and changes PSW bit 31, except that, for PROGRAM RETURN, a trace entry for mode tracing is not made if one due to ASN tracing is made. A detailed definition of tracing is contained in “Tracing” on page 4-12.

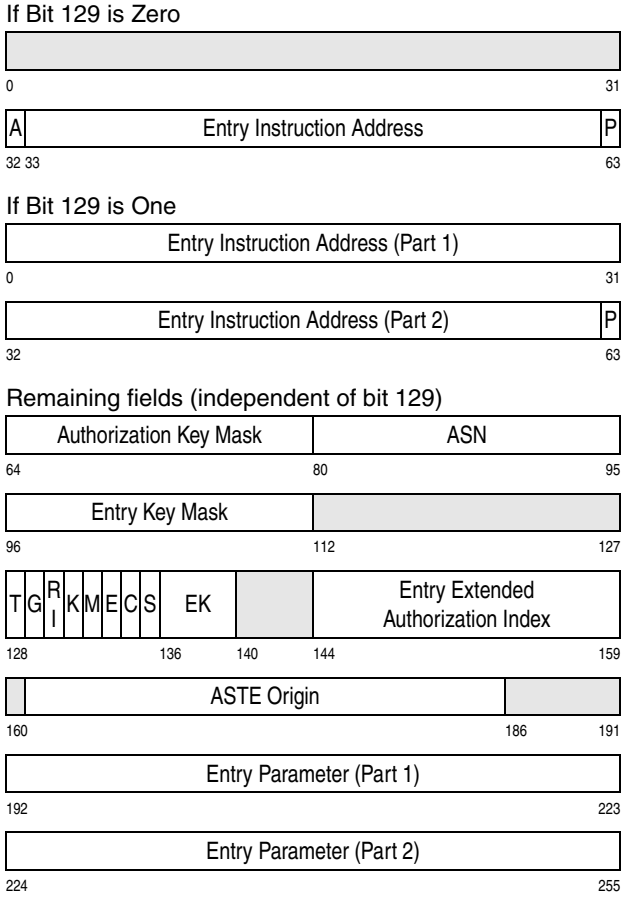
As a further analysis aid, BRANCH AND STACK when it causes branching, stacking PROGRAM CALL, and PROGRAM RETURN are also recognized as PER successful-branching events. For PROGRAM RETURN, the unstacked state entry may have been formed by BRANCH AND STACK or PROGRAM CALL.

The execution of a space-switching stacking PROGRAM CALL or PROGRAM RETURN instruction causes a space-switch event if the primary space-switch-event control is one before or after the operation or if a PER event is to be indicated.

Linkage-Stack Entry-Table Entries

All of the fields in the entry-table entry except bits 130-159 are described in “Entry-Table Entries” on page 5-37. This section describes only bits 130-159.

The entry-table entry has the following format:



The fields in bit positions 130-159 are allocated as follows:

Reserved (RI): Bit 130 is reserved for IBM use.

PSW-Key Control (K): Bit 131, when one, specifies that bits 136-139 are to replace the PSW key in the PSW as part of the stacking PROGRAM CALL operation. When this bit is zero, the PSW key remains unchanged. Bit 131 is ignored during the basic PROGRAM CALL operation.

PSW-Key-Mask Control (M): Bit 132, when one, specifies that bits 96-111 are to replace the PSW-key

mask in control register 3 as part of the stacking PROGRAM CALL operation. When this bit is zero, bits 96-111 are ORed into the PSW-key mask in control register 3 as part of the stacking PROGRAM CALL operation. Bit 132 is ignored during the basic PROGRAM CALL operation.

Extended-Authorization-Index Control (E): Bit 133, when one, specifies that bits 144-159 are to replace the current extended authorization index in control register 8 as part of the stacking PROGRAM CALL operation. When this bit is zero, the current extended authorization index remains unchanged. Bit 133 is ignored during the basic PROGRAM CALL operation.

Address-Space-Control Control (C): Bit 134, when one, specifies that bit 17 of the current PSW is to be set to one as part of the stacking PROGRAM CALL operation. When this bit is zero, bit 17 is set to zero. Because the CPU must be in either the primary-space mode or the access-register mode when a stacking PROGRAM CALL instruction is issued, the result is that the CPU is placed in the access-register mode if bit 134 is one or the primary-space mode if bit 134 is zero. Bit 134 is ignored during the basic PROGRAM CALL operation.

Secondary-ASN Control (S): Bit 135, when one, specifies that bits 80-95 are to become the new secondary ASN, and the new SASCE is to be set equal to the new PASCE, as part of the stacking PROGRAM CALL with-space-switching (PC-ss) operation. When this bit is zero, the new SASN and SASCE are set equal to the PASN and PASCE, respectively, of the calling program. When the ASN-and-LX-reuse facility is installed and enabled, bit 135 similarly specifies, when one, that the new SASTEIN is to be set equal to the new PASTEIN or, when zero, that the new SASTEIN is to be set equal to the PASTEIN of the calling program. Bit 135 is ignored during the basic PROGRAM CALL operation and the stacking PROGRAM CALL to-current-primary (PC-cp) operation.

Entry Key (EK): Bits 136-139 replace the PSW key in the PSW as part of the stacking PROGRAM CALL operation if the PSW-key control, bit 131, is one. Bits 136-139 are ignored, and the current PSW key remains unchanged, if bit 131 is zero. Bits 136-139 are ignored during the basic PROGRAM CALL operation.

Entry Extended Authorization Index: Bits 144-159 replace the current extended authorization index, bits 32-47 of control register 8, as part of the stacking PROGRAM CALL operation if the extended-authorization-index control, bit 133, is one. Bits 144-159 are ignored, and the current extended authorization index remains unchanged, if bit 133 is zero. Bits 144-159 are ignored during the basic PROGRAM CALL operation.

Bits 130 and 140-143 are reserved for possible future extensions and should be zeros.

Linkage-Stack Operations

A linkage stack may be formed by the control program for each dispatchable unit. The linkage stack is used to save the execution state and the contents of the general registers and access registers during the BRANCH AND STACK and stacking PROGRAM CALL operations. The linkage stack is also used to restore a portion of the execution state and general-register and access-register contents during the PROGRAM RETURN operation.

A linkage stack resides in virtual storage. The linkage stack for a dispatchable unit is in the home address space for that dispatchable unit. The home address space is designated by the home address-space-control element in control register 13.

The linkage stack is intended to be protected from problem-state programs so that these programs cannot examine or modify the information saved in the linkage stack, except by means of the EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE instructions. This protection can be obtained by means of key-controlled protection.

A linkage stack may consist of a number of linkage-stack sections chained together. A linkage-stack section is variable in length. The maximum length of each linkage-stack section is 65,560 bytes.

There are three types of entry in the linkage stack: header entry, trailer entry, and state entry. A header entry and a trailer entry are at the beginning and end, respectively, of a linkage-stack section, and they are used to chain linkage-stack sections together. Header entries and trailer entries are formed by the control program. A state entry is used to contain the

execution state and register contents that are saved during the BRANCH AND STACK or stacking PROGRAM CALL operation, and it is formed during the operation. A state entry is further distinguished as being a branch state entry if it was formed by BRANCH AND STACK or as being a program-call state entry if it was formed by PROGRAM CALL.

The actions of forming a state entry and saving information in it during the BRANCH AND STACK and stacking PROGRAM CALL operations are called the stacking process. The actions of restoring information from a state entry and logically deleting the entry during the PROGRAM RETURN operation are called the unstacking process. The part of the unstacking process that locates a state entry is also performed during the EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE operations.

Each type of linkage-stack entry has a length that is a multiple of eight bytes. A header entry and trailer entry each has a length of 16 bytes. A state entry has a length of 296 bytes.

Each of the header entry, trailer entry, and state entry has a common eight-byte area at its end, called the entry descriptor. The linkage-stack-entry address in control register 15 designates the leftmost byte of the entry descriptor of the last linkage-stack entry, other than the trailer entry, in a linkage-stack section. This entry is called the current linkage-stack entry, and the section is called the current linkage-stack section.

Each entry descriptor in a linkage-stack section, except the one in the trailer entry of the section, includes a field that specifies the amount of space existing between the end of the entry descriptor and the beginning of the trailer entry. This field is named the remaining-free-space field. The remaining-free-space field in a trailer entry is unused.

When a new state entry is to be formed in the linkage stack during the stacking process, the new entry is placed immediately after the entry descriptor of the current linkage-stack entry, provided that there is enough remaining free space in the current linkage-stack section to contain the new entry. If there is not enough remaining free space in the current section, and if the trailer entry in the current section indicates that another section follows the current section, the new entry is placed immediately after the entry descriptor of the header entry of that following section, provided that there is enough remaining free space in that section. If the trailer entry indicates that there is not a following section, an exception is recognized, and a program interruption occurs. It is then the responsibility of the control program to allocate another section, chain it to the current section, and cause the BRANCH AND STACK or stacking PROGRAM CALL instruction to be reexecuted. If there is a following section but there is not enough remaining free space in it, an exception is recognized.

If the remaining-free-space value that is used to locate a trailer entry is not a multiple of 8, an exception is recognized. The remaining-free-space value in the header entry of a linkage-stack section must be set to a multiple of 8 to ensure that the remaining-free-space value that may be used to locate the trailer entry of the section will be a multiple of 8.

When the stacking process is successful in forming a new state entry, it updates the linkage-stack-entry address in control register 15 so that the address designates the leftmost byte of the entry descriptor of the new entry, which thus becomes the new current linkage-stack entry.

When, during the unstacking process in PROGRAM RETURN, the current linkage-stack entry is a state entry, the process operates on that entry and then updates the linkage-stack-entry address so that it designates the entry descriptor of the preceding entry in the same linkage-stack section. The preceding entry thus becomes the current entry. The new current entry may be another state entry, or it may be a header entry.

The header entry of a linkage-stack section indicates whether there is a preceding section. If there is a preceding section, the header entry contains the address of the last linkage-stack entry, other than the trailer entry, in the preceding section. That last entry should be a state entry (not another header entry), unless there is an error in the linkage stack.

If the unstacking process is performed when the current linkage-stack entry is a header entry, and if the header entry indicates that a preceding linkage-stack section exists, the unstacking process proceeds by treating the entry designated in the preceding section as if it were the current entry, provided that this entry is a state entry. If the header entry does not indicate a preceding section, or if the entry designated in the preceding section is not a state entry, an exception is recognized.

When the unstacking process is performed in EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, or MODIFY STACKED STATE, the process locates a state entry but does not change the linkage-stack-entry address in control register 15.

Each entry descriptor in a linkage-stack section includes a field that specifies the length of the next linkage-stack entry, other than the trailer entry, in the section. When a state entry is created during the stacking process, zeros are placed in this field in the created entry, and the length of the state entry is placed in this field in the preceding entry. When a state entry is logically deleted during the unstacking process in PROGRAM RETURN, zeros are placed in this field in the preceding entry. This field is named the next-entry-size field.

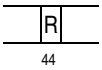
When the stacking or unstacking process operates on the linkage stack, key-controlled protection does not apply, but low-address and DAT protection do apply.

Linkage-Stack-Operations Control

The use of the linkage stack is controlled by the ASN-and-LX-reuse control in control register 0, the home address-space-control element in control register 13 and the linkage-stack-entry address in control register 15. The home address-space-control element is described in “Dynamic Address Translation” on page 3-38.

The ASN-and-LX-reuse control and the linkage-stack-entry address are

Control Register 0



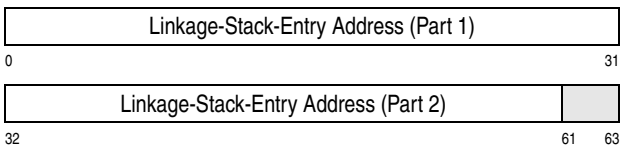
ASN-and-LX-Reuse Control (R): Bit 44 of control register 0 is the ASN-and-LX-reuse-control bit (R) and is assigned if the ASN-and-LX-reuse facility is installed. The effects of a one value of the bit specifically on the linkage stack are as follows:

- BRANCH AND STACK and stacking PROGRAM CALL save the secondary ASN-second-table-entry instance number (ASTEIN), bits 0-31 of control register 3, and the primary ASTEIN, bits 0-31 of control register 4, in bytes 176-179 and 180-183, respectively, of the linkage-stack state entry that either of the instructions forms.
- PROGRAM RETURN with space switching, after it has used the PASN in bytes 134 and 135 of the state entry to locate an ASN-second-table entry, compares the primary ASTEIN in bytes 180-183 of the state entry to the ASTEIN in the ASN-second-table entry. PROGRAM RETURN to current primary or with space switching, if it uses the SASN in bytes 130 and 131 of the state entry to locate an ASN-second-table entry (because the new SASN is not equal to the new PASN), compares the secondary ASTEIN in bytes 176-179 of the state entry to the ASTEIN in the ASN-second-table entry. The one or two comparisons of ASTEINs must each give equal results; otherwise, an ASTE-instance exception is recognized. The comparisons occur regardless of the reusable-ASN bits in the ASN-second-table entries. PROGRAM RETURN to current primary or with space switching restores the secondary and primary ASTEINs saved in the state entry to bit positions 0-31 of control registers 3 and 4, respectively (if no exception is recognized). When unstacking a state entry formed by BRANCH AND STACK, PROGRAM RETURN does not compare or restore ASTEINs.

The effect of the ASN-and-LX-reuse control on PC-number translation is described in “ASN-and-LX-Reuse Control (R):” on page 5-34. Other effects of the control are described in “ASN-Second-Table-Entry Instance Number and ASN Reuse” on page 3-25.

Control Register 15

The location of the entry descriptor of the current linkage-stack entry is specified in control register 15. The register has the following format:



Linkage-Stack-Entry Address: Bits 0-60 of control register 15, with three zeros appended on the right, form the 64-bit home virtual address of the

entry descriptor of the current linkage-stack entry in the current linkage-stack section. Bits 0-60 are changed during the stacking process in BRANCH AND STACK and stacking PROGRAM CALL and during the unstacking process in PROGRAM RETURN. Bits 61-63 of control register 15 are set to zeros when bits 0-60 are changed.

Linkage Stack

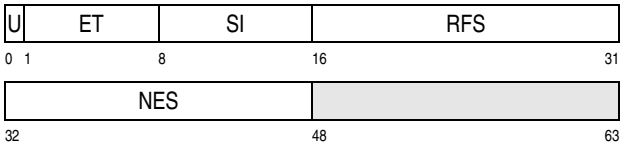
The linkage stack consists of one or more linkage-stack sections containing linkage-stack entries. There are three principal types of linkage-stack entry: header entry, trailer entry, and state entry. A state entry is further distinguished as being either a branch state entry or a program-call state entry.

Each type of linkage-stack entry has an entry descriptor at its end. The leftmost byte of the entry descriptor of the current linkage-stack entry in the current linkage-stack section is designated by the linkage-stack-entry address in control register 15.

The linkage stack resides in the home address space, designated by the home address-space-control element in control register 13.

Entry Descriptors

An entry descriptor is at the end of each linkage-stack entry. The entry descriptor is eight bytes in length and has the following format:



The fields in the entry descriptor are allocated as follows:

Unstack-Suppression Bit (U): When bit 0 is one in the entry descriptor of a header entry or state entry encountered during the unstacking process in PROGRAM RETURN, a stack-operation exception is recognized. Bit 0 is ignored in a trailer entry and during the unstacking process in EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE. The control program can temporarily set bit 0 to one in the current linkage-stack entry (a header entry or state entry) to prevent PROGRAM RETURN from being executed successfully while still allowing EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE to be executed successfully. Bit 0 is set to zero in the entry descriptor of a state entry when the entry is formed during the stacking process.

Entry Type (ET): Bits 1-7 are a code that specifies the type of the linkage-stack entry containing the entry descriptor. The assigned codes are:

Code (in Binary)	Entry Type
0001001	Header entry
0001010	Trailer entry
0001100	Branch State entry
0001101	Program-call state entry

Codes 0000000-0001000, 0001011, and 0001110 through 0111111 binary are reserved for possible future assignments. Codes 1000000 through 1111111 binary are available for use by programming.

Bits 1-7 are set to 0001100 or 0001101 binary in the entry descriptor of a state entry when the entry is formed during the stacking process.

A stack-type exception is recognized during the unstacking process in EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, MODIFY STACKED STATE, or PROGRAM RETURN if bits 1-7 in the current linkage-stack entry do not indicate that the entry is a state entry or a header entry; or, when the current entry is a header entry, if bits 1-7 in the entry designated by the backward stack-entry address in the header entry do not indicate that the designated entry is a state entry. However, a stack-specification exception is recognized, instead of a stack-type exception, if both the current entry and the designated entry are header entries.

Section Identification (SI): Bits 8-15 are an identification, provided by the control program, of the linkage-stack section containing the entry descriptor. In the state entry formed by a stacking process, the process sets bits 8-15 equal to the contents of the section-identification field in the preceding linkage-stack entry.

Remaining Free Space (RFS): Bits 16-31 specify the number of bytes between the end of this entry descriptor and the beginning of the trailer entry in the same linkage-stack section, except that this field in a trailer entry has no meaning. Thus, in the last state

entry in a section, or in the header entry if there is no state entry, bits 16-31 specify the number of bytes available in the section for performance of the stacking process. In the state entry formed by a stacking process, the process sets bits 16-31 equal to the contents of the remaining-free-space field in the preceding linkage-stack entry minus the size, in bytes, of the new entry. Bits 16-31 must be a multiple of 8 (bits 29-31 must be zeros) in the entry descriptor of the header entry in a linkage-stack section; otherwise, a value that is not a multiple of 8 will be propagated to bits 16-31 in the entry descriptor of each state entry in the section, and a stack-specification exception will be recognized if the stacking process attempts to locate the trailer entry in the section in order to proceed to the next section.

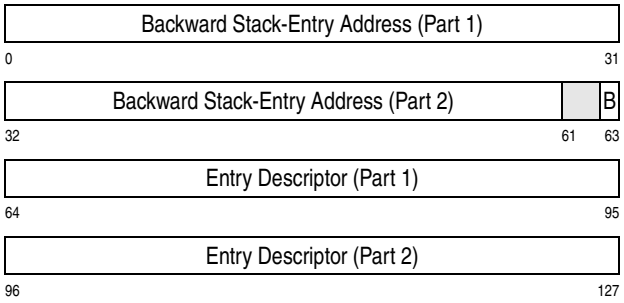
Next-Entry Size (NES): Bits 32-47 specify the size in bytes of the next linkage-stack entry, other than a trailer entry, in the same linkage-stack section. This field in the current linkage-stack entry contains all zeros. This field in a trailer entry has no meaning. When the stacking process forms a state entry, it places zeros in the next-entry-size field of the new entry, and it places the size of the new entry in the next-entry-size field of the preceding entry. When the unstacking process logically deletes a state entry, it places zeros in the next-entry-size field of the preceding entry, which entry becomes the current entry.

Bits 48-63 are set to zeros in a state entry when the entry is formed during the stacking process. In a header entry, trailer entry, or state entry, bits 48-63 are reserved for possible future extensions and should always be zeros.

Programming Note: No entry-type code will be assigned in which the leftmost bit of the code is one. The control program can temporarily set the leftmost bit to one in the entry-type code of the current linkage-stack entry (a header entry or a state entry) to prevent the successful execution of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, MODIFY STACKED STATE, or PROGRAM RETURN.

Header Entries

A header entry is at the beginning of each linkage-stack section. The header entry is 16 bytes in length and has the following format:



The fields in the first eight bytes of the header entry are allocated as follows:

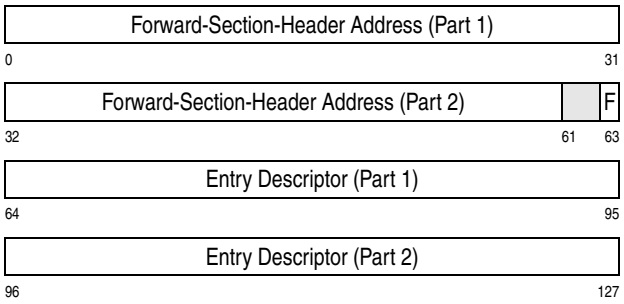
Backward Stack-Entry Validity Bit (B): Bit 63 when one, specifies that the preceding linkage-stack section is available and that the backward stack-entry address, bits 0-60 is valid. Bit 63 is set to one during the stacking process when the process proceeds to this section from the preceding one because there is not enough space available in the preceding section to perform the process. During the unstacking process when this header entry is the current linkage-stack entry, a stack-empty exception is recognized if bit 63 is zero.

Backward Stack-Entry Address (BSEA): When bit 63 is one, bits 0-60 with three zeros appended on the right, form the 64-bit home virtual address of the entry descriptor of the last linkage-stack entry, other than the trailer entry, in the preceding linkage-stack section. However, if the current linkage-stack entry is in the preceding or an earlier linkage-stack section, bits 0-60 may have no meaning because the entry they designate, and earlier entries, may have been logically deleted. Bits 0-60 are set during the stacking process when the process proceeds to this section from the preceding one because there is not enough space available in the preceding section to perform the process. During the unstacking process when this header entry is the current linkage-stack entry and bit 63 is one, the entry designated by bits 0-60 is treated as the current entry.

Bits 61 and 62 are set to zeros when bits 0-60 are set during the stacking process. Bits 61 and 62 are reserved for possible future extensions.

Trailer Entries

A trailer entry is at the end of each linkage-stack section. The trailer entry begins immediately after the area specified by the remaining-free-space field in the entry descriptors of the header entry and each state entry in the same linkage-stack section. The trailer entry is 16 bytes in length and has the following format:



The fields in the first eight bytes of the trailer entry are allocated as follows:

Forward-Section Validity Bit (F): Bit 63 when one, specifies that the next linkage-stack section is available and that the forward-section-header address, bits is valid. During the stacking process when there is not enough space available in the current linkage-stack section to perform the process, a stack-full exception is recognized if bit 63 in the trailer entry of the current section is zero.

Forward-Section-Header Address (FSHA):

When bit 63 is one, bits 0-60, with three zeros appended on the right, form the 64-bit home virtual address of the entry descriptor of the header entry in the next linkage-stack section. During the stacking process when there is not enough space available in the current section to perform the process and bit 63 is one, the header entry designated by bits 0-60 becomes the current linkage-stack entry.

Bits 61 and 62 are reserved for possible future extensions.

Programming Note: All of the fields in the trailer entry are set only by the control program.

State Entries

Zero, one, or more state entries may follow the header entry in each linkage-stack section. A state entry may be a branch state entry, formed by a BRANCH AND STACK instruction, or a program-call state entry, formed by a stacking PROGRAM CALL instruction. The state entry is 296 bytes in length and has the following format:

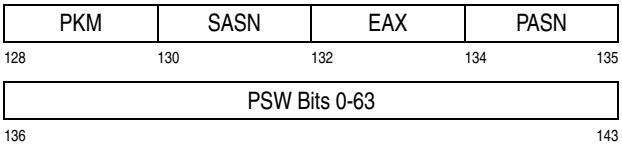
Hex	Dec	Contents	Bytes
0 8 ⋮ 70 78	0 8 ⋮ 112 120	Contents of General Registers 0 - 15	128
80 88 ⋮ D0 D8	128 136 ⋮ 208 216	Other Status Information	96
E0 E8 ⋮ 110 118	224 232 ⋮ 272 280	Contents of Access Registers 0 - 15	64
120	288	Entry Descriptor	8

Bytes 0-127 of the state entry contain the contents of general registers 0-15 in the ascending order of the register numbers. Bytes 224-287 contain the contents of access registers 0-15 in the ascending order of the register numbers. The contents of these fields are moved from the registers to the state entry during the BRANCH AND STACK and stacking PROGRAM CALL operations. The contents of general registers 2-14 and access registers 2-14 are restored from the state entry to the registers during the PROGRAM RETURN operation. The contents of a specified range of general registers and access registers can be restored from the state entry to the registers by EXTRACT STACKED REGISTERS.

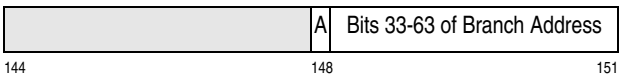
Bytes 128-223 of the state entry contain the other status information that is placed in the entry by BRANCH AND STACK, stacking PROGRAM CALL,

and MODIFY STACKED STATE. A portion of this status information is restored to the PSW and control registers by PROGRAM RETURN, and all of the information can be examined by means of EXTRACT STACKED STATE. Bytes 288-295 contain the entry descriptor. EXTRACT STACKED STATE sets the condition code to indicate whether the entry-type code in the entry descriptor specifies a branch state entry or a program-call state entry.

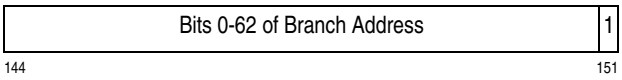
Bytes 128-223 of the state entry have the following detailed format:



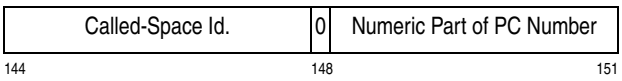
In a Branch State Entry Made in 24-Bit or 31-Bit Mode



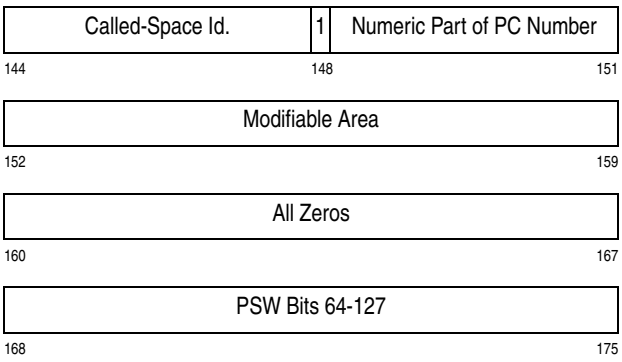
In a Branch State Entry Made in 64-Bit Mode



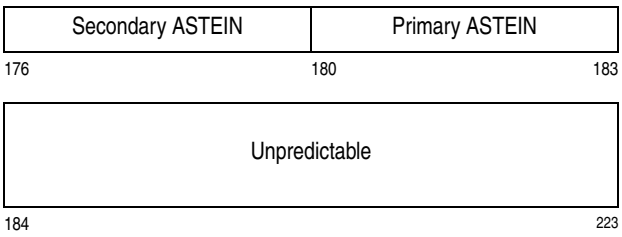
In a Program-Call State Entry Made When Resulting Mode Is 24 Bit or 31 Bit



In a Program-Call State Entry Made When Resulting Mode Is 64 Bit



If ASN-and-LX Reuse Is Enabled; otherwise Unpredictable



The fields in bytes 128-183 are allocated as follows. In the following,"of the calling program" means the value existing at the beginning of the execution of the BRANCH AND STACK or stacking PROGRAM CALL instruction that formed the state entry.

PSW-Key Mask (PKM): Bytes 128-129 contain the PSW-key mask, bits 32-47 of control register 3, of the calling program. The PSW-key mask is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL, and it is restored to the control register by a PROGRAM RETURN instruction that unstacks an entry formed by stacking PROGRAM CALL.

Secondary ASN (SASN): Bytes 130-131 contain the secondary ASN, bits 48-63 of control register 3, of the calling program. The SASN is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL, and it is restored to the control register by a PROGRAM RETURN instruction that unstacks an entry formed by stacking PROGRAM CALL.

Extended Authorization Index (EAX): Bytes 132-133 contain the extended authorization index, bits 32-47 of control register 8, of the calling program. The EAX is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL, and it is restored to the control register by a PROGRAM RETURN instruction that unstacks an entry formed by stacking PROGRAM CALL.

Primary ASN (PASN): Bytes 134-135 contain the primary ASN, bits 48-63 of control register 4, of the calling program. The PASN is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL, and it is restored to the control register by a PROGRAM RETURN instruction that unstacks an entry formed by stacking PROGRAM CALL.

Program-Status Word (PSW): In a branch state entry formed by a BRANCH AND STACK instruction in which the R_1 field is zero, and in a program-call state entry, bytes 136-143 and 168-175 contain the updated PSW of the calling program. Bytes 136-143 contain bits 0-63 of the PSW, and bytes 168-175 contain bits 64-127 of the PSW. Thus, the basic and extended addressing-mode bits in this PSW specify the addressing mode of the calling program, and the instruction address designates the next sequential instruction following the BRANCH AND STACK or stacking PROGRAM CALL instruction that formed the state entry, or following an execute-type instruction that had the BRANCH AND STACK or stacking PROGRAM CALL instruction as its target instruction. In a branch state entry formed by a BRANCH AND STACK instruction in which the R_1. field is nonzero, bytes 136-143 and 168-175 contain the PSW of the calling program, except that the extended-address-

ing-mode bit in bit position 31 of bytes 136-139, the basic-addressing-mode bit in bit position 0 of byte 140, and the instruction address in bytes 168-175 are as specified by the contents of the general register designated by the R_1 field. See the definition of BRANCH AND STACK in Chapter 10, "Control Instructions" for how the basic- and extended-addressing-mode bits and instruction address are specified. The value of the PER mask in bytes 136-143 is always unpredictable. The PSW is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL and is restored as the current PSW by PROGRAM RETURN, except that the PER mask is not restored. PROGRAM RETURN does not change the PER mask in the current PSW.

Basic Addressing Mode (A): In a branch state entry made in the 24-bit or 31-bit addressing mode, bit position 0 of bytes 148-151 contains the basic-addressing-mode bit, bit 32 of the PSW, at the end of the execution of the BRANCH AND STACK instruction that formed the state entry. The basic-addressing-mode bit is saved in bit position 0 of bytes 148-151 by BRANCH AND STACK. BRANCH AND STACK does not change the basic-addressing-mode bit in the PSW.

Branch Address: In a branch state entry made in the 24-bit or 31-bit addressing mode, bit positions 1-31 of bytes 148-151 contain bits 33-63 of the instruction address in the PSW at the end of the execution of the BRANCH AND STACK instruction that formed the state entry, and the contents of bytes 144-147 are unpredictable. In a branch state entry made in the 64-bit addressing mode, bytes 144-151 contain bits 0-62 of that instruction address with a one appended on the right. The instruction address is saved in bytes 148-151 or 144-151 (depending on the addressing mode) by BRANCH AND STACK. When the R_2. field of BRANCH AND STACK is nonzero, the instruction causes branching, and the instruction address in bytes 148-151 or 144-151 is the branch address. When the R_2 field of BRANCH AND STACK is zero, the instruction is executed without branching, and the instruction address in bytes 148-151 or 144-151 is the address of the next sequential instruction following the BRANCH AND STACK instruction, or following an execute-type instruction that had the BRANCH AND STACK instruction as its target instruction.

Called-Space Identification: In a program-call state entry, bytes 144-147 contain the called-space identification (CSI). The CSI is saved in the state entry by stacking PROGRAM CALL.

If the PROGRAM CALL operation was space switching, bytes 0 and 1 of the CSI (bytes 144 and 145 of the state entry) contain the new primary ASN that was placed in control register 4 by the PROGRAM CALL instruction. When the ASN-and-LX-reuse facility is not enabled, bytes 2 and 3 of the CSI (bytes 146 and 147 of the state entry) contain the rightmost two bytes of the ASTE sequence number (ASTESN) in the new primary ASTE whose address was placed in control register 5 by the PROGRAM CALL instruction. When the ASN-and-LX-reuse facility is installed and enabled, bytes 2 and 3 of the CSI (bytes 146 and 147 of the state entry) contain the rightmost two bytes of the ASTE instance number (ASTEIN) in the new primary ASTE whose address was placed in control register 5 by the PROGRAM CALL instruction.

If the PROGRAM CALL operation was the to-current-primary operation, the CSI is all zeros.

Numeric Part of PC Number: In a program-call state entry, bit positions 1-31 of bytes 148-151 contain the numeric part of the PC number used by the stacking PROGRAM CALL instruction that formed the entry. When ASN-and-LX reuse is not enabled, or when it is and bit 44 of the effective address used by stacking PROGRAM CALL is zero, stacking PROGRAM CALL places bits 44-63 of the effective address, with 11 zeros appended on the left, in bit positions 1-31 of bytes 148-151. When ASN-and-LX reuse is enabled and bit 44 of the effective address is one, stacking PROGRAM CALL places bits 45-63 of the effective address, with bits 32-43 of the effective address appended on the left, in bit positions 1-31 of bytes 148-151. In any case, stacking PROGRAM CALL places a zero in bit position 0 of the bytes if the resulting addressing mode is the 24-bit or 31-bit mode or a one in bit position 0 if the resulting addressing mode is the 64-bit mode.

Modifiable Area: Bytes 152-159 are the field that is set by MODIFY STACKED STATE. BRANCH AND STACK and stacking PROGRAM CALL place all zeros in bytes 152-159.

Secondary ASTEIN (SASTEIN): If the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control in control register 0, bytes 176-179 contain the secondary ASTEIN, bits 0-31 of control register 3, of the calling

program. The SASTEIN is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL. When PROGRAM RETURN unstacks a program-call state entry, it compares the SASTEIN in the state entry to the ASTEIN in the ASN-second-table entry if SASN translation occurs, and it always restores the SASTEIN to control register 3.

Primary ASTEIN (PASTEIN): If the ASN-and-LX-reuse facility is installed and is enabled by the ASN-and-LX-reuse control in control register 0, bytes 180-183 contain the primary ASTEIN, bits 0-31 of control register 4, of the calling program. The PASTEIN is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL. When PROGRAM RETURN unstacks a program-call state entry, it compares the PASTEIN in the state entry to the ASTEIN in the ASN-second-table entry if PASN translation occurs, and it always restores the PASTEIN to control register 4.

All zeros are placed in bytes 160-167 by BRANCH AND STACK and stacking PROGRAM CALL.

The contents of bytes 184-223 are unpredictable.

Stacking Process

The stacking process is performed as part of a BRANCH AND STACK or stacking PROGRAM CALL operation. The process locates space for a new linkage-stack state entry, forms the entry, updates the next-entry-size field in the preceding entry, and updates the linkage-stack-entry address in control register 15 so that the new entry becomes the current linkage-stack entry.

For the stacking process to be performed successfully, DAT must be on and the CPU must be in the primary-space mode or access-register mode; otherwise, a special-operation exception is recognized, and the operation is suppressed.

Except as just mentioned, the stacking process is performed independent of the current addressing mode and translation mode, as specified by bits 31, 32, 16, and 17 of the current PSW. All addresses used during the stacking process are always 64-bit home virtual addresses.

During the stacking process when any address is formed through the addition or subtraction of a value to or from another address, a carry out of, or a borrow into, bit position 0 of the address, if any, is ignored.

When the stacking process fetches or stores by using an address that designates, after translation, a location that is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Key-controlled protection does not apply to the accesses made during the stacking process, but DAT protection and low-address protection do apply. A protection exception causes the operation to be suppressed.

Locating Space for a New Entry

The linkage-stack-entry address in control register 15 is used to locate the current linkage-stack entry. Bits 0-60 of control register 15, with three zeros appended on the right, form the 64-bit home virtual address of the leftmost byte of the entry descriptor of the current linkage-stack entry.

The first word of the entry descriptor of the current linkage-stack entry is fetched by using the 64-bit home virtual address. This fetch is for the purpose of obtaining the section-identification and remaining-free-space fields in the word; the unstack-suppression bit and entry-type field in the word are not examined.

The 16-bit unsigned binary value in the remaining-free-space field, bits 16-31 of the entry descriptor, is compared against the size in bytes of the linkage-stack entry to be formed. The size of a state entry is 296 bytes. If the value in the field is equal to or greater than the size of the entry to be formed, processing continues as described in “Forming the New Entry” on page 5-85; otherwise, processing continues as described below.

When the remaining-free-space field in the current linkage-stack entry indicates that there is not enough space available in the current linkage-stack section to form the new entry, the first doubleword of the trailer entry of the current section is fetched. The address for fetching this doubleword is determined as follows: to the address formed from the contents of control register 15, add 8 to address the first byte after the entry descriptor of the current entry, and then add the contents of the remaining-free-space field of the current entry to address the first byte of the trailer entry. The remaining-free-space value used in the addition

must be a multiple of 8; otherwise, a stack-specification exception is recognized, and the operation is nullified.

If the forward-section-validity bit, bit 63 of the trailer entry is zero, a stack-full exception is recognized, and the operation is nullified; otherwise, the forward-section-header address in the trailer entry is used to locate the header entry in the next linkage-stack section. Bits 0-60 of the trailer entry, with three zeros appended on the right, form the 64-bit home virtual address of the leftmost byte of the entry descriptor of the header entry in the next section.

The first word of the entry descriptor of the header entry in the next linkage-stack section is fetched. This fetch is for the purpose of obtaining the section-identification and remaining-free-space fields in the word; the unstack-suppression bit and entry-type field in the word are not examined.

The value in the remaining-free-space field of the header entry in the next linkage-stack section is compared against the size in bytes of the entry to be formed. If the value in the field is equal to or greater than the size of the entry to be formed, the following occurs:

- The linkage-stack-entry address, bits 0-60 of control register 15, is placed, as the backward stack-entry address, in bit positions 0-60 of the header entry in the next linkage-stack section, and zeros are placed in bit positions 61 and 62.
- The backward stack-entry validity bit, bit 63, in the header entry in the next section is set to one.
- Bits 0-60 of the 64-bit home virtual address of the entry descriptor of the header entry in the next section are placed in bit positions 0-60 of control register 15, and zeros are placed in bit positions 61-63 of control register 15. Thus, the header entry in the next section becomes the current linkage-stack entry, and the next section becomes the current linkage-stack section.
- Processing continues as described in “Forming the New Entry”.

If the value in the remaining-free-space field of the header entry in the next section (before the next section becomes the current section) is less than the size of the linkage-stack entry to be formed, a stack-specification exception is recognized, and the operation is nullified.

Forming the New Entry

When the remaining-free-space field in the current linkage-stack entry indicates that there is enough space available in the current linkage-stack section to form the new entry, the new entry is formed beginning immediately after the entry descriptor of the current entry.

The new entry is a state entry. The contents of general registers 0-15 are stored in bytes 0-127 of the new entry, in the ascending order of the register numbers. The contents of access registers 0-15 are stored in bytes 224-287 of the new entry, in the ascending order of the register numbers. The PSW-key mask, bits 32-47 of control register 3; secondary ASN, bits 48-63 of control register 3; extended authorization index, bits 32-47 of control register 8; and primary ASN, bits 48-63 of control register 4, are stored in bytes 128-129, 130-131, 132-133, and 134-135, respectively, of the new entry. The current PSW, in which the instruction address has been updated, is stored in bytes 136-143 and 168-175 of the new entry. Bytes 136-143 contain bits 0-63 of the PSW, and bytes 168-175 contain bits 64-127 of the PSW. However, the value of the PER mask, bit 1 in the PSW stored, is unpredictable. Also, if the instruction being executed is a BRANCH AND STACK instruction in which the R_1. field is nonzero, the extended- and basic-addressing-mode bits stored in bytes 139 and 140, respectively, of the new entry, and the instruction address stored in bytes 168-175 of the new entry, are as specified by the contents of the general register designated by the R_1 field.

If the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control, bit 44 of control register 0, the secondary ASTEIN, bits 0-31 of control register 3, and the primary ASTEIN, bits 0-31 of control register 4, are stored in bytes 176-179 and 180-183, respectively, of the new entry.

When the instruction is PROGRAM CALL, the called-space identification is stored in bytes 144-147 of the new entry. When the instruction is performing the space-switching PROGRAM CALL operation and the ASN-and-LX-reuse facility is not enabled, the called-space identification is the two-byte ASN, bytes 10 and 11 in the entry-table entry used by the instruc-

tion, followed by bytes 2 and 3 of the ASTE sequence number, bytes 2 and 3 being bits 176-191, in the ASN-second-table entry specified by the ASN. When the instruction is performing the space-switching PROGRAM CALL operation and the ASN-and-LX-reuse facility is installed and enabled, the called-space identification is the two-byte ASN, bytes 10 and 11 in the entry-table entry used by the instruction, followed by bytes 2 and 3 of the ASTE instance number, bytes 2 and 3 being bits 368-383, in the ASN-second-table entry specified by the ASN. When the instruction is performing the to-current-primary PROGRAM CALL operation, the called-space identification is all zeros.

When the instruction is BRANCH AND STACK in the 24-bit or 31-bit addressing mode, the basic-addressing-mode bit from the current PSW is stored in bit position 0 of byte 148 in the state entry, bits 33-63 of the branch address, or of the updated instruction address if the operation is performed without branching, are stored in bit positions 1-31 of bytes 148-151, and the contents of bytes 144-147 are unpredictable. In the 64-bit addressing mode, bits 0-62 of the branch address or updated instruction address, with a one appended on the right, are stored in bytes 144-151 of the state entry.

When the instruction is PROGRAM CALL, the numeric part of the PC number used, with 11 zeros appended on the left if the number is 20 bits, is stored in bit positions 1-31 of bytes 148-151. If the resulting addressing mode after the execution of PROGRAM CALL is the 24-bit or 31-bit addressing mode, a zero is stored in bit position 0 of byte 148. If the resulting addressing mode is the 64-bit addressing mode, a one instead of a zero is stored in bit position 0 of byte 148.

Zeros are stored in bytes 152-167 of the new entry. The contents of bytes 176-223 are unpredictable, or the contents of bytes 184-223 are unpredictable if the secondary ASTEIN and primary ASTEIN were stored.

Bytes 288-295 of the new entry are its entry descriptor. The unstack-suppression bit, bit 0, of this entry descriptor is set to zero. The code 0001100 binary is stored in the entry-type field, bits 1-7, of this entry descriptor if the instruction being executed is BRANCH AND STACK. The code 0001101 binary is stored if the instruction is PROGRAM CALL. The value in the section-identification field of the current linkage-stack entry is stored in the section-identification field, bits 8-15, of this entry descriptor. The value in the remaining-free-space field of the current entry, minus the size in bytes of the new entry, is stored in the remaining-free-space field of this entry descriptor. Zeros are stored in the next-entry-size field, bits 32-47, and in bit positions 48-63 of this entry descriptor.

The stores into the new entry appear to be word concurrent as observed by other CPUs. The order in which the stores occur is unpredictable.

Updating the Current Entry

The size in bytes of the new linkage-stack entry is stored in the next-entry-size field of the current entry. The remainder of the current entry remains unchanged.

The order of the stores into the current entry and the new entry is unpredictable.

Updating Control Register 15

Bits 0-60 of the 64-bit home virtual address of the entry descriptor of the new linkage-stack entry are placed in bit positions 0-60 of control register 15, the linkage-stack-entry address. Zeros are placed in bit positions 61-63 of control register 15. Thus, the new entry becomes the current linkage-stack-entry.

Recognition of Exceptions during the Stacking Process

The exceptions which can be encountered during the stacking process and their priority are described in the definitions of the BRANCH AND STACK and PROGRAM CALL instructions.

Programming Note: Any exception recognized during the execution of BRANCH AND STACK and PROGRAM CALL causes either nullification or suppression. Therefore, if an exception is recognized, the stacking process does not store into any linkage-stack entry or change the contents of control register 15.

Unstacking Process

The unstacking process is performed as part of the PROGRAM RETURN operation. The process locates the last state entry in the linkage stack, restores a portion of the information in the entry to the CPU registers, updates the next-entry-size field in the preceding entry, and updates the linkage-stack-entry

address in control register 15 so that the preceding entry becomes the current linkage-stack entry. The part of the unstacking process that locates the last state entry is also performed as part of the EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE operations.

For the unstacking process to be performed successfully, DAT must be on and the CPU must be in the primary-space mode or access-register mode; otherwise, a special-operation exception is recognized, and the operation is suppressed. However, when the unstacking process is performed as part of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, or MODIFY STACKED STATE, the CPU may be in the primary-space, access-register, or home-space mode.

Except as just mentioned, the unstacking process is performed independent of the current addressing mode and translation mode, as specified by bits 31, 32, 16, and 17 of the current PSW. All addresses used during the unstacking process are always 64-bit home virtual addresses.

During the unstacking process when any address is formed through the addition or subtraction of a value to or from another address, a carry out of, or a borrow into, bit position 0 of the address, if any, is ignored.

When the unstacking process fetches or stores by using an address that designates, after translation, a location that is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Key-controlled protection does not apply to the accesses made during the unstacking process, but DAT protection and low-address protection do apply. A protection exception causes the operation to be suppressed.

Locating the Current Entry and Processing a Header Entry

The linkage-stack-entry address in control register 15 is used to locate the current linkage-stack entry. Bits 0-60 of control register 15, with three zeros appended on the right, form the 64-bit home virtual address of the leftmost byte of the entry descriptor of the current linkage-stack entry.

The first word of the entry descriptor of the current linkage-stack entry is fetched by using the 64-bit home virtual address. If the entry-type code in bits 1-7 of the entry descriptor is not 0001001 binary, indicating that the entry is not a header entry, processing continues as described in “Checking for a State Entry”; otherwise, processing continues as described below.

When the entry-type code in the current linkage-stack entry is 0001001 binary, indicating a header entry, the next processing depends on which instruction is being executed. When the unstacking process is performed as part of the PROGRAM RETURN operation and the unstack-suppression bit, bit 0, in the entry descriptor of the current entry is one, a stack-operation exception is recognized, and the operation is nullified. When the unstacking process is performed as part of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, or MODIFY STACKED STATE, the unstack-suppression bit is ignored.

When there is not an exception due to the unstack-suppression bit, the first doubleword of the current linkage-stack entry (a header entry) is fetched. The address of this doubleword is determined by subtracting 8 from the address of the entry descriptor of the current entry.

If the backward stack-entry validity bit, bit 63, of the current entry is zero, a stack-empty exception is recognized, and the operation is nullified; otherwise, the backward stack-entry address in the current entry is used to locate a linkage-stack entry referred to here as the designated entry. Bits 0-60 of the current entry, with three zeros appended on the right, form the 64-bit home virtual address of the leftmost byte of the entry descriptor of the designated entry.

It is assumed in this definition of the unstacking process that the designated linkage-stack entry is the last entry, other than the trailer entry, in the preceding linkage-stack section. This assumption does not imply any processing that is not explicitly described.

The first word of the entry descriptor of the designated entry is fetched. If the entry-type code in this entry descriptor is not 0001001 binary, indicating that the entry is not a header entry, the following occurs:

- When the unstacking process is performed as part of the PROGRAM RETURN operation, bits 0-60 of the 64-bit home virtual address of the

entry descriptor of the designated entry are placed in bit positions 0-60 of control register 15, and zeros are placed in bit positions 61-63 of control register 15. Thus, the designated entry becomes the current linkage-stack entry, and the preceding section (based on the assumption) becomes the current linkage-stack section. When the unstacking process is performed as part of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, or MODIFY STACKED STATE, the contents of control register 15 remain unchanged, but the designated entry is temporarily, during the remainder of the definition of the instruction, referred to as the current linkage-stack entry.

- Processing continues as described in "Checking for a State Entry".

If the entry-type code in the designated entry is 0001001 binary, indicating a header entry, a stack-specification exception is recognized, and the operation is nullified.

Checking for a State Entry

When the entry-type code in the current linkage-stack entry indicates that the entry is not a header entry, the code is checked for being 0001100 or 0001101 binary, which are the codes assigned to a state entry.

If the current linkage-stack entry is a state entry, the next processing depends on which instruction is being executed. When the unstacking process is performed as part of the PROGRAM RETURN operation, processing continues as described in "Restoring Information" on page 5-88. When the process is performed as part of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, or MODIFY STACKED STATE, the process is completed; that is, no additional processing occurs as a part of the unstacking process.

If the current linkage-stack entry is not a state entry (and necessarily not a header entry either), a stack-type exception is recognized, and the operation is nullified.

Restoring Information

The remaining parts of the unstacking process occur only in the PROGRAM RETURN operation.

The current linkage-stack entry is a state entry. If the unstack-suppression bit in the entry is one, a stack-operation exception is recognized, and the operation is nullified.

When there is not an exception due to the unstack-suppression bit, a portion of the contents of the current linkage-stack entry are restored to the CPU registers. The contents of general registers 2-14 and access registers 2-14 are restored to those registers from where they were saved in the current entry by the stacking process. When the entry-type code in the current entry is 0001101 binary, indicating a program-call state entry, the PSW-key mask and secondary ASN in control register 3, extended authorization index in control register 8, and primary ASN in control register 4 are similarly restored. During this restoration, the authorization index in control register 4 and the monitor masks and enhanced-monitor masks in control register 8 remain unchanged. (The authorization index may be changed by the part of the PROGRAM RETURN execution that occurs after the unstacking process.)

When the state entry is a program-call state entry, and if the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control in control register 0, the secondary ASTEIN in control register 3 and the primary ASTEIN in control register 4 are restored from where they were saved in the state entry. However, the primary ASTEIN and secondary ASTEIN may be subject to certain tests, as described in the PROGRAM RETURN definition.

When the entry-type code is 0001100 binary, indicating a branch state entry, the PSW-key mask, secondary ASN, extended authorization index, primary ASN, secondary ASTEIN, and primary ASTEIN in the current entry are ignored, and all contents of the control registers remain unchanged.

When the current entry is either a branch state entry or a program-call state entry, bits 0-63 and 64-127 of the current PSW are restored from bytes 136-143 and bytes 168-175, respectively, of the entry, except that the PER mask is not restored. The PER mask in the current PSW remains unchanged. Bytes 144-159 and bytes 160-167 of the current entry are ignored.

The fetches from the current entry appear to be word concurrent as observed by other CPUs. The order in which the fetches occur is unpredictable.

Updating the Preceding Entry

Zeros are stored in the next-entry-size field, bits 32-47, of the entry descriptor of the preceding linkage-stack entry. The remainder of the preceding entry remains unchanged. The address of the entry descriptor of the preceding entry is determined by subtracting the size in bytes of the current entry from the address of the entry descriptor of the current entry.

The order of the store into the preceding entry and the fetches from the current entry is unpredictable.

Updating Control Register 15

Bits 0-60 of the 64-bit home virtual address of the entry descriptor of the preceding linkage-stack entry are placed in bit positions 0-60 of control register 15, the linkage-stack-entry address. Zeros are placed in bit positions 61-63 of control register 15. Thus, the preceding entry becomes the current linkage-stack entry.

Recognition of Exceptions during the Unstacking Process

The exceptions which can be encountered during the unstacking process and their priority are described in the definition of the PROGRAM RETURN instruction. The exceptions which apply to EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE are described in the definitions of those instructions.

Programming Notes:

1. Any exceptions recognized during the execution of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, MODIFY STACKED STATE, or PROGRAM RETURN cause either nullification or suppression. Therefore, if an exception is recognized, the unstacking process does not change the contents of any CPU register (except for updating the instruction address in the PSW in the case of suppression) or store into any linkage-stack entry.

2. The unstacking process in PROGRAM RETURN does not restore the PER mask in the PSW so that an act of turning PER on or off after the execution of the related BRANCH AND STACK or PROGRAM CALL instruction but before the execution of the PROGRAM RETURN instruction will not be counteracted. When PROGRAM CALL or PROGRAM RETURN is space switching, the space-switch event can be used as a signal to turn PER on or off, if desired.

Transactional-Execution Facility

This section describes the registers, instructions, controls, and operation of the transactional-execution facility. The transactional-execution facility provides the means by which a program can issue multiple instructions, the storage-operand accesses of which appear to occur as a single concurrent operation as observed by other CPUs and by the channel subsystem.

Three special-purpose controls (the transaction-abort PSW, the transaction diagnostic-block address, and the transaction nesting depth), five control register bits, and six general instructions are used to control the transactional-execution facility. When the facility is installed, it is installed in all CPUs in the configuration. Facility indication 73, when one, indicates that the transactional-execution facility is installed.

When the transactional-execution facility is installed, the configuration may also provide the constrained transactional-execution facility. See “Constrained Transaction” (below) for details on constrained transactions. When facility indications 50 and 73 are both one, the constrained transactional-execution facility is installed.

Note: In the following discussion of the transactional-execution facility, the instruction name TRANSACTION BEGIN refers to the instructions having the mnemonics TBEGIN and TBEGINC. Discussions pertaining to a specific instruction are indicated by the instruction name followed by the mnemonic in parentheses or brackets, or simply by the mnemonic.

Transactional-Execution Terminology

The following terms are used pervasively in the description of the transactional-execution facility.

Note: Of necessity, some of these terms are cross-referential. They appear in alphabetical order only for convenience.

Abort: A transaction is said to abort when it is ended prior to a TRANSACTION END instruction

that results in a transaction nesting depth of zero. When a transaction aborts, the following occurs:

- Transactional store accesses made by any and all levels of the transaction are discarded (that is, not committed).
- Nontransactional store accesses made by any and all levels of the transaction are committed.
- Registers designated by the general-register-save mask of the outermost TRANSACTION BEGIN instruction are restored to their contents prior to transactional execution (that is, to their contents at execution of the outermost TRANSACTION BEGIN instruction). General registers not designated by the general-register-save mask of the outermost TRANSACTION BEGIN instruction are not restored.
- Access registers, vector registers, floating-point registers, and the floating-point control register are not restored. Any changes made to these registers during transactional execution are retained when the transaction aborts.

A transaction may be aborted due to a variety of reasons, including attempted execution of a restricted instruction, attempted modification of a restricted resource, transactional conflict, exceeding various CPU resources, any interruption, a TRANSACTION ABORT instruction, and other reasons. "Transaction Abort Code (TAC)" on page 5-95 provides specific reasons why a transaction may be aborted.

Commit: At the completion of the outermost TRANSACTION END instruction, the CPU is said to commit the store accesses made by the transaction such that they are visible to other CPUs and the channel subsystem. As observed by other CPUs and by the channel subsystem, all fetch and store accesses made by all nested levels of the transaction appear to occur as a single concurrent operation when the commit occurs.

The contents of the general registers, access registers, vector registers, floating-point registers, and the floating-point control register are not modified by the commit process. Any changes made to these registers during transactional execution are retained when the transaction's stores are committed.

Conflict: A transactional access made by one CPU is said to conflict with either (a) a transactional access or nontransactional access made by another CPU, or (b) a nontransactional access made by the channel subsystem, if both accesses are to any location within the same cache line, and one or both of the accesses is a store.

A conflict may be detected by a CPU's speculative execution of instructions, even though the conflict may not be detected in the conceptual sequence.

Programming Note: Because of speculative execution by the CPU, a conflict may be detected at a storage location that would not necessarily be accessed by the conceptual execution sequence.

Constrained Transaction: A constrained transaction is a transaction that executes in the constrained transactional-execution mode (see below) and is subject to the following limitations:

- A subset of the general instructions is available.
- A limited number of instructions may be executed.
- A limited number of storage-operand locations may be accessed.
- The transaction is limited to a single nesting level.

In the absence of repeated interruptions, guarded-storage-event detection, or conflicts with other CPUs or the channel subsystem, a constrained transaction will eventually complete, thus an abort-handler routine is not required. Constrained transactions are described in detail in "Constrained Transaction" on page 5-107.

Programming Note: When the TBEGINC instruction is executed while the CPU is already in the nonconstrained transactional-execution mode, execution continues as a nested nonconstrained transaction.

Constrained Transactional-Execution Mode: When the transaction nesting depth is zero, and a transaction is initiated by the TRANSACTION BEGIN (TBEGINC) instruction, the CPU enters the constrained transactional-execution mode. While the CPU is in the constrained transactional-execution mode, the transaction nesting depth is always one.

Nested Transaction: When the TRANSACTION BEGIN instruction is issued while the CPU is in the nonconstrained transactional-execution mode, the

transaction is said to be nested. See “Transaction Nesting Depth (TND)” on page 5-93 below.

The transactional-execution facility uses a model called *flattened nesting*. In the flattened-nesting model, stores made by an inner transaction are not observable by other CPUs and by the channel subsystem until the outermost transaction commits its stores. Similarly, if a transaction aborts, all nested transactions abort, and all transactional stores of all nested transactions are discarded.

Nonconstrained Transaction: A nonconstrained transaction is a transaction that executes in the nonconstrained transactional-execution mode (see below). Although a nonconstrained transaction is not limited in the manner described in “Constrained Transaction” on page 5-107, it may still be aborted due to a variety of causes as described in “Abort” on page 5-89.

Nonconstrained Transactional-Execution Mode: When a transaction is initiated by the TRANSACTION BEGIN (TBEGIN) instruction, the CPU enters the nonconstrained transactional-execution mode. While the CPU is in the nonconstrained transactional-execution mode, the transaction nesting depth may vary from one to the maximum transaction nesting depth.

Nontransactional Access: Nontransactional accesses are storage operand accesses made by the CPU when it is not in the transactional-execution mode (that is, classic storage accesses outside of a transaction). Additionally, the NONTRANSACTIONAL STORE instruction may be used to cause a nontransactional store access while the CPU is in the nonconstrained transactional-execution mode.

Accesses made by the channel subsystem are nontransactional accesses.

Outer / Outermost Transaction: A transaction with a lower-numbered transaction nesting depth is said to be an outer transaction. A transaction with a transaction-nesting-depth value of one is said to be the outermost transaction.

An outermost TRANSACTION BEGIN instruction is one that is executed when the transaction nesting depth is initially zero. An outermost TRANSACTION END instruction is one that causes the transaction nesting depth to transition from one to zero. A constrained transaction is always the outermost transaction.

Program-Interruption Filtering: When a transaction is aborted due to certain program-interruption conditions, the program can optionally prevent the interruption from occurring. This technique is called program-interruption filtering. Program-interruption filtering is subject to the transactional class of the interruption, the effective program-interruption-filtering control from the TRANSACTION BEGIN instruction, and the transactional-execution program-interruption-filtering override in control register 0.

Transaction: A transaction comprises the storage-operand accesses made, and selected general registers altered, while the CPU is in the transactional-execution mode. For a nonconstrained transaction, storage-operand accesses may include both transactional accesses and nontransactional accesses. For a constrained transaction, storage-operand accesses are limited to transactional accesses. As observed by other CPUs and by the channel subsystem, all storage-operand accesses made by the CPU while in the transactional-execution mode appear to occur as a single concurrent operation. If a transaction is aborted, transactional store accesses are discarded, and any registers designated by the general-register-save mask of the outermost TRANSACTION BEGIN instruction are restored to their contents prior to transactional execution.

Transactional Accesses: Transactional accesses are storage operand accesses made while the CPU is in the transactional-execution mode, with the exception of accesses made by the NONTRANSACTIONAL STORE instruction.

Transactional-Execution Mode: The term transactional-execution mode describes the common operation of both the nonconstrained and the constrained transactional-execution modes. Thus, when the transaction nesting depth is nonzero, the CPU is in the transactional-execution mode. Where specific operation is described, the terms nonconstrained and constrained are used to qualify the transactional-execution mode.

When the transaction nesting depth is zero, the CPU is not in the transactional-execution mode (also called the nontransactional-execution mode).

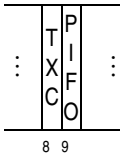
Transactional-Execution Facility Controls

The following section describes the controls that affect the operation of the transactional-execution facility.

Control Register Bits

The transactional-execution facility is controlled by two bits in control register zero and three bits in control register two.

Control Register 0 Bits: The bit assignments are as follows:



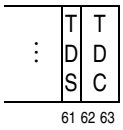
Transactional-Execution Control (TXC): Bit 8 of control register zero is the transactional-execution control. This bit provides a mechanism whereby the control program can indicate whether or not the transactional-execution facility is usable by the program. Bit 8 must be one to successfully enter the transactional-execution mode.

When bit 8 of control register 0 is zero, attempted execution of the EXTRACT TRANSACTION NESTING DEPTH, TRANSACTION BEGIN, and TRANSACTION END instructions results in a special-operation exception.

Transactional-Execution Program-Interruption-Filtering Override (PIFO): Bit 9 of control register zero is the transactional-execution program-interruption-filtering override. This bit provides a mechanism by which the control program can ensure that any program-exception condition that occurs while the CPU is in the transactional-execution mode results in an interruption, regardless of the effective program-interruption-filtering control specified or implied by the TRANSACTION BEGIN instruction(s). See "Program-Interruption Filtering on a Transaction Abort" on page 5-103 for details.

Programming Note: Even though facility indication 73 may be one (indicating that the transactional-execution facility is installed in the configuration), the program should determine whether the facility is enabled by examining an OS-provided indication of whether the OS supports the facility.

Control Register 2 Bits: **The bit assignments are as follows:**



Transaction Diagnostic Scope (TDS): Bit 61 of control register 2 controls the applicability of the transaction diagnostic control (TDC) in bits 62-63 of the register, as follows.

TDS

Value	Meaning
0	The TDC applies regardless of whether the CPU is in the problem or supervisor state.
1	The TDC applies only when the CPU is in the problem state. When the CPU is in the supervisor state, processing is as if the TDC contained zero.

Transaction Diagnostic Control (TDC): Bits 62-63 of control register 2 are a 2-bit unsigned integer that may be used to cause transactions to be randomly aborted for diagnostic purposes. The encoding of the TDC is as follows:

TDC

Value	Meaning
0	Normal operation; transactions are not aborted as a result of the TDC.
1	Abort every transaction at a random instruction, but before execution of the outermost TRANSACTION END instruction.
2	Abort random transactions at a random instruction.
3	Reserved.

When a transaction is aborted due to a nonzero TDC, then either of the following may occur:

- The abort code is set to any of codes 7-11, 13-16, or 255, with the value of the code randomly chosen by the CPU; the condition code is set corresponding to the abort code, as indicated in Figure 5-14 on page 5-101.

- For a nonconstrained transaction, the condition code is set to one. In this case, the abort code is not applicable.

It is model dependent whether TDC value 1 is implemented. If not implemented, a value of 1 acts as if 2 was specified.

For a constrained transaction, a TDC value of 1 is treated as if a TDC value of 2 was specified.

If a TDC value of 3 is specified, the results are unpredictable.

Transaction-Diagnostic-Block Address (TDBA)

When the B_1 field of an outermost TRANSACTION BEGIN (TBEGIN) instruction is nonzero, a valid transaction-diagnostic-block address (TDBA) is set from the first-operand address of the instruction. When the CPU is in the primary-space or access-register mode, the TDBA designates a location in the primary address space. When the CPU is in the secondary-space, or home-space mode, the TDBA designates a location in the secondary or home address space, respectively. When DAT is off, the TDBA designates a location in real storage.

The TDBA is used by the CPU to locate the transaction diagnostic block – called the *TBEGIN-specified TDB* – if the transaction is subsequently aborted. The rightmost three bits of the TDBA are always zero, meaning that the TBEGIN-specified TDB is on a doubleword boundary. The transaction diagnostic block is described in "Transaction Diagnostic Block (TDB)" on page 5-93.

When the B_1 field of an outermost TRANSACTION BEGIN (TBEGIN) instruction is zero, the transaction-diagnostic-block address is invalid, and no TBEGIN-specified TDB is stored if the transaction is subsequently aborted.

Programming Note: Abort processing may be significantly slower when the TDBA is valid.

Transaction-Abort PSW (TAPSW)

During the execution of the TRANSACTION BEGIN (TBEGIN) instruction when the nesting depth is initially zero, the transaction-abort PSW is set to the contents of the current PSW; the instruction address of the transaction-abort PSW designates the next sequential instruction (that is, the instruction following the outermost TBEGIN). During the execution of the TRANSACTION BEGIN (TBEGINC) instruction when the nesting depth is initially zero, the transaction-abort PSW is set to the contents of the current PSW, except that the instruction address of the transaction-abort PSW designates the TBEGINC instruction (rather than the next sequential instruction following the TBEGINC).

When a transaction is aborted, the condition code in the transaction-abort PSW is replaced with a code indicating the severity of the abort condition. Subsequently, if the transaction was aborted due to causes that do not result in an interruption, the PSW is loaded from the transaction-abort PSW; if the transaction was aborted due to causes that result in an interruption, the transaction-abort PSW is stored as the interruption old PSW.

The transaction-abort PSW is not altered during the execution of any inner TRANSACTION BEGIN instruction.

Transaction Nesting Depth (TND)

The transaction nesting depth is a 16-bit unsigned value that is incremented each time a TRANSACTION BEGIN instruction is completed with condition code 0 and decremented each time a TRANSACTION END instruction is completed with condition code 0. The transaction nesting depth is reset to zero when a transaction is aborted or by CPU reset.

All models implement a maximum TND of 15.

Programming Notes:

1. When the CPU is in the constrained transactional-execution mode, the transaction nesting depth is always one.
2. Although the maximum TND can be represented as a 4-bit value, the TND is defined to be a 16-bit value to facilitate its inspection in the transaction-diagnostic block.

Transaction Diagnostic Block (TDB)

When a transaction is aborted, various status information is saved in a transaction diagnostic block (TDB). The CPU places status into one or two TDBs, as follows:

TBEGIN-Specified TDB: The 256-byte location specified by a valid transaction-diagnostic-block address. When the transaction-diagnostic-block address is valid, the TBEGIN-specified TDB is always stored on a transaction abort. When the transaction-diagnostic-block address is not valid, a TBEGIN-specified TDB is not stored.

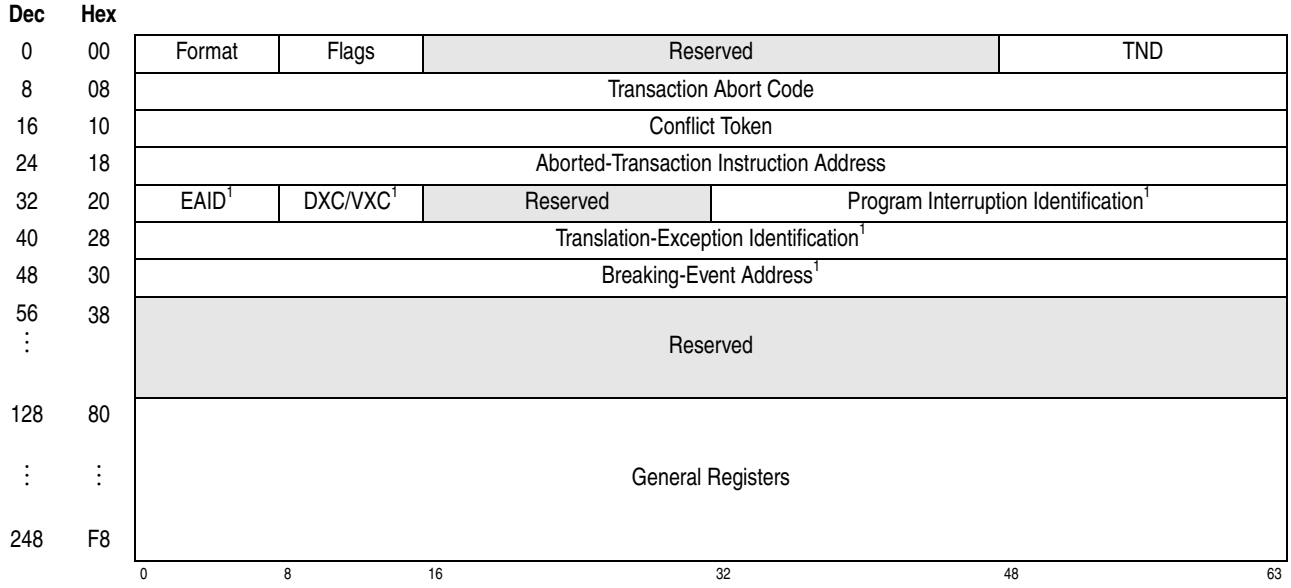
Access exceptions for the TBEGIN-specified TDB are recognized during the execution of the outermost TRANSACTION BEGIN (TBEGIN) instruction. If an access exception is recognized for the TBEGIN-specified TDB, the CPU does not enter the transactional-execution mode.

The recognition of PER storage-alteration events and zero-address-detection events for the TBEGIN-specified TDB is described in the section "PER Operation" on page 4-28.

Program-Interruption TDB: Real locations 6,144-6,399 (1800-18FF hex). The program-interruption TDB is stored when a transaction is aborted due to program interruption . When a transaction is aborted due to other causes, the contents of the program-interruption TDB are unpredictable.

The program-interruption TDB is not subject to any protection mechanism. PER storage-alteration events are not detected for the program-interruption TDB when it is stored during a program interruption.

The transaction diagnostic block has the following format:



Explanation:

1 Field is stored only in the TBEGIN-specified TDB; otherwise, the field is reserved. The program interruption identification is only stored for filtered program-interruption conditions. The translation-exception identification are stored only for filtered access-list-controlled, DAT, or instruction-execution protection, ASCE-type, page translation, region-first translation, region-second translation, region-third translation, and segment translation program-interruption conditions. The DXC is stored only for filtered data program-exception conditions. The VXC is stored only for filtered vector-processing program-exception conditions.

Figure 5-13. Transaction Diagnostic Block (TDB)

The fields of the transaction diagnostic block are as follows:

Format: Byte 0 contains a validity and format indication, as follows:

Value	Meaning
0	The remaining fields of the TDB are unpredictable.

Value	Meaning
1	A format-1 TDB, the remaining fields of which are described below.
2-255	Reserved

Programming Note: A TDB in which the format field is zero is referred to as a *null TDB*.

Flags: Byte 1 contains various indications, as follows:

Conflict-Token Validity (CTV): When a transaction is aborted due to a fetch or store conflict (that is, abort codes 9 or 10, respectively), bit 0 of byte 1 is the conflict-token-validity indication. When the CTV indication is one, the conflict token in bytes 16-23 of the TDB contain the logical address at which the conflict was detected. When the CTV indication is zero, bytes 16-23 of the TDB are unpredictable.

When a transaction is aborted due to any reason other than a fetch or store conflict, bit 0 of byte 1 is stored as zero.

Constrained-Transaction Indication (CTI): When the CPU was in the constrained transactional-execution mode, bit 1 of byte 1 is set to one. When the CPU was in the nonconstrained transactional-execution mode, bit 1 of byte 1 is set to zero.

Reserved: Bits 2-7 of byte 1 are reserved, and stored as zeros.

Transaction Nesting Depth (TND): Bytes 6-7 contain the transaction nesting depth when the transaction was aborted.

Transaction Abort Code (TAC): Bytes 8-15 contain a 64-bit unsigned transaction abort code. Each code point indicates a reason for a transaction being aborted, as summarized in Figure 5-14 on page 5-101.

It is model dependent whether the transaction abort code is stored in the program-interruption TDB when a transaction is aborted due to conditions other than a program interruption.

Conflict Token: For transactions that are aborted due to fetch or store conflict (that is, abort codes 9 and 10, respectively), bytes 16-23 contain the logical address of the storage location at which the conflict was detected. The conflict token is meaningful only when the CTV bit, bit 0 of byte 1, is one.

When the CTV bit is zero, bytes 16-23 are unpredictable.

Programming Note: Because of speculative execution by the CPU, the conflict token may designate a storage location that would not necessarily be accessed by the transaction's conceptual execution sequence.

Aborted-Transaction Instruction Address (ATIA): Bytes 24-31 contain an instruction address that identifies the instruction that was executing when an abort was detected. When a transaction is aborted due to abort codes 2, 5, 6, 11, 13, 19, or 256 or higher, or when a transaction is aborted due to abort codes 4 or 12 and the program-exception condition is nullifying, the ATIA points directly to the instruction that was being executed. When a transaction is aborted due to abort codes 4 or 12, and the program-exception condition is not nullifying, the ATIA points past the instruction that was being executed (see the programming note, below).

When a transaction is aborted due to abort codes 7-10, 14-16, or 255, the ATIA does not necessarily indicate the exact instruction causing the abort, but may point to an earlier or later instruction within the transaction.

If a transaction is aborted due to an instruction that is the target of an execute-type instruction, the ATIA identifies the execute-type instruction, either pointing to the instruction or past it, depending on the abort code as described above. The ATIA does not indicate the target of the execute-type instruction.

The ATIA is subject to the addressing mode when the transaction is aborted. In the 24-bit addressing mode, bits 0-39 of the field contain zeros. In the 31-bit addressing mode, bits 0-32 of the field contain zeros.

It is model dependent whether the aborted-transaction instruction address is stored in the program-interruption TDB when a transaction is aborted due to conditions other than a program interruption.

Programming Note: When a transaction is aborted due to abort code 4 or 12, and the program-exception condition is not nullifying, the ATIA does not point to the instruction causing the abort. By subtracting

the number of halfwords indicated by the interruption-length code (ILC) from the ATIA, the instruction causing the abort can be identified in conditions that are suppressing or terminating, or for non-PER events that are completing. When a transaction is aborted due to a PER event, and no other program-exception condition is present, the ATIA is unpredictable.

When the transaction-diagnostic-block address is valid, the ILC may be examined in program-interruption identification (PIID) in bytes 36-39 of the TBEGIN-specified TDB. When filtering does not apply, the ILC may be examined in the PIID at location 140-143 in real storage.

Exception Access Identification (EAID): For transactions that are aborted due to certain filtered program-exception conditions, byte 32 of the TBEGIN-specified TDB contains the exception access identification. The format of the EAID, and the cases for which it is stored, are identical to those described in real location 160 when the exception condition results in an interruption. See "Exception Access Identification" on page 3-74 for details.

For transactions that are aborted for other reasons, including any exception conditions that result in a program interruption, byte 32 is unpredictable. Byte 32 is always unpredictable in the program-interruption TDB .

Data-Exception Code (DXC) / Vector-Exception Code (VXC): For transactions that are aborted due to filtered data-exception program-exception conditions, byte 33 of the TBEGIN-specified TDB contains the data-exception code. The format of the DXC, and the cases for which it is stored, are identical to those described in real location 147 when the exception condition results in an interruption. See "Data-Exception Code" on page 3-74 and page 6-17 for details.

For transactions that are aborted due to filtered vector-processing exception program-exception conditions, byte 33 of the TBEGIN-specified TDB contains the vector-exception code. The format of the VXC, and the cases for which it are stored, are identical to those described in real location 147 when the exception condition results in an interruption. See ~~"Vector Processing Exception" on page 6-47~~ for details.

For transactions that are aborted for other reasons, including any exception conditions that result in a program interruption, byte 33 is unpredictable. Byte 33 is always unpredictable in the program-interruption TDB .

Program Interruption Identification (PIID): For transactions that are aborted due to filtered program-exception conditions, bytes 36-39 of the TBEGIN-specified TDB contain the program interruption identification. The format of the PIID is identical to that described in real locations 140-143 when the condition results in an interruption, except that (a) the instruction-length code in bits 13-14 of the PIID is respective to the instruction at which the exception condition was detected, and (b) bit 22 of the PIID is unpredictable. See "Program-Interruption Identification" on page 3-74 for details.

For transactions that are aborted for other reasons, including exception conditions that result in a program interruption, bytes 36-39 are unpredictable. Bytes 36-39 are always unpredictable in the program-interruption TDB .

Translation-Exception Identification (TEID): For transactions that are aborted due to any of the following filtered program-exception conditions, bytes 40-47 of the TBEGIN-specified TDB contain the translation-exception identification.

- Access-list-controlled, DAT, or instruction-execution protection
- ASCE-type
- Page translation
- Region-first translation
- Region-second translation
- Region-third translation
- Segment translation exception

The format of the TEID is identical to that described in real locations 168-175 when the condition results in an interruption. See "Translation-Exception Identification" on page 3-76 for details.

For transactions that are aborted for other reasons, including exception conditions that result in a program interruption, bytes 40-47 are unpredictable. Bytes 40-47 are always unpredictable in the program-interruption TDB .

Breaking-Event Address: For transactions that are aborted due to filtered program-exception conditions, bytes 48-55 of the TBEGIN-specified TDB contain the breaking-event address. The format of the breaking-event address is identical to that described in real locations 272-279 when the condition results

in an interruption. See “Breaking-Event Address” on page 3-80 and page 4-44 for details.

For transactions that are aborted for other reasons, including exception conditions that result in a program interruption, bytes 48-55 are unpredictable. Bytes 48-55 are always unpredictable in the program-interruption TDB.

General Registers: Bytes 128-255 contain the contents of general registers 0-15 at the time the transaction was aborted. The registers are stored in ascending order, beginning with general register 0 in bytes 128-135, general register 1 in bytes 136-143, and so forth.

Reserved: All other fields are reserved. Unless indicated otherwise, the contents of reserved fields are unpredictable.

As observed by other CPUs and the channel subsystem, accesses to the TDB(s) during a transaction abort are multiple-access update references occurring after any nontransactional stores, and the fields of the TDB(s) are not necessarily accessed in any order.

Programming Notes:

1. A transaction may be aborted due to causes that are outside the scope of the immediate configuration in which it executes. For example, transient events recognized by a hypervisor (such as LPAR or z/VM) may cause a transaction to be aborted.

2. The information provided in the transaction-diagnostic block is intended for diagnostic purposes and is substantially correct. However, because an abort may have been caused by an event outside the scope of the immediate configuration, information such as the abort code or program interruption identification may not accurately reflect conditions within the configuration, and thus should not be used in determining program action.

3. An example of the scenario described in the above notes may occur if a configuration’s first execution of an advanced-floating-point (AFP) instruction occurs within a transaction. The control program can eliminate this scenario by executing the first AFP instruction outside of a transaction.

In addition to the diagnostic information saved in the TDB, when a transaction is aborted due to any data-exception program-exception condition and both the AFP-register control, bit 45 of control register 0, and the effective allow-floating-point-operation control (F) are one, the data-exception code (DXC) is placed into byte 2 of the floating-point control register (FPCR), regardless of whether filtering applies to the program-interruption condition. When a transaction is aborted, and either or both the AFP-register control or effective allow-floating-point-operation control are zero, the DXC is not placed into the FPCR.

Transactional-Execution Facility Instructions

When the transactional-execution facility is installed, the following general instructions are provided.

- EXTRACT TRANSACTION NESTING DEPTH
- NONTRANSACTIONAL STORE
- TRANSACTION ABORT
- TRANSACTION BEGIN
- TRANSACTION END

Restricted Instructions

When the CPU is in the transactional-execution mode, attempted execution of certain instructions is restricted and causes the transaction to be aborted.

When issued in the constrained transactional-execution mode, attempted execution of restricted instructions may also result in a transaction-constraint program interruption, or may result in execution proceeding as if the transaction was not constrained. See “Constrained Transaction” on page 5-107 for further details.

Restricted instructions include all instructions that are not defined in Chapters 7-9 and 18-25 of this document and the following nonprivileged instructions.

- COMPARE AND SWAP AND STORE
- LOAD GUARDED STORAGE CONTROLS
- PERFORM LOCKED OPERATION
- PERFORM PROCESSOR ASSIST
- STORE FACILITY LIST EXTENDED
- STORE GUARDED STORAGE CONTROLS
- SUPERVISOR CALL

Under the conditions listed below, the following instructions are restricted:

- BRANCH AND LINK (BALR), BRANCH AND SAVE (BASR), and BRANCH AND SAVE AND SET MODE, when the R_2 field of the instruction is nonzero and branch tracing is enabled
- BRANCH AND SAVE AND SET MODE and BRANCH AND SET MODE, when the R_2 field is nonzero and mode tracing is enabled; SET ADDRESSING MODE, when mode tracing is enabled
- MONITOR CALL, when a monitor-event condition is recognized

When the CPU is in the transactional-execution mode, it is model dependent whether the following instructions are restricted:

- CIPHER MESSAGE
- CIPHER MESSAGE WITH AUTHENTICATION
- CIPHER MESSAGE WITH CIPHER FEEDBACK
- CIPHER MESSAGE WITH CHAINING
- CIPHER MESSAGE WITH COUNTER
- CIPHER MESSAGE WITH OUTPUT FEEDBACK
- COMPRESSION CALL
- COMPUTE INTERMEDIATE MESSAGE DIGEST
- COMPUTE LAST MESSAGE DIGEST
- COMPUTE MESSAGE AUTHENTICATION CODE
- CONVERT UNICODE-16 TO UNICODE-32
- CONVERT UNICODE-16 TO UNICODE-8
- CONVERT UNICODE-32 TO UNICODE-16
- CONVERT UNICODE-32 TO UNICODE-8
- CONVERT UNICODE-8 TO UNICODE-16
- CONVERT UNICODE-8 TO UNICODE-32
- PERFORM CRYPTOGRAPHIC COMPUTATION
- PERFORM RANDOM NUMBER OPERATION

When the effective allow-AR-modification (A) control is zero, the following instructions are restricted:

- COPY ACCESS
- LOAD ACCESS MULTIPLE
- LOAD ADDRESS EXTENDED
- SET ACCESS

When the effective allow-floating-point-operation (F) control is zero, all floating-point and vector instructions are restricted (that is, all instructions described in Chapters 9 and 18-25).

Under certain circumstances, the following instructions may be restricted:

- EXTRACT CPU TIME
- EXTRACT PSW
- STORE CLOCK
- STORE CLOCK EXTENDED
- STORE CLOCK FAST

It is model dependent whether the following instructions are restricted:

- PREFETCH DATA (RELATIVE LONG), when the code in the M_1 field is either 6 or 7.
- STORE CHARACTERS UNDER MASK (STCMH), when the M_3 field is zero and the code in the R_1 field is either 6 or 7.

If a model does not restrict the above two instructions for the cited conditions, then it is unpredictable whether attempted execution results in transactional execution being aborted with abort code 16.

When a nonconstrained transaction is aborted because of the attempted execution of a restricted instruction, the transaction abort code in the transaction diagnostic block is set to 11 (restricted instruction), and the condition code is set to 3, except as follows. When a nonconstrained transaction is aborted due to the attempted execution of an instruction that would otherwise result in a privileged-operation exception, it is unpredictable whether the abort code is set to 11 (restricted instruction) or 4 (unfiltered program interruption resulting from the recognition of the privileged-operation program interruption). When a nonconstrained transaction is aborted due to the attempted execution of MONITOR CALL, and both a monitor-event condition and a specification-exception condition are present, it is unpredictable whether the abort code is set to 11 or 4, or, if the program interruption is filtered, 12.

Additional instructions may be restricted in a constrained transaction, as described in “Constrained Transaction” on page 5-107. Although these instructions are not currently defined to be restricted in a nonconstrained transaction, they may be restricted under certain circumstances in a nonconstrained transaction on future processors. See “Constrained Transaction” on page 5-107 for details on the additional restrictions and alternate results if a restricted instruction is attempted.

Programming Notes:

1. Certain restricted instructions may be allowed in the transactional-execution mode on future processors. Therefore, the program should not rely on the transaction being aborted due to the attempted execution of a restricted instruction. The TRANSACTION ABORT instruction should be used to reliably cause a transaction to be aborted.

2. In a nonconstrained transaction, the program should provide an alternative nontransactional code path to accommodate a transaction that aborts due the attempted execution of an instruction that is restricted under certain conditions.

3. The effective allow-AR-modification and effective allow-floating-point-operation controls (A and F bits, respectively) are described in "TRANSACTION BEGIN (TBEGIN)" on page 7-395.

Transactional-Execution Facility Operation

The transactional-execution facility is under the control of bits 8-9 of control register 0, bits 61-63 of control register 2, the transaction nesting depth, the transaction-diagnostic-block address, and the transaction-abort PSW. See "Transactional-Execution Facility Controls" on page 5-92 for details on these controls.

Following an initial CPU reset, the contents of bit positions 8-9 of control register 0, bit positions 62-63 of control register 2, and the transaction nesting depth are set to zero. When the transactional-execution control, bit 8 of control register 0, is zero, the CPU cannot be placed into the transactional-execution mode.

Transaction Initiation

When the transaction nesting depth is zero, execution of the TRANSACTION BEGIN (TBEGIN) instruction resulting in condition code zero causes the CPU to enter the nonconstrained transactional-execution mode. When the transaction nesting depth is zero, execution of the TRANSACTION BEGIN (TBEGINC) instruction resulting in condition code zero causes the CPU to enter the constrained transactional-execution mode.

When the CPU is in the nonconstrained transactional-execution mode, execution of the TRANSACTION BEGIN instruction resulting in condition code zero causes the CPU to remain in the nonconstrained transactional-execution mode. Details of the instructions may be found in "TRANSACTION BEGIN (TBEGIN)" on page 7-395 and "TRANSACTION BEGIN (TBEGINC)" on page 7-399.

Execution in the Transactional-Execution Mode

Except where explicitly noted otherwise, all rules that apply for nontransactional execution also apply to transactional execution. This section describes additional characteristics of processing while the CPU is in the transactional-execution mode.

As observed by the CPU, fetches and stores made in the transactional-execution mode are no different than those made while not in the transactional-execution mode. See "Storage-Operand Consistency" on page 5-125 and "Relation between Operand Accesses" on page 5-129 for a description of the rules of storage ordering.

As observed by other CPUs and by the channel subsystem, all storage-operand accesses made while a CPU is in the transactional-execution mode appear to be a single block-concurrent access. Storage accesses for instruction and DAT- and ART-table fetches follow the non-transactional rules.

Normal Transaction Ending

The CPU leaves the transactional-execution mode normally by means of a TRANSACTION END instruction that causes the transaction nesting depth to transition to zero; in this case the transaction is said to complete. See "TRANSACTION END" on page 7-401 for the details.

When the CPU leaves the transactional-execution mode by means of the completion of a TRANSACTION END instruction, all stores made while in the transactional-execution mode are committed, that is, the stores appear to occur as a single block-concurrent operation as observed by other CPUs and by the channel subsystem.

Transaction Abort Conditions

A transaction may be implicitly aborted for a variety of causes, or it may be explicitly aborted by the TRANSACTION ABORT instruction. The following text enumerates each possible cause of a transaction

abort, the corresponding abort code, and the condition code that is placed into the transaction-abort PSW.

External Interruption: The transaction-abort code is set to 2, and the condition code in the transaction-abort PSW is set to 2. The transaction-abort PSW is stored as the external-old PSW as a part of external-interruption processing.

Program Interruption (Unfiltered): A program-exception condition that results in an interruption (that is, an unfiltered condition) causes the transaction to be aborted with code 4. The condition code in the transaction-abort PSW is set specific to the program interruption code, as described in "Transaction Abort Processing" on page 5-102 and Figure 5-15 on page 5-104. The transaction-abort PSW is stored as the program-old PSW as a part of program-interruption processing.

An instruction that would otherwise result in a transaction being aborted due to an operation exception may yield alternate results: For a nonconstrained transaction, the transaction may instead aborted with abort code 11 (restricted instruction); for a constrained transaction, a transaction-constraint program interruption may be recognized instead of the operation exception.

When a PER event is recognized in conjunction with any other unfiltered program-exception condition, the condition code is set to 3.

Machine-Check Interruption: The transaction-abort code is set to 5, and the condition code in the transaction-abort PSW is set to 2. The transaction-abort PSW is stored as the machine-check-old PSW as a part of machine-check interruption processing.

I/O Interruption: The transaction-abort code is set to 6, and the condition code in the transaction-abort PSW is set to 2. The transaction-abort PSW is stored as the I/O-old PSW as a part of I/O interruption processing.

Fetch Overflow: A fetch-overflow condition is detected when the transaction attempts to fetch from more locations than the CPU supports. The transaction-abort code is set to 7, and the condition code is set to either 2 or 3.

Store Overflow: A store-overflow condition is detected when the transaction attempts to store to more locations than the CPU supports. The transaction-abort code is set to 8, and the condition code is set to either 2 or 3.

Fetch Conflict: A fetch-conflict condition is detected when another CPU or the channel subsystem attempts to store into a location that has been transactionally fetched by this CPU. The transaction-abort code is set to 9, and the condition code is set to 2.

Store Conflict: A store-conflict condition is detected when another CPU or the channel subsystem attempts to access a location that has been stored during transactional execution by this CPU. The transaction-abort code is set to 10, and the condition code is set to 2.

Restricted Instruction: When the CPU is in the transactional-execution mode, attempted execution of a restricted instruction causes the transaction to be aborted. The transaction-abort code is set to 11, and the condition code is set to 3.

Restricted instructions, and the conditions under which they are restricted is described in "Restricted Instructions" on page 5-97.

Program-Interruption Condition (Filtered): A program-interruption condition that does not result in an interruption (that is, a filtered condition) causes the transaction to be aborted with a transaction-abort code of 12. The condition code is set to 3. See "Program-Interruption Filtering on a Transaction Abort" on page 5-103 for details on program-interruption filtering.

Nesting Depth Exceeded: The nesting-depth-exceeded condition is detected when the transaction nesting depth is at the maximum allowable value for the configuration, and a TRANSACTION BEGIN instruction is executed. The transaction is aborted with a transaction-abort code of 13, and the condition code is set to 3.

Cache Fetch-Related Condition: A condition related to storage locations fetched by the transaction is detected by the CPU's cache circuitry. The transaction is aborted with a transaction-abort code of 14, and the condition code is set to either 2 or 3.

Cache Store-Related Condition: A condition related to storage locations stored by the transaction is detected by the CPU's cache circuitry. The trans-

action is aborted with a transaction-abort code of 15, and the condition code is set to either 2 or 3.

Cache Other Condition: A cache other condition is detected by the CPU's cache circuitry. The transaction is aborted with a transaction-abort code of 16, and the condition code is set to 2 or 3.

During transactional execution, if the CPU accesses instructions or storage operands using different logical addresses that are mapped to the same absolute address, it is model dependent whether the transaction is aborted. If the transaction is aborted due to accesses using different logical addresses mapped to the same absolute address, abort code 14, 15, or 16 is set, depending on the condition.

Guarded-Storage Event: Execution of a LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED instruction resulted in the recognition of a guarded-storage event. The transaction is aborted with a transaction-abort code of 19, and the condition code is set to two.

If a guarded-storage event is recognized while the CPU is in the transactional-execution mode, the guarded-storage event is processed as described in "Guarded-Storage-Event Processing" on page 4-70. If the guarded-storage event is recognized while the CPU is in the constrained transactional-memory mode, no transaction diagnostic block is stored.

Miscellaneous Condition: A miscellaneous condition is any other condition recognized by the CPU that causes the transaction to abort. The transaction-abort code is set to 255, and the condition code is set to either 2 or 3.

When multiple configurations are executing in the same machine (for example, logical partitions or virtual machines), a transaction may be aborted due to an external, machine-check, or I/O interruption that occurred in a different configuration.

Programming Notes:

1. The miscellaneous condition may result from any of the following:
   - COMPARE AND REPLACE DAT TABLE ENTRY, COMPARE AND SWAP AND PURGE, INVALIDATE DAT TABLE ENTRY, INVALIDATE PAGE TABLE ENTRY, PERFORM FRAME MANAGEMENT FUNCTION in which the NQ control is zero and SK control is one, SET STORAGE KEY EXTENDED in which the NQ control is zero, performed by another CPU in the configuration; the condition code is set to 2.
   - An operator function (such as reset, restart, or stop) or the equivalent SIGNAL PROCESSOR order is performed on the CPU. The condition code is set to 2.
   - Any other condition not enumerated above; the condition code is set to 2 or 3.
2. The location at which fetch and store conflicts are detected may be anywhere within the same cache line.
3. Under certain conditions, the CPU may not be able to distinguish between similar abort conditions. For example, a fetch or store overflow may be indistinguishable from a respective fetch or store conflict.
4. Speculative execution of multiple instruction paths by the CPU may result in a transaction being aborted due to conflict or overflow conditions, even if such conditions do not occur in the conceptual sequence. While in the constrained transactional-execution mode, the CPU may temporarily inhibit speculative execution, allowing the transaction to attempt to complete without detecting such conflicts or overflows speculatively.

When multiple abort conditions apply, it is unpredictable which abort code is reported by the CPU.

TABORT Instruction: Execution of the TRANSACTION ABORT instruction causes the transaction to abort. The transaction-abort code is set from the second-operand address. The condition code is set to either 2 or 3, depending on whether bit 63 of the second-operand address is zero or one, respectively.

Figure 5-14 summarizes the abort codes stored in the transaction diagnostic block, and the corresponding condition code.

Code	Reason for Abort	CC Set
2	External interruption	2
4	Program interruption (unfiltered)	2 or 3 †
5	Machine-check interruption	2
6	I/O interruption	2

Figure 5-14. Transaction Abort Codes

Code	Reason for Abort	CC Set
7	Fetch overflow	2 or 3
8	Store overflow	2 or 3
9	Fetch conflict	2
10	Store conflict	2
11	Restricted instruction	3
12	Program-interruption condition (filtered)	3
13	Nesting depth exceeded	3
14	Cache fetch-related	2 or 3
15	Cache store-related	2 or 3
16	Cache other	2 or 3
19	Guarded-storage event recognized	2
255	Miscellaneous condition	2 or 3
≥ 256	TABORT instruction	2 or 3
‡	Cannot be determined; no TDB stored	1

Explanation:

† See Figure 5-16 on page 5-104 to determine which condition code is set based on the interruption code.

‡ This situation occurs when a transaction aborts, but the TDB has become inaccessible subsequent to the successful execution of the outermost TRANSACTION BEGIN (TBEGIN) instruction. No TBEGIN-specified TDB is stored, and the condition code is set to 1.

Figure 5-14. Transaction Abort Codes (Continued)

Programming Notes:

1. Abort codes 0-255 are reserved for use by the CPU. If the program specifies any of these codes in the TRANSACTION ABORT instruction, a specification exception is recognized, and the transaction is aborted with the resulting abort code indicating a program interruption (code 4) or program-interruption condition (code 12).

2. Abort codes 17-18 and 20-254 are reserved for potential enhancements. Such enhancements may result in the setting of either condition code 2 or 3.

3. Depending on the model, the CPU may not be able to distinguish between certain abort reasons. For example, a fetch/store overflow and a fetch/store conflict may not be distinguishable by the CPU in all circumstances.

   Therefore, the program should be able to accommodate unanticipated abort reasons.

4. Abort code 0 is reserved and will never be assigned to a meaningful abort indication.

Transaction Abort Processing

Abort processing consists of the following steps:

1. All nontransactional store accesses made while the CPU was in the transactional-execution mode are committed.

   All other (transactional) stores made while the CPU was in the transactional-execution mode are discarded.

2. The CPU leaves the transactional-execution mode. Subsequent stores occur nontransactionally.

3. When the transaction is aborted due to a guarded-storage event, the PSW instruction address and the guarded-storage-event parameter list (if accessible) are updated as described in "Guarded-Storage-Event Processing" on page 4-70.

   In all other cases, the current PSW is replaced with the contents of the transaction-abort PSW, except that the condition code is set according to Figure 5-14 on page 5-101 and Figure 5-16 on page 5-104.

4. When the transaction-diagnostic-block address is valid, diagnostic information identifying the reason for the abort and the contents of the general registers are stored into the TBEGIN-specified transaction diagnostic block (TDB). The TDB fields stored and conditions under which they are stored are described in "Transaction Diagnostic Block (TDB)" on page 5-93.

   If the transaction-diagnostic-block address is valid, but the block has become inaccessible subsequent to the execution of the outermost TRANSACTION BEGIN (TBEGIN) instruction, the block is not accessed, and condition code 1 applies. If the transaction-diagnostic-block address is not valid, no TBEGIN-specified TDB is stored, and condition code 2 or 3 applies, depending on the reason for aborting.

   For transactions that are aborted due to program-exception conditions that result in an interruption, the program-interruption TDB is stored. See "Transaction Diagnostic Block (TDB)" on page 5-93 for a description of the contents of the TDB.

5. The transaction nesting depth is set to zero.

6. Any general register pairs designated to be saved by the outermost TRANSACTION BEGIN instruction are restored. General register pairs that were not designated to be saved by the outermost TRANSACTION BEGIN instruction are not restored when a transaction is aborted.

7. A serialization function is performed.

8. If the transaction is aborted due to an exception condition that results in an interruption, any and all other interruption codes or parameters associated with the interruption are stored at the assigned-storage locations corresponding to the type of interruption. The current PSW, as set in step 3, is stored into the interruption-old PSW.

Programming Notes:

1. Attempted execution of a SUPERVISOR CALL instruction while in the transactional-execution mode results in the transaction being aborted due to a restricted instruction. An SVC interruption cannot occur in this case.

2. Access registers, vector registers, floating-point registers, the floating-point-control register, and any general registers not specified by the outermost TRANSACTION BEGIN instruction's general-register-save mask are not restored when a transaction aborts.

3. When the CPU was in the nonconstrained transactional-execution mode, the instruction address of the transaction-abort PSW designates the storage location following the outermost TRANSACTION BEGIN (TBEGIN) instruction. The sequence of instructions at this location should be able to accommodate all four condition codes, even though a failing transaction only causes codes 1, 2, and 3 to be set. A summary of the condition code meanings is as follows:

CC	Meaning
0	The transaction was successfully initiated.
1	The transaction was aborted due to an indeterminate condition. The transaction diagnostic block could not be stored. Reexecution of the transaction is unlikely to be productive.
2	The transaction was aborted due to a transient condition. Reexecution of the transaction may be productive.
3	The transaction was aborted due to a persistent condition. Reexecution of the transaction under current conditions is unlikely to be productive. If conditions change, for example, data that is manipulated transactionally is rearranged, then reexecution may be more productive.

   When the CPU was in the constrained transactional-execution mode, the instruction address of the transaction-abort PSW designated the TRANSACTION BEGIN (TBEGINC) instruction. By definition, a constrained transaction does not normally provide an abort-handler routine; however, when a transaction is aborted due to the recognition of a guarded-storage event, the guarded-storage-event handler may be used to remove the cause of the guarded-storage event.

4. When using nested transactions, an inner transaction may cause abort conditions to occur that may not otherwise occur in the outermost transaction. Examples of such conditions include the following:

   - The inner transaction may issue the TRANSACTION ABORT instruction, specifying an unanticipated abort code.
   - The inner transaction may result in an unanticipated program-interruption condition.
   - The inner transaction may filter program-interruption conditions that are not filtered by the outermost transaction, thus resulting in a different abort code.

   Any transaction abort-handler routine must be able to accommodate unanticipated abort and exception conditions that occur within inner transactions, even if they never occur in the outermost transaction.

Program-Interruption Filtering on a Transaction Abort

A program-exception condition that is recognized while in the transactional-execution mode always results in the transaction being aborted. For a nonconstrained transaction, the program can optionally specify that certain program-exception conditions not result in an interruption. This action is called program-interruption filtering. Program-interruption filtering is subject to the following controls:

- The transactional-execution program-interruption-filtering override, bit 9 of control register 0
- The effective program-interruption-filtering control (PIFC)
- The program-interruption code corresponding to the exception condition recognized.

When the transactional-execution program-interruption filtering override is zero, the program specifies which classes of exception conditions are to be filtered by means of the program-interruption-filtering control (PIFC), bits 14-15 of the I_2 field of the TRANSACTION BEGIN (TBEGIN) instruction. The effective PIFC is the highest value of the PIFC in the TBEGIN instruction for the current nesting level and for all outer levels.

For most program-exception conditions, there is a corresponding transactional-execution class. The effective PIFC and the transactional-execution classes interact as follows:

Effective PIFC	Results based on Transactional-Execution Class
0	No program-interruption filtering occurs. Exception conditions having classes 1, 2, or 3 always result in an interruption.
1	Limited program-interruption filtering occurs. Exception conditions having classes 1 or 2 result in an interruption; conditions having class 3 do not result in an interruption.
2	Moderate program-interruption filtering occurs. Only exception conditions having class 1 result in an interruption; conditions having classes 2 or 3 do not result in an interruption.

The TRANSACTION BEGIN (TBEGINC) instruction provides no explicit program-interruption-filtering control; an implied PIFC of zero is assumed for TBEGINC. Thus, when the CPU enters the constrained transactional-execution mode as a result of TBEGINC, the effective PIFC is zero; when the CPU remains in the nonconstrained transactional-execution mode as a result of TBEGINC, the effective PIFC is unchanged.

Figure 5-15 summarizes the relationship of the effective PIFC, the type of program-interruption filtering, the transactional-execution class, and whether the exception condition results in an interruption.

Effective PIFC	Program-Interruption Filtering	Result Based on Transactional-Execution (TX) Class		
		1	2	3
0	None	Interrupt	Interrupt	Interrupt
1	Limited	Interrupt	Interrupt	Filtered
2	Moderate	Interrupt	Filtered	Filtered

Explanation:

PIFC Program-Interruption-Filtering Control (bits 14-15 of the I_2 field of the TBEGIN instruction; zeros for TBEGINC); the effective PIFC is the highest value specified (or implied) for the current and all outer-level TRANSACTION BEGIN instructions.

Figure 5-15. Effective PIFC and Transactional-Execution Program-Interruption Classes

Figure 5-16 lists the program-interruption conditions, the corresponding transactional-execution class, and the condition code that is set when the transaction is aborted due to a program-interruption condition.

Code (Hex)	Exception Condition	TX Class	CC Set
0001	Operation	1	3
0002	Privileged operation	1	3
0003	Execute	1	3
0004	Protection	1 or 2 †	2 or 3 ‡
0005	Addressing	1 or 2 †	2 or 3 ‡
0006	Specification	3	2 or 3 ‡
0007	Data (DXC 01, 02, 03, and FE)	1	2
0007	Data (all other DXCs)	3	2 or 3 ‡
0008	Fixed-point overflow	3	2 or 3 ‡
0009	Fixed-point divide	3	2 or 3 ‡
000A	Decimal overflow	3	2 or 3 ‡
000B	Decimal divide	3	2 or 3 ‡
000C	HFP exponent overflow	3	2 or 3 ‡
000D	HFP exponent underflow	3	2 or 3 ‡
000E	HFP significance	3	2 or 3 ‡
000F	HFP divide	3	2 or 3 ‡
0010	Segment translation	1 or 2 †	2 or 3 ‡
0011	Page translation	1 or 2 †	2 or 3 ‡
0012	Translation specification	1	3
0013	Special operation	1	3
0015	Operand	—	—
0016	Trace table	—	—
0018	Transaction constraint	1	3
001B	Vector processing	3	2 or 3‡
001C	Space-switch event	—	—
001D	HFP square root	3	2 or 3 ‡

Figure 5-16. Transactional-Execution Classes for Various Program-Interruption Conditions (Part 1 of 2)

Code (Hex)	Exception Condition	TX Class	CC Set
001F	PC-translation specification	—	—
0020	AFX translation	—	—
0021	ASX translation	—	—
0022	LX translation	—	—
0023	EX translation	—	—
0024	Primary authority	—	—
0025	Secondary authority	—	—
0026	LFX translation	—	—
0027	LSX translation	—	—
0028	ALET specification	2	2 or 3 ‡
0029	ALEN translation	2	2 or 3 ‡
002A	ALE sequence	2	2 or 3 ‡
002B	ASTE validity	2	2 or 3 ‡
002C	ASTE sequence	2	2 or 3 ‡
002D	Extended authority	2	2 or 3 ‡
002E	LSTE sequence	—	—
002F	ASTE instance	—	—
0030	Stack full	—	—
0031	Stack empty	—	—
0032	Stack specification	—	—
0033	Stack type	—	—
0034	Stack operation	—	—
0038	ASCE type	1 or 2 †	2 or 3 ‡
0039	Region-first translation	1 or 2 †	2 or 3 ‡
003A	Region-second translation	1 or 2 †	2 or 3 ‡
003B	Region-third translation	1 or 2 †	2 or 3 ‡
0040	Monitor event	—	—
0080	PER event	1	3 *

Explanation:

— Not applicable; the exception cannot occur in the transactional-execution mode because the instruction that causes the exception is a restricted instruction.

† Access exceptions recognized during the fetching of an instruction in the transactional-execution mode are TX class 1 (that is, they cannot be filtered). Access exceptions when accessing a storage operand in the transactional-execution mode are TX class 2.

‡ When the exception condition is not filtered, the condition code is 2; when the condition is filtered, the condition code is 3.

* The CPU sets condition code 3. A control program or hypervisor may change the condition code in the program-old PSW to 2.

Figure 5-16. Transactional-Execution Classes for Various Program-Interruption Conditions (Part 2 of 2)

When a program-interruption condition does not result in an interruption (that is, the condition is filtered), the program-new PSW is not loaded, and none of the assigned storage locations associated with a program interruption is stored; these locations include the program-interruption identification, the breaking-event address, the program-old PSW, and, when applicable, the data-exception code, PER code, PER address, exception access identification, PER access identification, operand access identification, and translation-exception identification. When a program-interruption condition results in an interruption (that is, the condition is unfiltered), most assigned storage locations associated with a program interruption are stored as is normal; however, the instruction-length code in bits 13-14 of the program-interruption identification is respective to the instruction at which the exception condition was detected, and the transaction-abort PSW is stored as the program-old PSW.

When a PER event is recognized in conjunction with any other filtered program-exception condition, the following applies:

- The transaction class and condition code for the PER event apply. In this case, the PER exception condition cannot be filtered, and the condition code is set to 3.
- The program-interruption code in the prefix area does not include the non-PER exception condition, nor are any other non-PER program-interruption parameters stored in the prefix area.

When the transactional-execution program-interruption filtering override (bit 9 of control register 0) is one, program-interruption conditions are not subject to program-interruption filtering. In this case, execution proceeds as if the effective PIFC was zero.

Access-exception conditions recognized during the fetching of an instruction are never subject to program-interruption filtering. In these cases, the exception condition results both in the transaction being aborted and in a program interruption.

In addition to the program-interruption code representing the cause of the interruption, bit 6 of the program-interruption code at real locations 142-143 is set to one, indicating that the program interruption occurred during transactional execution. The fields of the program-old PSW are set as described in step 3 of “Transaction Abort Processing” on page 5-102.

Programming Notes:

1. A MONITOR CALL instruction that would otherwise cause a monitor-event is a restricted

instruction. Therefore, a monitor-event program interruption can never occur while the CPU is in the transactional-execution mode, thus the monitor code at real locations 176-183 is not stored when the transaction is aborted.

Similarly, any other program-interruption condition listed in Figure 5-16 having a not-applicable transaction class (–) and condition code cannot occur as the instructions that cause these exceptions are restricted. Thus, neither the program-interruption identification nor any of the other ancillary program-interruption information are stored in real locations in the prefix area.

2. The following example illustrates the instructions necessary to initiate and end a nonconstrained transaction. Note that if the transaction is aborted, it is retried several times before finally branching to the non-transactional fallback path at label NO_RETRY.

```
            LHI     15,0            Zero counter
LOOP        TBEGIN  TDB,X'F000'     Restore GRs 0-7 if aborted.
            JNZ     ABORT           CC≠0: Aborted or can't init.
              ⋮
              ⋮     Transactional-execution code
              ⋮
            TEND                    End of transaction.
              ⋮
ABORT       JC      5,NO_RETRY      CC 1/3: Not worth retrying.
            AHI     15,1            Increment counter.
            CIJNL   15,6,NO_RETRY   Give up after 6 attempts
            PPA     15,0,1          Request assistance.
            J       LOOP            And try it again.
              ⋮
NO_RETRY    DS      0H              Perform fall-back path
```

3. As shown in programming note 2, the first conditional branch instruction following a TBEGIN should be one of the following:

   - A branch to the abort handler for all possible aborting conditions; that is, a conditional branch instruction having a mask of 0111 binary (as shown in programming note 2). In this case, separate conditional branches in an out-of-line abort handler may be used to accommodate the three possible aborting conditions. This is the preferred sequence.
   - A branch to the next instruction to be executed in the transactional-execution mode; that is, a conditional branch instruction having a mask of 1000 binary. In this case, separate conditional branches to accommodate the aborting conditions may immediately follow the first conditional branch. This is not the preferred sequence and may result in significant performance degradation.

   Other branching combinations – such as having a first, second, and third conditional branch following the TBEGIN to handle the respective condition code 1, 2, and 3 cases – may result in unnecessary or repetitive aborting of the transaction.

4. Programming note 2 shows the use of the PERFORM PROCESSOR ASSIST (PPA) instruction. The program can improve the likelihood of success when redriving the transaction by invoking the PPA instruction before branching back to the label LOOP following an abort. PPA is described on page 7-345.

5. As shown in programming note 2, the loop counter in general register 15 is not subject to the general-register save mask (GRSM) in the TRANSACTION BEGIN instruction; thus, the register is not restored if the transaction aborts. Therefore, the program should not alter the contents of general register 15 within the transaction.

   However, if the loop counter was changed to use a register that is subject to the GRSM, for example general register 7, then the transaction could safely alter that register within the transaction, *and* the program could use the same register for the loop counter. If the transaction completes successfully, any results in general register 7 are available following the TRANSACTION END instruction. However, if the transaction aborts, the original value of general register 7 (that is, the value before the TBEGIN) is restored, thus the register can also be used for the loop counter in the abort handler.

6. Program-interruption filtering may be useful in programs that, for various reasons, defer the validation of data – sometimes called *speculative execution*. Instead of establishing a potentially complicated recovery environment, the program simply executes nonconstrained transaction(s) in which the effective program-interruption filtering control (PIFC) is nonzero. This allows the program's abort-handler routine to receive control for certain types of program-interruption conditions directly – without operating-system intervention.

An effective PIFC of 1 indicates that limited filtering is to be performed by the CPU; this may be useful in the recognition of unexpected data or arithmetic exceptions. An effective PIFC of 2 indicates that moderate filtering is to be performed; this may be useful in the recognition of inaccessible storage locations.

However, it should be noted that if a program is aborted due to various types of filtered access exceptions, it does not necessarily indicate that the location would be inaccessible if the program attempted nontransactional execution. For example, the program might specify a PIFC of 2, and subsequently be aborted due to a page-translation exception. This exception may indicate that the storage location is not part of the virtual address space, or it may simply indicate that the block of storage has been paged out.

Priority of Abort Conditions

During transactional execution, multiple abort conditions may be present simultaneously. Abort conditions having corresponding interruptions are honored in the order defined in "Priority of Program-Interruption Conditions" on page 6-52. Except for the restricted-instruction abort condition, abort conditions that do not correspond to interruptions may occur in any order. The priority of the restricted-instruction abort condition is equivalent to a program-interruption priority of 7.D (see "Priority of Program-Interruption Conditions" on page 6-52). Figure 5-17 illustrates the priority of each of the abort conditions.

Priority	Abort Condition
A.	Fetch-overflow (abort code 7)
B.	Store overflow (abort code 8)
C.	Fetch conflict (abort code 9)
D.	Store Conflict (abort code 10)
E.	Nesting depth exceeded (abort code 13)
F.	Cache fetch-related condition (abort code 14)
G.	Cache store-related condition (abort code 15)
H.	Cache other condition (abort code 16)
I.	Miscellaneous condition (abort code 255)

Figure 5-17. Priority of Abort Conditions

Priority			Abort Condition
J.	1.		Exigent machine check (abort code 5)
	2.	a.	Program interruption (unfiltered; abort code 4)
		b.	Restricted instruction (abort code 11)
		c.	Program interruption (filtered; abort code 12)
	3.	a.	TABORT instruction (abort code > 255)
	4.		Repressible machine check (abort code 5)
	5.		External interruption (abort code 2)
	6.		Input/output interruption (abort code 7)

Explanation:
The meaning of the priority indications is similar to that described in Figure 6-8 on page 6-52.

Figure 5-17. Priority of Abort Conditions (Continued)

Constrained Transaction

In the absence of constraint violations, repeated occurrences of interruptions, guarded-storage events, or conflicts with other CPUs or the channel subsystem, a constrained transaction will eventually complete, thus an abort-handler routine is not required. A constrained transaction is initiated by the TRANSACTION BEGIN (TBEGINC) instruction when the transaction nesting depth is initially zero. A constrained transaction is subject to the following constraints:

1. The transaction executes no more than 32 instructions, not including the TRANSACTION BEGIN (TBEGINC) and TRANSACTION END instructions.

2. All instructions in the transaction must be within 256 contiguous bytes of storage, including the TRANSACTION BEGIN (TBEGINC) and any TRANSACTION END instructions.

3. In addition to all instructions listed in the section "Restricted Instructions" on page 5-97, the following restrictions apply to a constrained transaction.

   a. Instructions are limited to those defined in Chapter 7, "General Instructions."

   b. Branching instructions are limited to the following:

      - BRANCH RELATIVE ON CONDITION in which the M_1 field is nonzero and the RI_2 field contains a positive value

      - BRANCH RELATIVE ON CONDITION LONG in which the M_1 field is nonzero, and the RI_2 field contains a positive

value that does not cause address wrap-around.

- COMPARE AND BRANCH RELATIVE, COMPARE IMMEDIATE AND BRANCH RELATIVE, COMPARE LOGICAL AND BRANCH RELATIVE, and COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE in which the M_3 field is nonzero and the RI_4 field contains a positive value

(that is, only forward branches with nonzero branch masks)

c. Except for TRANSACTION END and instructions which cause a specific-operand serialization, instructions which cause a serialization function are restricted.

d. All SS- and SSE-format instructions are restricted.

e. All of the following general instructions are restricted.

- BRANCH INDIRECT ON CONDITION
- BRANCH PREDICTION PRELOAD
- BRANCH PREDICTION RELATIVE PRELOAD
- CHECKSUM
- CIPHER MESSAGE
- CIPHER MESSAGE WITH AUTHENTICATION
- CIPHER MESSAGE WITH CIPHER FEEDBACK
- CIPHER MESSAGE WITH CHAINING
- CIPHER MESSAGE WITH COUNTER
- CIPHER MESSAGE WITH OUTPUT FEEDBACK
- COMPARE AND FORM CODEWORD
- COMPARE LOGICAL LONG, COMPARE LOGICAL LONG EXTENDED, and COMPARE LOGICAL LONG UNICODE
- COMPARE LOGICAL STRING
- COMPARE UNTIL SUBSTRING EQUAL
- COMPRESSION CALL
- COMPUTE INTERMEDIATE MESSAGE DIGEST
- COMPUTE LAST MESSAGE DIGEST
- COMPUTE MESSAGE AUTHENTICATION CODE
- CONVERT TO BINARY
- CONVERT TO DECIMAL
- CONVERT UNICODE-16 TO UNICODE-32, CONVERT UNICODE-16 TO UNICODE-8, CONVERT UNICODE-32 TO UNICODE-16, CONVERT UNICODE-32 TO UNICODE-8, CONVERT UNICODE-8 TO UNICODE-16, and CONVERT UNICODE-8 TO UNICODE-32
- DIVIDE, DIVIDE LOGICAL, and DIVIDE SINGLE
- EXECUTE and EXECUTE RELATIVE LONG
- EXTRACT CPU ATTRIBUTE
- EXTRACT CPU TIME
- EXTRACT PSW
- EXTRACT TRANSACTION NESTING DEPTH
- LOAD AND ADD, LOAD AND ADD LOGICAL, LOAD AND AND, LOAD AND EXCLUSIVE OR, and LOAD AND OR
- LOAD PAIR DISJOINT
- LOAD PAIR FROM QUADWORD
- MONITOR CALL
- MOVE LONG, MOVE LONG EXTENDED, and MOVE LONG UNICODE
- MOVE STRING
- NONTRANSACTIONAL STORE
- PERFORM CRYPTOGRAPHIC COMPUTATION
- PERFORM RANDOM NUMBER OPERATION
- PREFETCH DATA (RELATIVE LONG)
- SEARCH STRING and SEARCH STRING UNICODE
- SET ADDRESSING MODE
- STORE CHARACTERS UNDER MASK (STCMH), when the M_3 field is zero
- STORE CLOCK FAST
- STORE FACILITY LIST EXTENDED
- STORE PAIR TO QUADWORD
- TEST ADDRESSING MODE
- TRANSACTION ABORT
- TRANSACTION BEGIN (both TBEGIN and TBEGINC)
- TRANSLATE AND TEST EXTENDED and TRANSLATE AND TEST REVERSE EXTENDED
- TRANSLATE EXTENDED
- TRANSLATE ONE TO ONE, TRANSLATE ONE TO TWO, TRANSLATE TWO TO ONE, and TRANSLATE TWO TO TWO
- UPDATE TREE

4. The transaction's storage operands access no more than four octowords. Note: LOAD ON CONDITION and STORE ON CONDITION are considered to reference storage regardless of the condition code.

5. The transaction executing on this CPU, or stores by other CPUs or the channel subsystem, do not access storage operands in any 4 K-byte blocks that contain the 256 bytes of storage beginning with the TRANSACTION BEGIN (TBEGINC) instruction.

6. The transaction does not access instructions or storage operands using different logical addresses that are mapped to the same absolute address.

7. Operand references made by each instruction in the transaction must be within a single doubleword, except that for LOAD ACCESS MULTIPLE, LOAD MULTIPLE, LOAD MULTIPLE HIGH, STORE ACCESS MULTIPLE, STORE MULTIPLE, and STORE MULTIPLE HIGH, operand references must be within a single octoword.

If a constrained transaction violates any of constraints 1-7, listed above, then it is unpredictable whether a transaction-constraint program interruption is recognized. However, if a transaction-constraint program interruption is not recognized for one transaction-constraint violation, it may still be recognized for a subsequent violation within the same constrained transaction.

If the execution of a LOAD GUARDED or LOAD LOGICAL AND SHIFT GUARDED instruction results in the recognition of a guarded-storage event while the CPU is in the constrained transactional-execution mode, the transaction is aborted and the PSW instruction address and guarded-storage-event parameters are updated as described in "Guarded-Storage-Event Processing" on page 4-70.

Programming Notes:

1. The chance of successfully completing a constrained transaction improves if the transaction meets the following criteria:

   a. The instructions issued are fewer than the maximum of 32.

   b. The storage-operand references are fewer than the maximum of 4 octowords.

   c. The storage-operand references are on the same cache line.

   d. Storage-operand references to the same locations occur in the same order by all transactions.

2. A constrained transaction is not necessarily assured of successfully completing on its first execution. However, if a constrained transaction that does not violate any of the listed constraints is aborted, the CPU employs special circuitry to ensure that a repeated execution of the transaction is subsequently successful.

3. Within a constrained transaction, TRANSACTION BEGIN is a restricted instruction, thus a constrained transaction cannot be nested.

4. Violation of any of constraints 1-7 by a constrained transaction may result in a program loop.

Engineering Notes:

Notes on the Definition:

Programming Notes:

Monitor-Event Counting

When the execution of a MONITOR CALL instruction causes a monitor event to occur, the enhanced-monitor facility is installed, and the enhanced-monitor-mask bit corresponding to the monitor class is one, a monitor-event counting operation is performed. See "MONITOR CALL" on page 7-288 for further details on the execution of the MONITOR CALL instruction.

The enhanced-monitor counter-array size contained in the word at real location 264, extended on the left with 32 binary zeros, is logically compared with the monitor code formed by the first-operand address. If the monitor code is greater than or equal to the enhanced-monitor counter-array size, then one is logically added to the enhanced-monitor exception count contained in the word at real location 268, and the instruction completes. Any carry as a result of this addition is ignored.

If the monitor code is less than the enhanced-monitor counter-array size (extended on the left with 32 zeros), then one is added to the enhanced-monitor counter corresponding to the monitor code. An enhanced-monitor counter comprises two discontiguous storage locations: an unsigned halfword which is incremented first, and an unsigned word which is incremented whenever a carry out of bit position 0 of the halfword addition occurs.

The enhanced-monitor counter-array origin in bits 0-60 of the doubleword at real location 256, appended on the right with three binary zeros, forms the 64-bit virtual address of the enhanced-monitor counter array in the home address space.

Although the enhanced-monitor counter array is located in the home address space (as designated by control register 13), the updating of a counter occurs regardless of whether DAT is on. The access to the counters may be performed with the use of the translation-lookaside buffer (TLB), and TLB entries may be formed, regardless of whether DAT is on.

The enhanced-monitor-counter-array-size field at real location 264 specifies the number of halfword and word entries in the array; the halfword entries in the array appear first, followed immediately by the word entries.

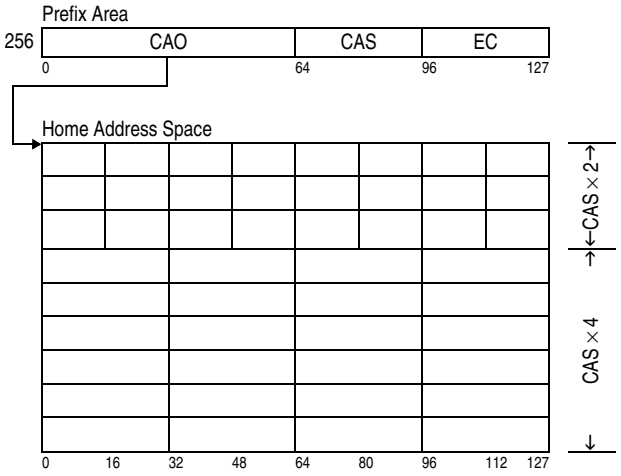
The monitor code multiplied by two is added to the enhanced-monitor counter-array origin to form the 64-bit virtual address of a halfword to which one is logically added. If this addition results in a carry out of bit position 0 of the halfword, then one is logically added to the corresponding word. The enhanced-monitor-counter-array-size value multiplied by two is added to the monitor code multiplied by four; this value is added to the enhanced-monitor counter-array origin to form the 64-bit virtual address of the word. Any carry in the word addition is ignored.

Key-controlled protection does not apply to any accesses made in the process of incrementing the counters. Except for addressing exceptions, any other access exception recognized in the process of incrementing the counters results in (a) one being logically added to the enhanced-monitor exception count contained in the word at real location 268, and (b) the instruction completing; any carry as a result of this addition is ignored. It is unpredictable whether an addressing-exception condition occurring in the process of incrementing an enhanced-monitor counter results in the incrementing of the enhanced-monitor exception counter or the recognition of the addressing exception.

The word portion of a counter is accessed only if the addition of the corresponding halfword portion of the counter results in an overflow. If the addition of the halfword results in an overflow, the storing of the updated halfword is performed only if both the halfword and word portions of the counter are accessible.

Locations within the enhanced-monitor counter array that are accessed by an enhanced-monitor counting operation are subject to PER storage-alteration events; however, PER storage-alteration events do not apply to updates made to the exception counter. During an enhanced-monitor counting operation, a PER storage-alteration event is recognized only if all counter-array locations updated by the event are accessible.

Figure 5-18 illustrates the structures used in the monitor-event-counting operation.



Explanation:

CAO Enhanced-monitor counter-array origin
CAS Enhanced-monitor counter-array size
EC Enhanced-monitor exception count

Figure 5-18. Monitor-Event-Counting Structures

Programming Notes:

1. It is recommended that the enhanced-monitor counter array be allocated on a cache-line boundary of the CPU's first-level data cache. The cache line size may be determined by the EXTRACT CPU ATTRIBUTES (ECAG) instruction.

2. The enhanced-monitor counters may be incremented using a non-interlocked update. Therefore, each enhanced-monitor counter array should be processor unique.

3. It is recommended that the enhanced-monitor-counter-array be allocated in fixed storage such that dynamic-address-translation (DAT) exceptions do not occur when accessing the array.

4. In the ESA/390-compatibility mode, all 16 bits of the enhanced-monitor mask (bits 16-31 of control register 8) are always zeros. Thus, an enhanced-monitor-counting event is never recognized in this mode.

ESA/390-Compatibility-Mode Facility

The ESA/390-compatibility-mode (390-CM) facility provides an environment supporting ESA/390 applications in a hybrid architectural mode having the following attributes:

- Except as described below, operation is similar to that of the ESA/390 architectural mode described in Reference [1.] on page xxix and the ESA/extended-configuration (ESA/XC) architecture described in Reference [12.] on page xxx.

- PSWs in assigned storage locations, in the second operand of LOAD PSW, and in the second operand of RESUME PROGRAM when the P bit in the parameter list is zero, all have the short-PSW format shown in Figure 4-3 on page 4-8, except that when loading the PSW, (a) bit 5 must be zero, and (b) it is unpredictable whether bit 31 must be zero.

- The prefix area is a 4 K-byte block, aligned on a 4 K-byte boundary. The prefix register accommodates absolute addresses on a 4 K-byte boundary.

  PSWs, interruption parameters, and the results of operator- or SIGP-initiated store-status operations in the prefix area are at the locations defined in the ESA/390 architecture.

- Low address protection applies only to effective addresses designating locations 0-511.

- Dynamic address translation is not supported. Attempted execution of any instruction that enables DAT results in the recognition of a specification exception.

  Attempted execution of an instruction that implicitly performs DAT, including LOAD PAGE TABLE ENTRY ADDRESS, LOAD REAL ADDRESS, and STORE REAL ADDRESS, results in an operation exception being recognized, except that attempted execution of LOAD REAL ADDRESS may result in a privileged-operation exception in the problem state.

- The expanded-storage facility is not installed. Attempted execution of the PAGE IN and PAGE OUT instructions results in an operation exception being recognized.

- Instructions that may perform address-space-number (ASN) translation are not supported. Attempted execution of an instruction that may perform ASN translation results in a special-operation exception being recognized, except that it is unpredictable whether attempted execution of LOAD ADDRESS SPACE PARAMETERS results in an operation exception, privileged-operation exception (when the CPU is in the problem state), special-operation exception, or specification exception (if the first operand is not doubleword aligned). Consequently, ASN tracing is not supported.

- The branch-and-set-authority facility is not supported. Attempted execution of the BRANCH AND SET AUTHORITY instruction results in either an operation exception or special-operation exception being recognized.

- Except as noted below, access-register translation is not supported within the scope of the configuration (since, in most cases ART requires that DAT is enabled).

  ART may be performed by the TEST ACCESS instruction. It is unpredictable whether a special-operation exception is recognized by TEST ACCESS when bit position 47 of control register 0 contains zero.

  For configurations that are operating in the ESA/XC mode, ART may be performed for the configuration using tables provided by the host program. See Reference [12.] on page xxx for details of ART performed in this context.

- Bits 0-31 of the general, control, and prefix registers as defined in the ESA/390 architecture

occupy bits 32-63 of the respective registers as defined in the z/Architecture architecture.

- Bits 0-31 of the (z/Architecture) control registers contain zeros and cannot be modified. Attempted execution of the LOAD CONTROL (LCTLG) instruction results in an operation exception being recognized. Because of this restriction, z/unique functions corresponding to the following controls are unavailable:
  - Transactional-execution control (CR0, bit 8)
  - Transactional-execution program-interruption-filtering override (CR0, bit 9)
  - Clock-comparator sign control (CR0, bit 10)
  - Measurement-counter-extraction-authorization control (CR0, bit 15)
  - Warning-track-interruption control (CR0, bit 30)
  - Enhanced-monitor masks (CR8, bits 16-31)
  - Branch and mode tracing controls (CR12, bits 0 and 1, respectively)
- It is unpredictable whether any of bits 32-63 of a control register that are unique to z/Architecture features have any effect, specifically:
  - It is unpredictable whether bit 32 of control registers 10 and 11 are used in forming the PER address range.
  - For explicit tracing, it is unpredictable whether bit 32 of control register 12 is used in forming the trace-entry address.
- If the program issues any other instruction that is defined as being unique to the z/Architecture architectural mode, it is unpredictable whether an operation exception is recognized or the instruction executes according to its z/Architecture definition.
- If the program issues an instruction that is defined to enable the 64-bit addressing mode, the results are as follows:
  - For BRANCH AND SAVE AND SET MODE and BRANCH AND SET MODE, when the R_2 field is nonzero and bit 63 of general register R_2 is one, it is unpredictable whether (a) the CPU enters the 64-bit addressing mode, (b) bit 31 of the PSW is set to one, resulting in an early specification exception being recognized, or (c) the branch address in general register R_2 is considered to be odd, resulting in a late specification exception being recognized.
  - For LOAD PSW and RESUME PROGRAM, it is unpredictable whether (a) the CPU enters the 64-bit addressing mode, or (b) bit 31 of the PSW is set to one, resulting in an early specification exception being recognized. Similarly, if LOAD PSW EXTENDED does not result in an operation exception, the results are as described for LOAD PSW, above.
  - For SET ADDRESSING MODE (SAM64), it is unpredictable whether (a) the CPU enters the 64-bit addressing mode, or (b) an operation exception is recognized.
- If the program issues an instruction that is valid in both the ESA/390 and z/Architecture architectural modes, but the instruction specifies an attribute that is unique to the z/Architecture architectural mode, it is unpredictable whether an exception condition is recognized, the attribute is ignored, or the instruction executes according to its z/Architecture definition. This behavior applies as follows :
  - It is unpredictable whether the long displacement facility is installed. If installed, the long-displacement facility may apply to the following instructions:
    - ADD LOGICAL WITH CARRY (ALC)
    - COMPARE LOGICAL LONG UNICODE
    - DIVIDE LOGICAL (DL)
    - LOAD REVERSED (LRV, LRVH)
    - MOVE LONG UNICODE
    - MULTIPLY LOGICAL (ML)
    - STORE REVERSED (STRV, STRVH)
    - SUBTRACT LOGICAL WITH BORROW (SLB)
  - It is unpredictable whether the ETF3-enhancement facility is considered to be installed by the following instructions:
    - CONVERT UTF-16 TO UTF8
    - CONVERT UTF8 TO UTF-16

    If the ETF3-enhancement facility is not installed, then there is no M_3 field in bits 16-19 of the instruction, and enhanced well-formedness checking does not occur.
  - It is unpredictable whether any of the CMPSC-enhancement facility, the entropy-

encoding compression facility, and the order-preserving compression facility are considered to be installed by the COMPRESSION CALL instruction. When the facilities are not installed, their respective enablement controls are ignored.

– It is unpredictable whether any new function code added in the message-security-assist extension 3 or higher is supported by the following instructions:

  - CIPHER MESSAGE
  - CIPHER MESSAGE WITH CHAINING
  - COMPUTE INTERMEDIATE MESSAGE DIGEST
  - COMPUTE LAST MESSAGE DIGEST
  - COMPUTE MESSAGE AUTHENTICATION CODE

  If the function code is not supported, then a specification exception is recognized if the function is attempted. The query function will only indicate the availability of functions that are valid in the ESA/390 architectural mode.

– It is unpredictable whether functions of the PERFORM LOCKED OPERATION that are unique to z/Architecture operate as defined or result in a specification exception being recognized. The test operation will only indicate the availability of functions that are valid in the ESA/390 architectural mode.

– It is unpredictable whether z/Architecture-unique functions that have been added to floating-point instructions are available. Similarly, it is unpredictable whether z/Architecture unique controls and indications in the floating-point-control register are available. See “Impacts on ESA/390 and ESA/390-Compatibility Mode” on page 9-52.

– It is unpredictable whether either or both the IPTE-range facility or the local-TLB-clearing facility is considered to be installed, as used by the INVALIDATE PAGE TABLE ENTRY instruction. When the facilities are not installed, their respective enablement controls are ignored. See “INVALIDATE PAGE TABLE ENTRY” on page 10-37 for details.

- It is unpredictable whether PER instruction-fetching nullification and zero-address-detection events are recognized.
- SIGP order codes that are unique to the z/Architecture architectural mode result in an invalid-order status condition being recognized.

The ESA/390-compatibility mode is available only to a virtual machine under the control of a hypervisor program such as z/VM.

Programming-System Notes:

Sequence of Storage References

The following sections describe the effects which can be observed in storage due to overlapped operations and piecemeal execution of a CPU program. Most of the effects described in these sections are observable only when two or more CPUs or channel programs are in simultaneous execution and access common storage locations. Thus, most of the effects need be taken into account by a program only if the program interacts with another CPU or a channel program.

Some of the effects described in the following sections are independent of interaction with another CPU or a channel program. These effects, which are therefore more readily observable, relate to prefetched instructions and overlapping operands of a single instruction. These effects are described in “Conceptual Sequence” and in “Interlocks for Virtual-Storage References” on page 5-115.

Conceptual Sequence

The CPU conceptually processes instructions one at a time, with the execution of one instruction preceding the execution of the following instruction. The execution of the instruction designated by a successful branch follows the execution of the branch. Similarly, an interruption takes place between instructions or, for interruptible instructions, between units of operation of such instructions.

The sequence of events implied by the processing just described is sometimes called the conceptual sequence.

Each operation of instruction execution appears to the program itself to be performed sequentially, with the current instruction being fetched after the preceding operation is completed and before the execution of the current operation is begun. This appearance is

maintained even though the storage-implementation characteristics and overlap of instruction execution with storage accessing may cause actual processing to be different. The results generated are those that would have been obtained had the operations been performed in the conceptual sequence. Thus, it is possible for an instruction to modify the next succeeding instruction in storage.

A case in which the copies of prefetched instructions are not necessarily discarded occurs when the fetch and the store are done by means of different effective addresses that map to the same real address. This case is described in more detail in "Interlocks for Virtual-Storage References" on page 5-115.

When the transactional-execution facility is installed, the CPU may appear to deviate from the conceptual sequence to the extent that when a transaction is aborted, the following may be visible by the program:

- Although transactional stores are discarded, stores made by the NONTRANSACTIONAL STORE instruction are observable by all CPUs and by the channel subsystem.
- General registers that were altered by the transaction, but not designated to be saved by the TRANSACTION BEGIN instruction, contain modified values
- Access registers, vector registers, and floating-point registers (including the floating-point-control register) that were altered by the transaction contain modified values.
- The breaking-event-address register may contain the address of a breaking-event instruction within the aborted transaction.
- While in the nonconstrained transactional-execution mode, if a CPU makes transactional and non-transactional stores to the same storage location, and the transaction then aborts, the contents of the storage location are unpredictable.

Overlapped Operation of Instruction Execution

In simple models in which operations are not overlapped, the conceptual and actual sequences are essentially the same. However, in more complex machines, overlapped operation, buffering of operands and results, and execution times which are comparable to the propagation delays between units can cause the actual sequence to differ considerably from the conceptual sequence. Except as noted below, these machines employ special circuitry to detect dependencies between operations and ensure that the results obtained, as observed by the CPU which generates them, are those that would have been obtained if the operations had been performed in the conceptual sequence. However, other CPUs and channel programs may, unless otherwise constrained, observe a sequence that differs from the conceptual sequence.

Exceptions to the conceptual-sequence ordering of instruction execution are as follows:

- The modification of ART and DAT table entries may not have immediate effect on the conceptual sequence. See "Modification of Translation Tables" on page 3-67, "Modification of DAT-Table Entries" on page 5-27, and "Modification of ART Tables" on page 5-66 for further details.
- To improve performance, a machine may speculatively execute multiple instructions – including multiple instruction paths – in parallel, discarding results from the speculatively-executed instructions that do not represent the conceptual sequence. Speculative execution may result in the formation of ALB and TLB entries, even though such entries might not be created in the conceptual sequence. However, TLB and ALB entries are not formed as a result of, or subsequent to, the execution of following instructions unless such execution occurs in the conceptual sequence:
  - BRANCH IN SUBSPACE GROUP
  - LOAD CONTROL that alters control registers 0, 1, 2, 5, 7, 8, or 13
  - LOAD PSW
  - LOAD PSW EXTENDED
  - LOAD PAGE TABLE ENTRY ADDRESS
  - LOAD REAL ADDRESS
  - MONITOR CALL that results in a monitor-counting operation
  - MOVE WITH OPTIONAL SPECIFICATIONS
  - PROGRAM CALL
  - PROGRAM RETURN
  - PROGRAM TRANSFER (WITH INSTANCE)
  - SET ADDRESS SPACE CONTROL
  - SET ADDRESS SPACE CONTROL FAST
  - SET SECONDARY ASN (WITH INSTANCE)
  - SET SYSTEM MASK
  - STORE REAL ADDRESS

- STORE THEN AND SYSTEM MASK
- STORE THEN OR SYSTEM MASK
- TEST ACCESS

Divisible Instruction Execution

It can normally be assumed that the execution of each instruction occurs as an indivisible event. However, in actual operation, the execution of an instruction consists in a series of discrete steps. Depending on the instruction, operands may be fetched and stored in a piecemeal fashion, and some delay may occur between fetching operands and storing results. As a consequence, intermediate or partially completed results may be observable by other CPUs and by channel programs.

When a program interacts with the operation on another CPU, or with a channel program, the program may have to take into consideration that a single operation may consist in a series of storage references, that a storage reference may in turn consist in a series of accesses, and that the conceptual and observed sequences of these accesses may differ.

Storage references associated with instruction execution are of the following types: instruction fetches, ART-table and DAT-table fetches, and storage-operand references. For the purpose of describing the sequence of storage references, accesses to storage in order to perform ASN translation, PC-number translation, tracing, and the linkage-stack stacking and unstacking processes are considered to be storage-operand references.

Programming Note: The sequence of execution of a CPU may differ from the simple conceptual definition in the following ways:

- As observed by the CPU itself, instructions may appear to be prefetched when different effective addresses are used. (See "Interlocks for Virtual-Storage References".)
- As observed by other CPUs and by channel programs, the execution of an instruction may appear to be performed as a sequence of piecemeal steps. This is described for each type of storage reference in the following sections.
- As observed by other CPUs and by channel programs, the storage-operand accesses associated with one instruction are not necessarily performed in the conceptual sequence. (See "Relation between Operand Accesses" on page 5-129.)
- As observed by channel programs, in certain unusual situations, the contents of storage may appear to change and then be restored to the original value.

Interlocks for Virtual-Storage References

As described in the immediately preceding sections, CPU operation appears, with certain exceptions, to be performed sequentially as observed by the CPU itself; the stores performed by one instruction generally appear to be completed before the next instruction and its operands are fetched. This appearance is maintained in overlapped machines by means of interlock circuitry that detects accesses to a common storage location.

For those instructions which alter the contents of storage and have more than one operand, the instruction definition normally describes the results that are obtained when the operands overlap in storage, this definition being in terms of a sequence of stores and fetches. The interlock circuitry is used in determining whether operand overlap exists.

The purpose of this section is to define those cases in which the machine must appear to operate sequentially, and in which operands of a single instruction must or must not be treated as overlapping.

Proper operation is provided in part by comparing effective addresses. For the purpose of this definition, the term "effective address" means an address before translation, if any, regardless of whether the address is virtual, real, or absolute. If two effective addresses have the same value, the effective addresses are said to be the same even though one may be real or in a different address space.

The values of two virtual effective addresses do not necessarily indicate whether or not the addresses designate the same storage location. The address-translation tables may be set up so that different effective addresses map to the same real address, or so that the same effective address in different address spaces maps to different real addresses.

The interlocks for virtual-storage references are considered in two situations: storage references of one instruction as they affect storage references of another instruction, and multiple storage references of a single instruction.

Interlocks between Instructions

Except as described below, as observed by the CPU itself, the storage accesses for operands for each instruction appear to occur in the conceptual sequence independent of the effective address used. That is, the operand stores for one instruction appear to be completed before the operand fetches for the next instruction occur. For instruction fetches, the operand stores for one instruction necessarily appear to be completed before the next instruction is fetched only when the same effective address is used for the operand store and the instruction fetch.

When an instruction changes the contents of a main-storage location in which a conceptually subsequent instruction is to be executed, either directly or by means of an execute-type instruction, and when different effective addresses are used to designate that location for storing the result and fetching the instruction, the instruction may appear to be fetched before the store occurs. If an intervening operation causes the prefetched instructions to be discarded, then the updated value is recognized. A definition of when prefetched instructions must be discarded is included in “Instruction Fetching” on page 5-118.

Any change to the storage key appears to be completed before the conceptually following reference to the associated storage block is made, regardless of whether the reference to the storage location is made by means of a virtual, real, or absolute address. Analogously, any conceptually prior references to the storage block appear to be completed when the key for that block is changed or inspected.

When the CPU is in the transactional-execution mode, and an instruction changes the contents of a main-storage location from which a conceptually subsequent instruction may be fetched, the transaction may be aborted with any of abort codes 14, 15, 16, or 255 and condition code 3.

Programming Note: Depending on the model, detection of a changed storage location may include any location within the same or adjacent cache lines as that of the store, regardless of whether the changed location is actually an instruction. Because of the characteristics described above and in the preceding paragraph, self-modifying code within a transaction is strongly discouraged.

Interlocks within a Single Instruction

For those instructions which alter the contents of storage and have more than one operand, the instruction definition normally describes the results which are obtained when the operands overlap in storage. This result is normally defined in terms of the sequence of the storage accesses; that is, a portion of the results of a store-type operand must appear to be placed in storage before some portion of the other operand is fetched. This definition applies provided that the store and fetch accesses are specified by means of the same effective addresses and the same effective space designations.

When multiple address spaces are involved in the access-register mode, the term “effective space designation” is used to denote the value used by the machine to determine whether two spaces are the same. In the access-register mode, the 32-bit access-list-entry-token (ALET) value associated with each storage-operand address is called the effective space designation. When a B field of zero is specified, a value of all zeros is used for the effective space designation. If the effective space designations are different, the spaces are considered to be different even if both ALETs map to the same address-space-control-element value.

When the store and the fetch accesses are specified by means of different effective space designations or by means of different effective addresses, the operand fetch may appear to precede the operand store.

Figure 5-19 on page 5-117 summarizes the cases of overlap and the specified results, including when

MOVE LONG (MVCL) sets condition code 3, for each case.

Effective Space Designations Equal?	Effective Addresses Overlap Destructively?	Operand Overlap Destructively in Real Storage?	Is Overlap Recognized?	
			MVCL Sets CC 3	Operand Results
Yes	No	No	No	No
		Yes	No	Unpredictable
	Yes	No	–	–
		Yes	Yes	Yes
No	No	No	No	No
		Yes	No	Unpredictable
	Yes	No	No	No
		Yes	No	Unpredictable

Explanation:

– This case cannot occur.

Figure 5-19. Virtual-Storage Interlocks within a Single Instruction

Effective space designations may be represented by ALB entries, and the test for whether two effective space designations are the same may be performed by comparing ALB entries. If the program changes an attached and valid ART-table entry without subsequently causing the execution of PURGE ALB or a COMPARE AND SWAP AND PURGE instruction that purges the ALB, two effective space designations that are the same may have different representations in the ALB, and failure to recognize operand overlap may result. The use of the ALB never causes overlap to be recognized when the effective space designations are different.

Programming Note: A single main-storage location can be accessed by means of more than one address in several ways:

1. The DAT tables may be set up such that multiple addresses in a single address space, or addresses in different address spaces, including the real address specified by a real-space designation, map to a single real address.

2. The translation of logical, instruction, and virtual addresses may be changed by loading the DAT parameters in the control registers, by changing the address-space-control bits in the PSW, or, for logical and instruction addresses, by turning DAT on or off.

3. In the access-register mode, different address spaces may be selected by means of each access register. In addition, the primary address space is selected for instruction fetching and the target of an execute-type instruction.

4. STORE USING REAL ADDRESS performs a store by means of a real address.

5. Certain other instructions also use real addresses (even when a logical address is not translated by means of a real-space designation, which is a situation covered in case 1), the instructions MOVE TO PRIMARY and MOVE TO SECONDARY access two address spaces, and the instruction MOVE WITH OPTIONAL SPECIFICATIONS may access one or two address spaces.

6. Accesses to storage for the purpose of storing and fetching information for interruptions is performed by means of real addresses, and, for the store-status function, by means of absolute addresses, whereas accesses by the program may be by means of virtual addresses.

7. The real-to-absolute mapping may be changed by means of the SET PREFIX instruction or a reset.

8. A main-storage location may be accessed by channel programs by means of an absolute address and by the CPU by means of an absolute, a real, or a virtual address.

9. A main-storage location may be accessed by another CPU by means of one type of address and by this CPU by means of a different type of address.

The primary purpose of this section on interlocks is to describe the effects caused in cases 1, 3, and 4, above.

For case 2, no effect is observable because prefetched instructions are discarded when the translation parameters are changed, and the delay of stores by a CPU is not observable by the CPU itself.

For case 5, for those instructions which fetch by using real addresses (for example, LOAD REAL ADDRESS, which fetches a segment-table entry and a page-table entry, and may fetch a region-table entry), no effect is observable because only operand accesses between instructions are involved. All instructions that store by using a real address, except STORE USING REAL ADDRESS, or that store across address spaces, except in the access-register mode, cause prefetched instructions to be discarded, and no effect is observable.

Cases 6 and 7 are situations which are defined to cause serialization, with the result that prefetched instructions are discarded. In these cases, no effect is observable.

The handling of cases 8 and 9 involves accesses as observed by other CPUs and by channel programs and is covered in the following sections in this chapter.

Instruction Fetching

Instruction fetching consists in fetching the one, two, or three halfwords designated by the instruction address in the current PSW. The immediate field of an instruction is accessed as part of an instruction fetch. If, however, an instruction designates a storage operand at the location occupied by the instruction itself, the location is accessed both as an instruction and as a storage operand. The fetch of the target instruction of an execute-type instruction is considered to be an instruction fetch.

The bytes of an instruction may be fetched piecemeal and are not necessarily accessed in a left-to-right direction. The instruction may be fetched multiple times for a single execution; for example, it may be fetched for testing the addressability of operands or for inspection of PER events, and it may be refetched for actual execution.

Instructions are not necessarily fetched in the sequence in which they are conceptually executed and are not necessarily fetched each time they are executed. In particular, the fetching of an instruction may precede the storage-operand references for an instruction that is conceptually earlier. The instruction fetch occurs prior to all storage-operand references for all instructions that are conceptually later.

An instruction may be prefetched by using a virtual address only when the associated DAT table entries are attached and valid or when entries which qualify for substitution for the table entries exist in the TLB. An instruction that has been prefetched may be interpreted for execution only for the same virtual address for which the instruction was prefetched.

No limit is established on the number of instructions which may be prefetched, and multiple copies of the contents of a single storage location may be fetched. As a result, the instruction executed is not necessarily the most recently fetched copy. Storing caused by other CPUs and by channel programs does not necessarily change the copy of prefetched instructions. However, if a store that is conceptually earlier is made by the same CPU using the same effective address as that by which the instruction is subsequently fetched, the updated information is obtained, except as noted below. If the effective addresses are different, the updated information is not necessarily obtained. However, the updated information is obtained if either execution is in the real mode since prefetched instructions are discarded if DAT is turned on or off.

All copies of prefetched instructions are discarded when:

- A serializing function is performed. However, for PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, and TRACE, it is unpredictable whether or not a store into a trace-table entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store; for PROGRAM CALL and PROGRAM RETURN it is unpredictable whether or not a store into a trace-table entry or linkage-stack entry from which a subsequent instruction is fetched will be observed by

the CPU that performed the store. Additionally, when the store-clock-fast facility is installed and the TRACE TOD-clock control in bit 32 of control register 0 is one, it is unpredictable whether explicit tracing causes serialization to be performed.

- The CPU enters the operating state.
- DAT is turned on or off.
- A change is made to a translation parameter in control register 1 when in the primary-space, secondary-space, or access-register mode, or in control register 13 when in the home-space mode.

The SET ADDRESS SPACE CONTROL instruction can change the translation mode between any of the primary-space, secondary-space, access-register, and home-space modes, and it performs serialization. The SET ADDRESS SPACE CONTROL FAST instruction can perform the same mode changes, but it does not serialize.

Programming Notes:

1. As observed by a CPU itself, its own instruction prefetching may be apparent when different effective addresses map to a single real address. This is described in “Conceptual Sequence” on page 5-113 and “Interlocks for Virtual-Storage References” on page 5-115.

2. Any means of changing PSW bits 16 and 17, except the SET ADDRESS SPACE CONTROL FAST instruction, causes serialization to be performed and prefetched instructions to be discarded. Turning DAT on or off causes prefetched instructions to be discarded. Therefore, any change of the translation mode, except a change made by SET ADDRESS SPACE CONTROL FAST, always causes prefetched instructions to be discarded.

3. The following are some effects of instruction prefetching on one CPU as observed by other CPUs and by channel programs.

   It is possible for one CPU to prefetch the contents of a storage location, after which another CPU or a channel program can change the contents of that storage location and then set a flag to indicate that the change has been made. Subsequently, the first CPU can test and find the flag set, branch to the modified location, and execute the original prefetched contents.

   It is possible, if another CPU or a channel program concurrently modifies the instruction, for one CPU to recognize the changes to some but not all bit positions of an instruction.

   It is possible for one CPU to prefetch an instruction and subsequently, before the instruction is executed, for another CPU to change the storage key. As a result, the first CPU may appear to execute instructions from a protected storage location. However, the copy of the instructions executed is the copy prefetched before the location was protected.

4. Previous versions of both the *ESA/390 Principles of Operation* and the *z/Architecture Principles of Operation* have allowed a CPU to not recognize a store into the instruction stream when the CPU was operating in the home-space or access-register modes. To ensure that stores into the instruction stream were recognized in these modes, the program was required to insert a serializing operation following the store.

   No IBM processor has ever implemented this characteristic; stores into the instruction stream using the same effective address as that by which a subsequent instruction is fetched have always been recognized, regardless of the address-space-control mode. This characteristic is now removed from the architecture. Any serializing operations that were inserted to ensure recognition of stores into the instruction stream while in the AR or home-space modes may be safely removed from programs that execute on IBM processors.

ART-Table and DAT-Table Fetches

The access-register-translation (ART) table entries are access-list designations, access-list entries, ASN-second-table entries, and authority-table entries. The dynamic-address-translation (DAT) table entries are region-table entries, segment-table entries, and page-table entries. The fetching of these entries may occur as follows:

1. An ART-table entry may be prefetched into the ART-lookaside buffer (ALB) and used from the ALB without refetching from storage, until the entry is cleared by a COMPARE AND SWAP

AND PURGE, PURGE ALB, or SET PREFIX instruction, by CPU reset, or by a set prefix or set-architecture SIGNAL PROCESSOR order. A DAT-table entry may be prefetched into the translation-lookaside buffer (TLB) and used from the TLB without refetching from storage, until the entry is cleared by a COMPARE AND REPLACE DAT TABLE ENTRY, COMPARE AND SWAP AND PURGE, INVALIDATE DAT TABLE ENTRY, INVALIDATE PAGE TABLE ENTRY, PURGE TLB, or SET PREFIX instruction, by CPU reset, or by a set prefix or set-architecture SIGNAL PROCESSOR order. ART-table and DAT-table entries are not necessarily fetched in the sequence conceptually called for; they may be fetched at any time they are attached and valid, including during the execution of conceptually previous instructions.

2. The fetching of access-list designations, access-list entries, ASN-second-table entries, and DAT-table entries appears to be word concurrent as observed by other CPUs, except that the fetching of an address-space-control element from an ASN-second-table entry appears to be double-word concurrent as observed by other CPUs. The reference to an entry may appear to access a single byte at a time as observed by channel programs.

3. The order in which the words of an access-list entry or ASN-second-table entry are fetched is unpredictable, except that the leftmost word of an entry is fetched first. However, the leftmost word of an ASN-second-table entry is not fetched when access-list-entry token 00000000 hex is translated for BRANCH IN SUBSPACE GROUP.

4. An ART-table or DAT-table entry may be fetched even after some operand references for the instruction have already occurred. The fetch may occur as late as just prior to the actual byte access requiring the ART-table or DAT-table entry.

5. An ART-table or DAT-table entry may be fetched for each use of the address, including any trial execution, and for each reference to each byte of each operand.

6. The DAT page-table-entry fetch precedes the reference to the page. When no copy of the page-table entry is in the TLB, the fetch of the associated segment-table entry precedes the fetch of the page-table entry. When no copy of the segment-table entry is in the TLB, the fetch of the region-third-table entry, if one is required, precedes the fetch of the segment-table entry. Similarly, the fetch of a required region-second-table entry precedes the fetch of the region-first-table entry, and the fetch of a required region-first-table entry precedes the fetch of the region-second-table entry.

7. When no copy of a region-table entry or segment-table entry designated by means of an ART-obtained address-space-control element is in the TLB, the ART fetch of the ASN-second-table entry precedes the DAT region-table-entry or segment-table-entry fetch. When no copy of a required authority-table entry is in the ALB, the ART fetch of the associated ASN-second-table entry precedes the fetch of the authority-table entry. When no copy of a required ASN-second-table entry is in the ALB, the fetch of the associated access-list entry precedes the fetch of the ASN-second-table entry. When no copy of a required access-list entry is in the ALB, the fetch of the associated access-list designation precedes the fetch of the access-list entry.

Storage-Key Accesses

In the following sections, the term *key-setting instruction* refers either to the SET STORAGE KEY EXTENDED instruction or to the PERFORM FRAME MANAGEMENT FUNCTION instruction when the set-key control is one. When the nonquiescing key-setting facility is installed, a *nonquiescing key-setting instruction* refers to a SET STORAGE KEY EXTENDED instruction in which the nonquiescing control is set to one and also refers to a PERFORM FRAME MANAGEMENT FUNCTION instruction in which the set-key control is set to one. (For PFMF, it is unpredictable whether a quiescing operation is performed when the nonquiescing key-setting facility is installed; thus in the following discussion, PFMF may be considered to be either quiescing or nonquiescing.)

References to the storage key are handled as follows:

1. Whenever a reference to storage is made and key-controlled protection applies to the reference, the four access-control bits and the fetch-protection bit associated with the storage location are inspected concurrently. The inspection of

the access-control and fetch-protection bits occurs concurrently with the reference to the storage location, except that when a nonquiescing key-setting operation is performed for a storage location by one CPU, the following applies:

- The inspection of the access-control and fetch-protection bits for the storage location by any CPU may precede a store reference to the location. In the case where the inspection of the access-control and fetch-protection bits precedes the store, the inspection occurs no earlier than after the last serialization operation on the CPU, and no earlier than the last quiescing key-setting operation for the same storage location on any CPU.
- When a unit of operation or instruction execution on any other CPU causes multiple accesses to the same 4 K-byte block as that of the key-setting operation, the other CPU necessarily inspects the access-control and fetch-protection bits only for the first reference to the block. The other CPU does not necessarily inspect the access-control and fetch-protection bits for the subsequent accesses within the block by the same unit of operation.

When (a) EDAT-1 does not apply,[1] (b) EDAT-1 applies but the storage is accessed by means of a segment-table entry in which the STE-format control is zero, (c) EDAT-1 applies, the storage is accessed by means of a segment-table entry in which the STE-format control is one, but the ACCF-validity control is zero, or (d) EDAT-2 applies, the storage is accessed by means of a region-third-table entry in which the RTTE-format control is one, but the ACCF-validity control is zero, the access-control bits and the fetch-protection bit are in bits 0-4 of the storage key for the 4 K-byte block.

When EDAT-1 applies and the storage is accessed by means of a segment-table entry in which both the STE-format control and ACCF-validity control are one, it is unpredictable whether bits 0-4 of the storage key or bits 48-52 of the segment-table entry provide the access-control bits and fetch-protection bit. Furthermore, when the segment-table entry provides the access-control bits and fetch-protection bit, a buffered copy from the translation-lookaside buffer may be used.

When EDAT-2 applies and the storage is accessed by means of a region-third-table entry in which both the RTTE-format control and ACCF-validity control are one, it is unpredictable whether bits 0-4 of the storage key or bits 48-52 of the region-third-table entry provide the access-control bits and fetch-protection bit. Furthermore, when the region-third-table entry provides the access-control bits and fetch-protection bit, a buffered copy from the translation-lookaside buffer may be used.

For accesses made by the channel subsystem, the access-control bits and fetch-protection bits are in bits 0-4 of the storage key for the 4 K-byte block.

2. When storing is performed by a CPU, the change bit is set to one in the associated storage key concurrently with the completion of the store access, as observed by the CPU itself. When storing is performed by a CPU or a channel program, the change bit is set to one in the associated storage key either before or after the completion of the store access, as observed by other (if the store was performed by a CPU) or all (if the store was performed by a channel program) CPUs. As observed by other or all CPUs, the change bit is set no earlier than (1) after the last serialization function performed previously by the CPU or channel program performing the store, and (2) after the execution, by any CPU in the configuration, of a quiescing key-setting instruction that last set the associated storage key before the completion of the store. As observed by other or all CPUs, a change-bit setting necessarily occurs only when any of the following occurs subsequent to the storing operation:
   - The CPU or channel program that performed the store performs a serialization function.
   - The store was performed by a CPU or a channel program, and any CPU in the configuration sets the subject change bit by executing a key-setting instruction after the store access is completed. The change-bit setting due to the store access occurs before the setting by the key-setting instruction, except that when the nonquiescing key-setting facility is installed, the change bit may appear to be set following a nonquiescing key-setting instruction.

1. See “Enhanced-DAT Terminology:” on page 3-41 for an explanation of EDAT applicability.

- The store was performed by a CPU and is or will be completed, and any CPU in the configuration executes a COMPARE AND REPLACE DAT TABLE ENTRY, COMPARE AND SWAP AND PURGE, INVALIDATE DAT TABLE ENTRY, or INVALIDATE PAGE TABLE ENTRY instruction that clears from the ALB or TLB of the storing CPU any entry used to complete the store. Completion of the clearing instruction is delayed until the subject store and change-bit setting have been completed.
- The store was performed by a CPU, and that CPU examines the subject change bit by means of an INSERT STORAGE KEY EXTENDED or RESET REFERENCE BIT EXTENDED instruction. See "Relation between Operand Accesses" on page 5-129.

3. The following discussion assumes that other CPUs are not simultaneously executing nonquiescing key-setting operations for the same storage key. See the programming note below.

   a. When the conditional-SSKE facility is not installed, the SET STORAGE KEY EXTENDED instruction causes all seven bits to be set concurrently in the storage key. When the conditional-SSKE facility is installed, SET STORAGE KEY EXTENDED may be used to set all or portions of the storage key based on program-specified criteria. The access to the storage key for SET STORAGE KEY EXTENDED follows the sequence rules for storage-operand store references and is a single-access reference. When the enhanced-DAT facility is installed, SET STORAGE KEY EXTENDED or PERFORM FRAME MANAGEMENT FUNCTION may be used to set all or portions of one or more storage keys based on program-specified criteria.

   b. The INSERT STORAGE KEY EXTENDED instruction provides a consistent image of bits 0-6 of the storage key for a 4 K-byte block. Similarly, the instructions INSERT VIRTUAL STORAGE KEY and TEST PROTECTION provide a consistent image of the access-control bits and the fetch-protection bit. The access to the storage key for all of these instructions follows the sequence rules for storage-operand fetch references and is a single-access reference.

   c. The instruction RESET REFERENCE BIT EXTENDED modifies only the reference bit. All other bits of the storage key remain unchanged. The reference bit and change bit are inspected concurrently to set the condition code. The fetch and store accesses to the storage key for RESET REFERENCE BIT EXTENDED follow the sequence rules for storage-operand fetch and store references, respectively, and are single-access references.

   d. For each of the 64 storage keys accessed, the instruction RESET REFERENCE BITS MULTIPLE modifies only the reference bit. All other bits of the storage key remain unchanged. The fetch and store accesses to each of the individual storage keys for RESET REFERENCE BITS MULTIPLE follow the sequence rules for storage-operand fetch and store references, respectively, and are single-access references.

The record of references provided by the reference bit is not necessarily accurate. However, in the majority of situations, reference recording approximately coincides with the related storage reference.

The change bit may be set in cases when no storing has occurred. See "Exceptions to Nullification and Suppression" on page 5-26.

Programming Note: When a nonquiescing key-setting instruction is executed on one CPU to set the storage key of a block, the program should ensure that no other CPU or I/O subsystem is simultaneously referencing the storage block in which the key is being set; similarly, the program should ensure that no other CPU is attempting to set the storage key of the block. In a virtual storage environment, the program can ensure that other CPUs do not access the block using a virtual address by simply not mapping the block to any virtual address.

Failure to comply with this restriction may result in either of the following:

- An unpredictable storage key for the block may be used in the enforcement of key-controlled protection.

- An unpredictable storage key for the block may be inspected by key-inspecting instructions such as IRBM, ISKE, IVSK, RRBE, and RRBM.

Storage-Operand References

A storage-operand reference is the fetching or storing of the explicit operand or operands in the storage locations designated by the instruction.

During the execution of an instruction, all or some of the storage operands for that instruction may be fetched, intermediate results may be maintained for subsequent modification, and final results may be temporarily held prior to placing them in storage. Stores caused by other CPUs and by channel programs do not necessarily affect these intermediate results.

Storage-operand references are of three types: fetches, stores, and updates.

Storage-Operand Fetch References

When the bytes of a storage operand participate in the instruction execution only as a source, the operand is called a fetch-type operand, and the reference to the location is called a storage-operand fetch reference. A fetch-type operand is identified in individual instruction definitions by indicating that the access exception is for fetch.

All bits within a single byte of a fetch-type operand are accessed concurrently. When an operand consists of more than one byte, the bytes may be fetched from storage piecemeal, one byte at a time. Unless otherwise specified, the bytes are not necessarily fetched in any particular sequence.

The storage-operand fetch references of one instruction occur after those of all preceding instructions and before those of subsequent instructions, as observed by other CPUs and by channel programs. The operands of any one instruction are fetched in the sequence specified for that instruction. The CPU may fetch the operands of instructions before the instructions are executed. There is no defined limit on the length of time between when an operand is fetched and when it is used. Still, as observed by the CPU itself, its storage-operand references are performed in the conceptual sequence.

For certain special instructions, the fetch references for multiple operands may appear to be interlocked against certain accesses by other CPUs and by channel programs. Such a fetch reference is called an interlocked-fetch reference. The fetch accesses associated with an interlocked-fetch reference do not necessarily occur one immediately after the other, but store accesses by other CPUs may not occur at the same locations as the interlocked-fetch reference between the fetch accesses of the interlocked-fetch reference.

The storage-operand fetch reference for the LOAD PAIR DISJOINT instruction may be an interlocked-fetch reference. Whether or not LOAD PAIR DISJOINT is able to fetch both operands by means of an interlocked fetch is indicated by the condition code.

Storage-Operand Store References

When the bytes of a storage operand participate in the instruction execution only as a destination, to the extent of being replaced by the result, the operand is called a store-type operand, and the reference to the location is called a storage-operand store reference.

A store-type operand is identified in individual instruction definitions by indicating that the access exception is for store.

All bits within a single byte of a store-type operand are accessed concurrently. When an operand consists of more than one byte, the bytes may be placed in storage piecemeal, one byte at a time. Unless otherwise specified, the bytes are not necessarily stored in any particular sequence.

The CPU may delay placing results in storage. There is no defined limit on the length of time that results may remain pending before they are stored. This delay does not affect the sequence in which results are placed in storage.

With the exceptions noted below, the results of one instruction are placed in storage after the results of all preceding instructions have been placed in storage and before any results of the succeeding instructions are stored, as observed by other CPUs and by the channel subsystem. However, the following exception cases exist:

- When a transaction aborts, only the results of non transactional stores are committed to storage; transactional stores are discarded.

The results of any one instruction are stored in the sequence specified for that instruction.

The CPU does not fetch operands, ART-table entries, or DAT-table entries from a storage location until all information destined for that location by the CPU has been stored. Prefetched instructions may appear to be updated before the information appears in storage.

The stores are necessarily completed only as a result of a serializing operation and before the CPU enters the stopped state.

Storage-Operand Update References

In some instructions, the storage-operand location participates both as a source and as a destination. In these cases, the reference to the location consists first in a fetch and subsequently in a store. The operand is called an update-type operand, and the combination of the two accesses is referred to as an update reference. Instructions such as MOVE ZONES, TRANSLATE, OR (OC, OI), and ADD DECIMAL cause an update to the first-operand location. An update-type operand is identified in the individual instruction definition by indicating that the access exception is for both fetch and store.

Noninterlocked-Update References: For most instructions which have update-type operands, the fetch and store accesses associated with an update reference do not necessarily occur one immediately after the other, and it is possible for other CPUs and channel programs to make fetch and store accesses to the same location during this time. Such an update reference is sometimes called a noninterlocked-update storage reference.

Interlocked-Update References: For certain special instructions, the update reference is interlocked against certain accesses by other CPUs and channel programs. Such an update reference is called an interlocked-update reference. The fetch and store accesses associated with an interlocked-update reference do not necessarily occur one immediately after the other, but all store accesses by other CPUs and channel programs and the fetch and store accesses associated with interlocked-update references by other CPUs are prevented from occurring at the same location between the fetch and the store accesses of an interlocked-update reference.

The storage-operand update reference for the following instructions appears to be an interlocked-update reference as observed by other CPUs and channel programs.

- ADD IMMEDIATE (ASI and AGSI), when the interlocked-access facility 1 is installed and the first operand is aligned on an integral boundary corresponding to its size
- ADD LOGICAL WITH SIGNED IMMEDIATE, when the interlocked-access facility 1 is installed and the first operand is aligned on an integral boundary corresponding to its size
- AND (NI and NIY), when the interlocked-access facility 2 is installed
- COMPARE AND REPLACE DAT TABLE ENTRY
- COMPARE AND SWAP
- COMPARE AND SWAP AND PURGE
- COMPARE AND SWAP AND STORE
- COMPARE DOUBLE AND SWAP
- EXCLUSIVE OR (XI and XIY), when the interlocked-access facility 2 is installed
- LOAD AND ADD
- LOAD AND ADD LOGICAL
- LOAD AND AND
- LOAD AND EXCLUSIVE OR
- LOAD AND OR
- OR (OI and OIY), when the interlocked-access facility 2 is installed
- STORE CHARACTERS UNDER MASK, on models in which the instruction with a mask of zero fetches and stores the byte designated by the second-operand address
- TEST AND SET

Within the limitations of the above requirements, the fetch and store accesses associated with an update reference follow the same rules as the fetches and stores described in the previous sections.

Programming Notes:

1. When two CPUs attempt to update information at a common main-storage location by means of a noninterlocked-update reference, it is possible for both CPUs to fetch the information and subsequently make the store access. The change made by the first CPU to store the result in such a case is lost. Similarly, if one CPU updates the contents of a field by means of a noninterlocked-update reference, but another CPU makes a store access to that field between the fetch and store parts of the update reference, the effect of the store is lost. If, instead of a store access, a

CPU makes an interlocked-update reference to the common storage field between the fetch and store portions of a noninterlocked-update reference due to another CPU, any change in the contents produced by the interlocked-update reference is lost.

2. Except for STORE CHARACTERS UNDER MASK, the instructions listed in this section (~~"Interlocked-Update References"~~) facilitate updating of a common storage field by two or more CPUs. To ensure that no changes are lost, all CPUs must use an instruction providing an interlocked-update reference.

3. Only those bytes which are included in the result field of both operations are considered to be part of the common main-storage location. However, all bits within a common byte are considered to be common even if the bits modified by the two operations do not overlap. As an example, if (1) one CPU executes the instruction OR (OC) with a length of 1 and the value 80 hex in the second-operand location, (2) the other CPU executes AND (NC) with a length of 1 and the value FE hex in the second-operand location, and (3) the first operand of both instructions is the same byte, then the result of one of the updates can be lost. This is because the updates to the byte by both NC and OC are not interlocked-update references.

4. When the store access is part of an update reference by the CPU, the execution of the storing is not necessarily contingent on whether the information to be stored is different from the original contents of the location. In particular, the contents of all designated byte locations are replaced, and, for each byte in the field, the entire contents of the byte are replaced.

   Depending on the model, an access to store information may be performed, for example, in the following cases:

   a. Execution of the OR instruction (OI, OIY, or OC) with a second operand of all zeros.

   b. Execution of OR (OC) with the first-and second-operand fields coinciding.

   c. For those locations of the first operand of TRANSLATE where the argument and function values are the same.

Storage-Operand Consistency

Single-Access References

A fetch reference is said to be a single-access reference if the value is fetched in a single access to each byte of the data field. In the case of overlapping operands, the location may be accessed once for each operand. A store-type reference is said to be a single-access reference if a single store access occurs to each byte location within the data field. An update reference is said to be single access if both the fetch and store accesses are each single access.

Except for the accesses associated with multiple-access references and the stores associated with storage change and restoration for DAT-associated access exceptions, all storage-operand references are single-access references.

Multiple-Access References

In some cases, multiple accesses may be made to all or some of the bytes of a storage operand. The following cases may involve multiple-access references:

1. The storage operands of the following instructions:

   - CHECKSUM
   - CIPHER MESSAGE
   - CIPHER MESSAGE WITH AUTHENTICATION
   - CIPHER MESSAGE WITH CIPHER FEEDBACK
   - CIPHER MESSAGE WITH CHAINING
   - CIPHER MESSAGE WITH COUNTER
   - CIPHER MESSAGE WITH OUTPUT FEEDBACK
   - COMPARE AND FORM CODEWORD
   - COMPARE UNTIL SUBSTRING EQUAL
   - COMPUTE INTERMEDIATE MESSAGE DIGEST
   - COMPUTE LAST MESSAGE DIGEST
   - COMPUTE MESSAGE AUTHENTICATION CODE
   - CONVERT FROM PACKED
   - CONVERT FROM ZONED
   - CONVERT TO BINARY
   - CONVERT TO DECIMAL
   - CONVERT TO PACKED
   - CONVERT TO ZONED
   - CONVERT UTF-16 TO UTF-32
   - CONVERT UTF-16 TO UTF-8

- CONVERT UTF-32 TO UTF-16
- CONVERT UTF-32 TO UTF-8
- CONVERT UTF-8 TO UTF-16
- CONVERT UTF-8 TO UTF-32
- LOAD ADDRESS SPACE PARAMETERS
- LOAD REVERSED
- MOVE INVERSE
- MOVE LONG UNICODE
- MOVE PAGE
- MOVE WITH OFFSET
- PACK
- PACK ASCII
- PACK UNICODE
- PERFORM CRYPTOGRAPHIC COMPUTATION
- PERFORM CRYPTOGRAPHIC KEY MANAGEMENT OPERATION
- PERFORM FRAME MANAGEMENT FUNCTION, when the clear-frame control is one
- PERFORM RANDOM NUMBER OPERATION
- RESUME PROGRAM
- STORE REVERSED
- STORE SYSTEM INFORMATION
- TEST BLOCK
- TRANSLATE
- TRANSLATE AND TEST
- TRANSLATE AND TEST EXTENDED
- TRANSLATE AND TEST REVERSE
- TRANSLATE AND TEST REVERSE EXTENDED
- TRANSLATE EXTENDED
- TRANSLATE ONE TO ONE
- TRANSLATE ONE TO TWO
- TRANSLATE TWO TO ONE
- TRANSLATE TWO TO TWO
- UNPACK
- UNPACK ASCII
- UNPACK UNICODE
- UPDATE TREE

2. The storage operands of MOVE LONG and MOVE LONG EXTENDED, when the padding character is not B1 hex.

3. The stores into that portion of the first operand of MOVE LONG, MOVE LONG EXTENDED, or MOVE LONG UNICODE which is filled with padding bytes.

4. The storage operands of the decimal instructions.

5. The main-storage operands of PAGE IN and PAGE OUT.

6. The storage operands of the I/O instructions.

7. The stores into a trace entry.

8. The stores associated with the stop-and-store-status and store-status-at-address SIGNAL PROCESSOR orders.

9. The trap control block and trap save area used by TRAP.

10. The operands, dictionaries, and symbol-translation table of COMPRESSION CALL.

When a storage-operand store reference to a location is not a single-access reference, the value placed at a byte location is not necessarily the same for each store access; thus, intermediate results in a single-byte location may be observed by other CPUs and by channel programs.

Programming Notes:

1. When multiple fetch, store, or update accesses are made to a single byte that is being changed by another CPU or by a channel program, the result is not necessarily limited to that which could be obtained by fetching or storing the bits individually. For example, the execution of MULTIPLY DECIMAL may consist in repetitive additions and subtractions, each of which causes the second operand to be fetched from storage and the first operand to be updated in storage.

2. When CPU instructions which make multiple-access references are used to modify storage locations being simultaneously accessed by another CPU or by a channel program, multiple store accesses to a single byte by the CPU may result in intermediate values being observed by the other CPU or by the channel program. To avoid these intermediate values (for example, when modifying a CCW chain), only instructions making single-access references should be used.

3. An instruction fetch, including the fetch of the target of an execute-type instruction, is a multiple-access reference.

Block-Concurrent References

For some references, the accesses to all bytes within a halfword, word, doubleword, or quadword are specified to appear to be block concurrent as observed by other CPUs and channel programs. The halfword,

word, doubleword, or quadword is referred to in this section as a block. When a fetch-type reference is specified to appear to be concurrent within a block, no store access to the block by another CPU or channel program is permitted during the time that bytes contained in the block are being fetched. When a store-type reference is specified to appear to be concurrent within a block, no access to the block, either fetch or store, is permitted by another CPU or channel program during the time that the bytes within the block are being stored.

For all instructions in the RX, RXE, RXY, S, SIL, SIY, VRV, and VRX formats, with the exception of CONVERT TO DECIMAL, CONVERT TO BINARY, LOAD PSW EXTENDED, LOAD REVERSED, RESUME PROGRAM, STORE CLOCK EXTENDED, STORE REVERSED, STORE SYSTEM INFORMATION, TRAP, VECTOR LOAD, VECTOR LOAD TO BLOCK BOUNDARY, VECTOR STORE, and the I/O instructions, when the operand is addressed on a boundary which is integral to the size of the operand, the storage-operand references appear to be block concurrent as observed by other CPUs. For LOAD PSW EXTENDED, the accesses to each of the two doublewords of the storage operand appear to be doubleword concurrent as observed by other CPUs.

For the instructions VECTOR LOAD, VECTOR LOAD MULTIPLE, VECTOR LOAD TO BLOCK BOUNDARY, VECTOR STORE, and VECTOR STORE MULTIPLE, when the operand is addressed on an integral block boundary up to eight bytes, the storage-operand references appear to be block concurrent with respect to that integral block boundary. Note that integral boundaries greater than eight bytes are by definition on an integral 8 byte boundary.

For VECTOR LOAD WITH LENGTH, VECTOR LOAD RIGHTMOST WITH LENGTH, VECTOR STORE WITH LENGTH, and VECTOR STORE RIGHTMOST WITH LENGTH, when the operand is addressed on an integral boundary up to eight bytes and the number of bytes accessed is a multiple of the integral boundary, the storage-operand references appear to be block concurrent with respect to that integral block boundary. If the number of bytes accessed is less than the integral boundary, the storage-operand references appear to be block concurrent with an integral boundary that is the greatest common power of two divisor between the storage-operand address and the number of bytes accessed. For example, if the storage-operand address is on a doubleword boundary and the number of storage-operand bytes is a multiple of two, the storage-operand references will appear halfword concurrent as observed by other CPUs.

For the instructions COMPARE AND SWAP, COMPARE AND SWAP AND PURGE, COMPARE DOUBLE AND SWAP, COMPARE HALFWORD RELATIVE LONG, COMPARE LOGICAL RELATIVE LONG, COMPARE RELATIVE LONG, LOAD AND ADD, LOAD AND ADD LOGICAL, LOAD AND AND, LOAD AND EXCLUSIVE OR, LOAD AND OR, LOAD HALFWORD RELATIVE LONG, LOAD LOGICAL HALFWORD RELATIVE LONG, LOAD LOGICAL RELATIVE LONG, LOAD PAIR DISJOINT, LOAD RELATIVE LONG, STORE HALFWORD RELATIVE LONG, and STORE RELATIVE LONG, all accesses to the storage operand appear to be block concurrent as observed by other CPUs. For COMPARE AND SWAP AND STORE, all accesses to the first operand and the store of the second operand appear to be block concurrent as observed by other CPUs.

For the instruction PERFORM LOCKED OPERATION, the accesses to the even-numbered storage operands appear to be word concurrent, as observed by other CPUs, for function codes that are a multiple of 4 and appear to be doubleword concurrent, as observed by other CPUs, for function codes that are one, 2, or 3 more than a multiple of 4. The accesses to the doublewords in the parameter list appear to be doubleword concurrent, as observed by other CPUs, regardless of the function code.

The instructions LOAD MULTIPLE (LM), LOAD MULTIPLE DISJOINT, LOAD MULTIPLE HIGH, STORE MULTIPLE (STM), and STORE MULTIPLE HIGH, when the operand or operands start on a word boundary; the instructions LOAD MULTIPLE (LMG) and STORE MULTIPLE (STMG), when the operand starts on a doubleword boundary; and the instructions COMPARE LOGICAL (CLC), COMPARE LOGICAL CHARACTERS UNDER MASK, INSERT CHARACTERS UNDER MASK, LOAD CONTROL (LCTLG), STORE CHARACTERS UNDER MASK, and STORE CONTROL (STCTG) access their storage operands in a left-to-right direction, and all bytes accessed within each doubleword appear to be accessed concurrently as observed by other CPUs.

The instructions VECTOR LOAD MULTIPLE and VECTOR STORE MULTIPLE, when the operands start on a quadword boundary accesses to their storage operands appear to be doubleword concurrent,

as observed by other CPUs. The doublewords may be accessed in any order.

When any operand of EXTRACT CPU TIME starts on a doubleword boundary, all eight bytes within the operand appear to be accessed concurrently as observed by other CPUs.

The instructions LOAD ACCESS MULTIPLE, LOAD CONTROL (LCTL), STORE ACCESS MULTIPLE, and STORE CONTROL (STCTL) access the storage operand in a left-to-right direction, and all bytes accessed within each word appear to be accessed concurrently as observed by other CPUs.

When destructive overlap does not exist, the operands of MOVE (MVC), MOVE WITH KEY, MOVE TO PRIMARY, and MOVE TO SECONDARY are accessed as follows:

1. The first operand is accessed in a left-to-right direction, and all bytes accessed within a doubleword appear to be accessed concurrently as observed by other CPUs.
2. The second operand is accessed in a left-to-right direction, and all bytes within a doubleword in the second operand that are moved into a single doubleword in the first operand appear to be fetched concurrently as observed by other CPUs. Thus, if the first and second operands begin on the same byte offset within a doubleword, the fetch of the second operand appears to be doubleword concurrent as observed by other CPUs. If the offsets within a doubleword differ by 4, the fetch of the second operand appears to be word concurrent as observed by other CPUs.

Destructive overlap is said to exist when the result location is used as a source after the result has been stored, assuming processing to be performed one byte at a time.

The operands of MOVE WITH SOURCE KEY, MOVE WITH DESTINATION KEY, and MOVE STRING are accessed the same as those of MOVE (MVC), except that destructive overlap is assumed not to exist.

Except as noted in the individual instruction descriptions, accesses to operands of MOVE LONG, MOVE LONG EXTENDED, and MOVE LONG UNICODE do not necessarily appear to occur in a left-to-right direction as observed by other CPUs and by channel programs. The operands of these instructions do appear to be accessed doubleword concurrent, as observed by other CPUs, when all of the following are true:

- Both operands start on doubleword boundaries and are an integral number of doublewords in length.
- The operands do not overlap.
- The nonpadding part of the operation is being executed.

The operands of COMPARE LOGICAL LONG, COMPARE LOGICAL LONG EXTENDED, and COMPARE LOGICAL LONG UNICODE appear to be accessed doubleword concurrent, as observed by other CPUs, when both operands start on doubleword boundaries and are an integral number of doublewords in length.

The operands of COMPARE LOGICAL STRING appear to be accessed doubleword concurrent, as observed by other CPUs, when both operands start on doubleword boundaries. The operands of SEARCH STRING and SEARCH STRING UNICODE appear to be accessed doubleword concurrent, as observed by other CPUs, when it starts on a doubleword boundary.

For EXCLUSIVE OR (XC), the operands are processed in a left-to-right direction, and, when the first and second operands coincide, all bytes accessed within a doubleword appear to be accessed concurrently as observed by other CPUs.

Programming Note: In the case of EXCLUSIVE OR (XC) designating operands which coincide exactly, the bytes within the field may appear to be accessed as many as three times, by two fetches and one store: once as the fetch portion of the first operand update, once as the second-operand fetch, and then once as the store portion of the first-operand update. Each of the three accesses appears to be doubleword concurrent as observed by other CPUs, but the three accesses do not necessarily appear to occur one immediately after the other. One or both fetch accesses may be omitted since the instruction can be completed without fetching the operands.

Relation between Operand Accesses

As observed by other CPUs and by channel programs, storage-operand fetches associated with one instruction execution appear to precede all storage-operand references for conceptually subsequent instructions. A storage-operand store specified by one instruction appears to precede all storage-operand stores specified by conceptually subsequent instructions, but it does not necessarily precede storage-operand fetches specified by conceptually subsequent instructions. However, a storage-operand store appears to precede a conceptually subsequent storage-operand fetch from the same main-storage location.

When an instruction has two storage operands both of which cause fetch references, it is unpredictable which operand is fetched first, or how much of one operand is fetched before the other operand is fetched. When the two operands overlap, the common locations may be fetched independently for each operand.

When an instruction has two storage operands the first of which causes a store and the second a fetch reference, it is unpredictable how much of the second operand is fetched before the results are stored. In the case of destructively overlapping operands, the portion of the second operand which is common to the first is not necessarily fetched from storage.

When an instruction has two storage operands the first of which causes an update reference and the second a fetch reference, it is unpredictable which operand is fetched first, or how much of one operand is fetched before the other operand is fetched. Similarly, it is unpredictable how much of the result is processed before it is returned to storage. In the case of destructively overlapping operands, the portion of the second operand which is common to the first is not necessarily fetched from storage.

The independent fetching of a single location for each of two operands may affect the program execution in the following situation. When the same storage location is designated by two operand addresses of an instruction, and another CPU or a channel program causes the contents of the location to change during execution of the instruction, the old and new values of the location may be used simultaneously. For example, comparison of a field to itself may yield a result other than equal, or EXCLUSIVE-ORing of a field with itself may yield a result other than zero.

Storage Operand References in the Transactional-Execution Mode

While the CPU is in the transactional-execution mode, the normal rules for instruction, ART and DAT table, and operand fetching apply, except where explicitly noted otherwise. This section describes additional rules for operand storage accesses. Instruction and ART and DAT table fetching are not impacted by transactional execution.

As observed by other CPUs and by the channel subsystem, when the CPU is in the transactional-execution mode, transactional and nontransactional stores are committed when the transaction ends. When a transaction is aborted all transactional stores are discarded; only nontransactional stores made by the transaction are committed when a transaction is aborted.

As observed by other CPUs and by the channel subsystem, all storage-operand fetches and stores made while a CPU is in the transactional-execution mode appear to be a single block-concurrent access. Storage accesses by other CPUs or by the channel subsystem may prevent a transaction from maintaining block-concurrent access, in which case the transaction aborts with either store or fetch conflict.

Programming Note: Storage accesses made while the CPU is in the transactional-execution mode are said to be block concurrent, even though the references may not represent a contiguous block of storage on an integral boundary.

Other Storage References

The restart, program, supervisor-call, external, input/output, and machine-check PSWs appear to be accessed doubleword concurrent as observed by other CPUs. These references appear to occur after the conceptually previous unit of operation and before the conceptually subsequent unit of operation. The relationship between the new-PSW fetch, the old-PSW store, and the interruption-code store is unpredictable.

Store accesses for interruption codes are not necessarily single-access stores. The store accesses for

the external and supervisor-call-interruption codes appear to occur between the conceptually previous and conceptually subsequent operations. The store accesses for the program-interruption codes may precede the storage-operand references associated with the instruction which results in the program interruption.

Relation between Storage-Key Accesses

As observed by other CPUs, storage-key fetches and stores due to instructions that explicitly manipulate a storage key (INSERT REFERENCE BITS MULTIPLE, INSERT STORAGE KEY EXTENDED, INSERT VIRTUAL STORAGE KEY, PERFORM FRAME MANAGEMENT FUNCTION [when SK is one], RESET REFERENCE BIT EXTENDED, and SET STORAGE KEY EXTENDED) are ordered among themselves and among storage-operand references as if the storage-key accesses were themselves storage-operand fetches and stores, respectively.

Accesses of the access-control and fetch-protection bits due to storage-operand references are concurrent with the references. When EDAT-1 applies and a storage-operand reference is made using a segment-table entry in which the STE-format and ACCF-validity controls are both one, the source of the access-control and fetch-protection bits may be any of the following: (a) the storage key for the corresponding 4 K-byte block, (b) the access-control bits and the fetch-protection bit in the segment-table entry, or (c) a buffered copy of these bits in the TLB; it is unpredictable which source is used. Similarly, when EDAT-2 applies and a storage-operand reference is made using a region-third-table entry in which the RTTE-format and ACCF-validity controls are both one, the source of the access-control and fetch-protection bits may be any of the following: (a) the storage key for the corresponding 4 K-byte block, (b) the access-control bits and the fetch-protection bit in the region-third-table entry, or (c) a buffered copy of these bits in the TLB; it is unpredictable which source is used.

Accesses of the reference and change bits due to storage-operand references are in no particular order within the interval in which they are defined to occur. (See the description of when the change bit is set in “Storage-Key Accesses” on page 5-120.) However, whether due to an instruction that explicitly manipulates a storage key or due to a storage-operand reference, a storage-key store appears to precede a conceptually subsequent storage-key fetch from the same storage key.

Serialization

The sequence of functions performed by a CPU is normally independent of the functions performed by other CPUs and by channel programs. Similarly, the sequence of functions performed by a channel program is normally independent of the functions performed by other channel programs and by CPUs. However, at certain points in its execution, serialization of the CPU occurs. Serialization also occurs at certain points for channel programs.

CPU Serialization

All interruptions, entering or leaving the transactional-execution mode, and the execution of certain instructions cause a serialization of CPU operations. A serialization operation consists in completing all conceptually previous storage accesses and related reference-bit and change-bit settings by the CPU, as observed by other CPUs and by the channel subsystem, before the conceptually subsequent storage accesses and related reference-bit and change-bit settings occur. Serialization affects the sequence of all CPU accesses to storage and to the storage keys, except for those associated with ART-table-entry and DAT-table-entry fetching.

As observed by all CPUs and by the channel subsystem, a serializing operation performed while a CPU is in the transactional-execution mode occurs when the CPU leaves the transactional-execution mode, as a result of any of the following: a TRANSACTION END instruction that decrements the transaction nesting depth to zero (normal ending), or a transaction-abort condition.

Serialization is performed by CPU reset, all interruptions, the entering or leaving of the transactional-execution mode, and the execution of the following instructions:

- The general instruction BRANCH ON CONDITION (BCR) with the M_1 and R_2 field containing 1111 binary and 0000 binary, respectively.

- When the fast-BCR-serialization facility is installed, the general instruction BRANCH ON CONDITION (BCR) with the M_1 and R_2 fields containing 1110 binary and 0000 binary, respectively.
- The general instructions COMPARE AND SWAP, COMPARE AND SWAP AND STORE, COMPARE DOUBLE AND SWAP, STORE CLOCK, STORE CLOCK EXTENDED, SUPERVISOR CALL, and TEST AND SET.
- COMPARE AND SWAP AND PURGE, which can also cause the ALB and the TLB to be cleared of all entries on all CPUs.
- COMPARE AND REPLACE DAT TABLE ENTRY, INVALIDATE DAT TABLE ENTRY, and INVALIDATE PAGE TABLE ENTRY.
- All I/O instructions.
- LOAD PSW, LOAD PSW EXTENDED, PERFORM FRAME MANAGEMENT FUNCTION, and SET STORAGE KEY EXTENDED.
- MOVE TO PRIMARY, MOVE TO SECONDARY, and SET ADDRESS SPACE CONTROL.
- PAGE IN and PAGE OUT.
- PERFORM LOCKED OPERATION. A serialization function is performed immediately after the lock is obtained and again immediately before it is released. However, values fetched from the parameter list before the lock is obtained are not necessarily refetched.
- PROGRAM CALL, and, when the state entry to be unstacked is a program-call state entry, PROGRAM RETURN. However, it is unpredictable whether or not a store into a trace-table entry or linkage-stack entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store.
- PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, and SET SECONDARY ASN WITH INSTANCE, and TRACE. However, it is unpredictable whether or not a store into a trace-table entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store. Additionally, when the store-clock-fast facility is installed and the TRACE TOD-clock control in bit 32 of control register 0 is one, it is unpredictable whether explicit tracing causes a serialization function to be performed.
- PURGE ALB, PURGE TLB, and SET PREFIX. PURGE ALB and SET PREFIX also cause the ALB to be cleared of all entries. PURGE TLB and SET PREFIX also cause the TLB to be cleared of all entries.
- SIGNAL PROCESSOR. The set-architecture SIGNAL PROCESSOR order causes serialization on all CPUs in the configuration.
- TEST BLOCK.
- TRANSACTION BEGIN and TRANSACTION END

The four trace functions — branch tracing, ASN tracing, mode tracing, and explicit tracing — cause a serialization function to be performed before the trace action and after completion of the trace action. However, it is unpredictable whether or not a store into a trace-table entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store. Additionally, when the store-clock-fast facility is installed and the TRACE TOD-clock control in bit 32 of control register 0 is one, it is unpredictable whether explicit tracing causes a serialization function to be performed.

In the following discussion, the term *serializing instruction* refers to an instruction which causes one or more serialization functions to be performed. The term *serializing operation* refers to a unit of operation within an instruction or to a machine operation such as an interruption which causes a serialization function is performed.

The sequence of events associated with a serializing operation is as follows:

1. All conceptually previous storage accesses by the CPU are completed as observed by other CPUs and by channel programs. This includes all conceptually previous stores and changes to the storage keys.
2. The normal function associated with the serializing operation is performed. In the case of instruction execution, operands are fetched, and the storing of results is completed. The exceptions are LOAD PSW, LOAD PSW EXTENDED, and SET PREFIX, in which the operand may be fetched before previous stores have been com-

pleted, and interruptions, in which the interruption code and associated fields may be stored prior to the serialization. The fetching of the serializing instruction occurs before the execution of the instruction and may precede the execution of previous instructions, but may not precede the completion of any previous serializing operation. In the case of an interruption, the old PSW, the interruption code, and other information, if any, are stored, and the new PSW is fetched, but not necessarily in that sequence.

3. Finally, instruction fetch and operand accesses for conceptually subsequent operations may begin.

A serializing function affects the sequence of storage accesses that are under the control of the CPU in which the serializing function takes place. It does not affect the sequence of storage accesses under the control of other CPUs and of channel programs.

Programming Notes:

1. The following are some effects of a serializing operation:

   a. When the execution of an instruction changes the contents of a storage location that is used as a source of a following instruction and when different addresses are used to designate the same absolute location for storing the result and fetching the instruction, a serializing operation following the change ensures that the modified instruction is executed.

   b. When a serializing operation takes place, other CPUs and channel programs observe instruction and operand fetching and result storing to take place in the sequence established by the serializing operation.

2. Storing into a location from which a serializing instruction is fetched does not necessarily affect the execution of the serializing instruction unless a serializing function has been performed after the storing and before the execution of the serializing instruction.

3. Following is an example showing the effects of serialization. Location A initially contains FF hex.

CPU 1		CPU 2		
MVI	A,X'00'	G	CLI	A,X'00'
BCR	15,0		BNE	G

   The BCR 15,0 instruction executed by CPU 1 is a serializing instruction that ensures that the store by CPU 1 at location A is completed. However, CPU 2 may loop indefinitely, or until the next interruption on CPU 2, because CPU 2 may already have fetched from location A for every execution of the CLI instruction. A serializing instruction must be in the CPU-2 loop to ensure that CPU 2 will again fetch from location A.

Specific-Operand Serialization

Certain instructions cause specific-operand serialization to be performed for an operand of the instruction. As observed by other CPUs and by the channel subsystem, a specific-operand-serialization operation consists in completing all conceptually previous storage access to the specified operand by the CPU before a conceptually subsequent accesses to the specific storage operand of the instruction may occur. At the completion of an instruction causing specific-operand serialization, the instruction's store is completed as observed by other CPUs and channel programs.

Specific-operand serialization is performed by the execution of the following instructions:

- ADD IMMEDIATE (ASI, AGSI) and ADD LOGICAL WITH SIGNED IMMEDIATE, for the first operand, when the interlocked-access facility 1 is installed and the first operand is aligned on a boundary which is integral to the size of the operand.
- AND (NI, NIY), when the interlocked-access facility 2 is installed
- EXCLUSIVE OR (XI, XIY), when the interlocked-access facility 2 is installed
- LOAD AND ADD, LOAD AND ADD LOGICAL, LOAD AND AND, LOAD AND EXCLUSIVE OR, LOAD AND OR, for the second operand.
- OR (OI, OIY), when the interlocked-access facility 2 is installed

Channel-Program Serialization

Serialization of a channel program occurs as follows:

1. All storage accesses and storage-key accesses by the channel program follow initiation of the execution of START SUBCHANNEL, or, if suspended, RESUME SUBCHANNEL, as observed by CPUs and by other channel programs. This includes all accesses for the CCWs, IDAWs, MIDAWs, and data.
2. All storage accesses and storage-key accesses by the channel program are completed, as observed by CPUs and by other channel programs, before the subchannel status indicating status-pending with primary status is made available to any CPU.
3. If a CCW contains a PCI flag or a suspend flag which is one, all storage accesses and storage-key accesses due to CCWs preceding it in the CCW chain are completed, as observed by CPUs and by other channel programs, before the subchannel status indicating status-pending with intermediate status (PCI or suspended) is made available to any CPU.

The serialization of a channel program does not affect the sequence of storage accesses or storage-key accesses performed by other channel programs or by a CPU.

Quiescing

Quiescing is an additional means of serialization that may be performed to ensure that updates to a configuration-wide resource are immediately visible to all CPUs. A quiescing operation initiated by one CPU causes the following to occur on all other CPUs in the configuration:

- Any instruction or unit of operation executing on the CPU is completed.
- Any transaction executing on the CPU is aborted with abort code 255; the condition code is set to 2.
- All locally-cached copies of the configuration-wide resource being updated are discarded.
- Resumption of instruction execution is delayed until the update to the configuration-wide resource is visible to all CPUs.

Quiescing may occur as a result of the execution of the following key-setting instructions:

- PERFORM FRAME MANAGEMENT FUNCTION (when the set-key control is one)
- SET STORAGE KEY EXTENDED.

When the nonquiescing key-setting facility is not installed, a quiescing operation is performed by either of the above instructions. Similarly, when the nonquiescing key-setting facility is installed, but the nonquiescing (NQ) control of the SSKE instruction is zero, a quiescing operation is performed. When the nonquiescing key-setting facility is installed and the NQ control of the SSKE instruction is one, the quiescing operation is not performed. When the nonquiescing key-setting facility is installed, it is unpredictable whether execution of PFMF with the set-key control set to 1 results in a quiescing operation.

Programming Note: Although quiescing is defined as a additional serialization mechanism, the term *serialization function* (used in the description of CPU instructions and channel operations) is limited to CPU serialization and channel-program serialization.

Chapter 6. Interruptions

Interruption Action 6-2
Interruption Code 6-5
Enabling and Disabling 6-6
Handling of Floating Interruption Conditions . . 6-6
Instruction-Length Code 6-7
Zero ILC 6-7
ILC on Instruction-Fetching Exceptions 6-8
Exceptions Associated with the PSW 6-9
Early Exception Recognition 6-9
Late Exception Recognition 6-10
External Interruption 6-10
Clock Comparator 6-12
CPU Timer 6-12
Emergency Signal 6-12
External Call 6-13
Interrupt Key 6-13
Malfunction Alert 6-13
Measurement Alert 6-13
Service Signal 6-14
Timing Alert 6-14
ETR-Timing-Alert Condition 6-14
STP-Timing-Alert Condition 6-14
Warning Track 6-15
I/O Interruption 6-15
Machine-Check Interruption 6-16
Program Interruption 6-16
Data-Exception Code (DXC) 6-17
Priority of Program Interruptions for Data Exceptions 6-17
Vector-Exception Code 6-20
Program-Interruption Conditions 6-20
Addressing Exception 6-20
AFX-Translation Exception 6-22
ALEN-Translation Exception 6-22
ALE-Sequence Exception 6-22
ALET-Specification Exception 6-22
ASCE-Type Exception 6-22
ASTE-Instance Exception 6-23
ASTE-Sequence Exception 6-23
ASTE-Validity Exception 6-24
ASX-Translation Exception 6-24
Crypto-Operation Exception 6-24
Data Exception 6-25
Decimal-Divide Exception 6-26
Decimal-Overflow Exception 6-26
Execute Exception 6-26
EX-Translation Exception 6-26
Extended-Authority Exception 6-27
Fixed-Point-Divide Exception 6-27
Fixed-Point-Overflow Exception 6-27
HFP-Divide Exception 6-28
HFP-Exponent-Overflow Exception 6-28
HFP-Exponent-Underflow Exception 6-28
HFP-Significance Exception 6-28
HFP-Square-Root Exception 6-29
LFX-Translation Exception 6-29
LSTE-Sequence Exception 6-29
LSX-Translation Exception 6-29
LX-Translation Exception 6-30
Monitor Event 6-30
Operand Exception 6-31
Operation Exception 6-31
Page-Translation Exception 6-32
PC-Translation-Specification Exception . . . 6-32
PER Event 6-32
Primary-Authority Exception 6-33
Privileged-Operation Exception 6-33
Protection Exception 6-34
Region-First-Translation Exception 6-35
Region-Second-Translation Exception 6-36
Region-Third-Translation Exception 6-36
Secondary-Authority Exception 6-37
Segment-Translation Exception 6-37
Space-Switch Event 6-37
Special-Operation Exception 6-38
Specification Exception 6-40
Stack-Empty Exception 6-44
Stack-Full Exception 6-44
Stack-Operation Exception 6-44
Stack-Specification Exception 6-45
Stack-Type Exception 6-45
Trace-Table Exception 6-45
Transaction-Constraint Exception 6-45
Transactional-Execution-Aborted Event . . . 6-46
Translation-Specification Exception 6-46
Vector Processing Exception 6-47
Collective Program-Interruption Names 6-47
Recognition of Access Exceptions 6-47
Multiple Program-Interruption Conditions 6-50
Access Exceptions 6-53
ASN-Translation Exceptions 6-56
Subspace-Replacement Exceptions 6-56
Trace Exceptions 6-56
Restart Interruption 6-56
Supervisor-Call Interruption 6-57
Priority of Interruptions 6-57

© Copyright IBM Corp. 2000, 2017

The interruption mechanism permits the CPU to change its state as a result of conditions external to the configuration, within the configuration, or within the CPU itself. To permit fast response to conditions of high priority and immediate recognition of the type of condition, interruption conditions are grouped into six classes: external, input/output, machine check, program, restart, and supervisor call.

Interruption Action

An interruption consists in storing the current PSW as an old PSW, storing information identifying the cause of the interruption, and fetching a new PSW. Processing resumes as specified by the new PSW.

The old PSW stored on an interruption normally contains the address of the instruction that would have been executed next had the interruption not occurred, thus permitting resumption of the interrupted program. For program and supervisor-call interruptions, the information stored also contains a code that identifies the length of the last-executed instruction, thus permitting the program to respond to the cause of the interruption. In the case of some program conditions for which the normal response is reexecution of the instruction causing the interruption, the instruction address directly identifies the instruction last executed.

Except for restart, an interruption can occur only when the CPU is in the operating state. The restart interruption can occur with the CPU in either the stopped or operating state.

The details of source identification, location determination, and instruction execution are explained in later sections and are summarized in Figure 6-1 on page 6-2.

Source Identification	Interruption Code		PSW-Mask Bits	Mask Bits in Control Registers Reg	Mask Bits in Control Registers Bit	ILC Set	Execution of Instruction Identified by Old PSW
MACHINE CHECK	Locations 232-239[1]						
Old PSW: 352-367 (z/Arch), 48-55 (390-CM)							
New PSW: 480-495 (z/Arch), 112-119 (390-CM)							
Exigent condition			13			u	terminated or nullified[2]
Repressible condition			13	14	35-39	u	unaffected[2]
SUPERVISOR CALL	Locations 138-139						
Old PSW: 320-335 (z/Arch), 32-39 (390-CM)							
New PSW: 448-463 (z/Arch), 96-103 (390-CM)							
Instruction bits	00000000 ssssssss					1, 2, 3	completed
PROGRAM	Locations 142-143						
Old PSW: 336-351 (z/Arch), 40-47 (390-CM)							
New PSW: 464-479 (z/Arch), 104-111 (390-CM)	Binary	Hex[3]					
Operation	000000t0 p0000001	0001				1, 2, 3	suppressed
Privileged operation	000000t0 p0000010	0002				1, 2, 3	suppressed
Execute	000000t0 p0000011	0003				2, 3	suppressed
Protection	000000t0 p0000100	0004				1, 2, 3	suppressed or terminated
Addressing	000000t0 p0000101	0005				1, 2, 3	suppressed or terminated
Specification	000000t0 p0000110	0006				0, 1, 2, 3	suppressed or completed
Data	000000t0 p0000111	0007				1, 2, 3	suppressed, terminated or completed
Fixed-point overflow	000000t0 p0001000	0008	20			1, 2, 3	completed
Fixed-point divide	000000t0 p0001001	0009				1, 2, 3	suppressed or completed
Decimal overflow	000000t0 p0001010	000A	21			2, 3	completed
Decimal divide	000000t0 p0001011	000B				2, 3	suppressed

Figure 6-1. Interruption Action (Part 1 of 4)

Source Identification	Interruption Code			PSW-Mask Bits	Mask Bits in Control Registers		ILC Set	Execution of Instruction Identified by Old PSW
					Reg	Bit		
HFP exp. overflow	000000t0	p0001100	000C				1, 2, 3	completed
HFP exp. underflow	000000t0	p0001101	000D	22			1, 2, 3	completed
HFP significance	000000t0	p0001110	000E	23			1, 2, 3	completed
HFP divide	000000t0	p0001111	000F				1, 2, 3	suppressed
Segment translation	000000t0	p0010000	0010				1, 2, 3	nullified
Page translation	000000t0	p0010001	0011				1, 2, 3	nullified
Translation spec	000000t0	p0010010	0012				1, 2, 3	suppressed
Special operation	000000t0	p0010011	0013		0	33	1, 2, 3	suppressed
Operand	00000000	p0010101	0015				2, 3	suppressed
Trace table	00000000	p0010110	0016				1, 2, 3	nullified
Transaction constraint	000000t0	00011000	0018				1, 2, 3	suppressed
Vector-processing	000000t0	p0011011	001B				2 3	suppressed
Space-switch event	00000000	p0011100	001C		1	57	0, 1, 2, 3	completed
HFP square root	000000t0	p0011101	001D				2, 3	suppressed
PC-transl spec	00000000	p0011111	001F				2, 3	suppressed
AFX translation	00000000	p0100000	0020				1, 2, 3	nullified
ASX translation	00000000	p0100001	0021				1, 2, 3	nullified
LX translation	00000000	p0100010	0022				2, 3	nullified
EX translation	00000000	p0100011	0023				2, 3	nullified
Primary authority	00000000	p0100100	0024				2, 3	nullified
Secondary authority	00000000	p0100101	0025				1, 2, 3	nullified
LFX translation	00000000	p0100110	0026				2, 3	nullified
LSX translation	00000000	p0100111	0027				2, 3	nullified
ALET specification	000000t0	p0101000	0028				1, 2, 3	suppressed
ALEN translation	000000t0	p0101001	0029				1, 2, 3	nullified
ALE sequence	000000t0	p0101010	002A				1, 2, 3	nullified
ASTE validity	000000t0	p0101011	002B				1, 2, 3	nullified
ASTE sequence	000000t0	p0101100	002C				1, 2, 3	nullified
Extended authority	000000t0	p0101101	002D				1, 2, 3	nullified
LSTE sequence	00000000	p0101110	002E				2, 3	nullified
ASTE instance	00000000	p0101111	002F				1, 2, 3	nullified
Stack full	00000000	p0110000	0030				2, 3	nullified
Stack empty	00000000	p0110001	0031				1, 2, 3	nullified
Stack specification	00000000	p0110010	0032				1, 2, 3	nullified
Stack type	00000000	p0110011	0033				1, 2, 3	nullified
Stack operation	00000000	p0110100	0034				1, 2, 3	nullified
ASCE type	000000t0	p0111000	0038				1, 2, 3	nullified
Region first trans	000000t0	p0111001	0039				1, 2, 3	nullified
Region second trans	000000t0	p0111010	003A				1, 2, 3	nullified
Region third trans	000000t0	p0111011	003B				1, 2, 3	nullified
Monitor event	000000t0	p1000000	0040		8	32-47	2, 3	completed

Figure 6-1. Interruption Action (Part 2 of 4)

Source Identification	Interruption Code		PSW-Mask Bits	Mask Bits in Control Registers		ILC Set	Execution of Instruction Identified by Old PSW
				Reg	Bit		
PER basic event	000000tn 1nnnnnnn[5]	0080	1	9	32-36	0, 1, 2, 3	completed[6]
PER nullification event	000000t0 10000000	0080	1	9	33,39	0	nullified[7]
Crypto operation	000000t1 p0011001	0119				2, 3	nullified
Transactional-execution aborted event	0000001n nnnnnnnn	0200		0	8-9	1, 2, 3	completed[8]
EXTERNAL Old PSW: 304-319 (z/Arch), 24-31 (390-CM) New PSW: 432-447 (z/Arch), 88-95 (390-CM)	Location 134-135 Binary	Hex[3]					
Interrupt key	00000000 01000000	0040	7	0	57	u	unaffected
Clock comparator	00010000 00000100	1004	7	0	52	u	unaffected
CPU timer	00010000 00000101	1005	7	0	53	u	unaffected
Warning track	00010000 00000111	1007	7	0	30	u	unaffected
Malfunction alert	00010010 00000000	1200	7	0	48	u	unaffected
Emergency Signal	00010010 00000001	1201	7	0	49	u	unaffected
External call	00010010 00000010	1202	7	0	50	u	unaffected
Timing alert	00010100 00000110	1406	7	0	59	u	unaffected
Measurement alert	00010100 00000111	1407	7	0	58	u	unaffected
Service signal	00100100 00000001	2401	7	0	54	u	unaffected
INPUT/OUTPUT Old PSW: 368-383 (z/Arch), 56-63 (390-CM) New PSW: 496-511 (z/Arch), 120-127 (390-CM)	Locations 184-191						
I/O-interruption subclass			6	6	32-39[4]	u	unaffected
RESTART Old PSW: 288-303 (z/Arch), 8-15 (390-CM) New PSW: 416-431 (z/Arch), 0-7 (390-CM)	None						
Restart key						u	unaffected

Explanation:

Locations for the old PSWs, new PSWs, and interruption codes are real locations.

z/Arch Interruption PSW locations when the CPU is in the z/Architecture architectural mode.

390-CM Interruption PSW locations when the CPU is in the ESA/390-compatibility mode.

1 A model-independent machine-check interruption code of 64 bits is stored at real locations 232-239.

2 The effect of the machine-check condition is indicated by bits in the machine- check-interruption code. The setting of these bits indicates the extent of the damage and whether the unit of operation is nullified, terminated, or unaffected.

3 The interruption code in the column labeled “Hex” is the hex code for the basic interruption; this code does not show the effects of concurrent interruption conditions represented by n or p in the column labeled “Binary.”

4 Bits 32-39 of control register 6 provide detailed masking of I/O-interruption subclasses 0-7 respectively.

5 When the interruption code indicates a PER basic event, an ILC of 0 may be stored only when bits 8-15 of the interruption code are 10000110 (PER, specification).

6 The unit of operation is completed, unless a program exception concurrently indicated causes the unit of operation to be nullified, suppressed, or terminated.

7 The unit of operation is nullified and no other program interruption is indicated when a nullifying PER event is recognized.

Figure 6-1. Interruption Action (Part 3 of 4)

Source Identification	Interruption Code	PSW-Mask Bits	Mask Bits in Control Registers		ILC Set	Execution of Instruction Identified by Old PSW
			Reg	Bit		
[8] A transactional-execution aborted event indication always accompanies another program-interruption indication. The completed indication refers only to the TBEGIN or TBEGINC instruction, as identified by the old PSW. The instruction identified by the aborted-transaction instruction address in the TDB is either nullified, suppressed, or completed.						
n A possible nonzero code indicating another concurrent program-interruption condition						
p If one, the bit indicates a concurrent PER-event interruption condition.						
s Bits of the I field of SUPERVISOR CALL.						
t Transactional-execution-aborted event						
u Not stored.						

Figure 6-1. Interruption Action (Part 4 of 4)

Programming Note: In the ESA/390-compatibility mode, an old PSW stored during interruption processing has the short format, as illustrated in Figure 4-3 on page 4-8. If the model does not recognize a specification exception when PSW bit 31 is one in the ESA/390-compatibility mode (thus allowing the CPU to operate in the 64-bit addressing mode), any nonzero bits in bit positions 0-32 of the PSW instruction address will be lost when storing the interruption-old PSW.

Similarly in the ESA/390-compatibility mode, if the model allows the execution of either (a) LOAD PSW EXTENDED or (b) RESUME PROGRAM with the P bit, bit 13 of the parameter list, set to one, then a 16-byte PSW may be loaded. If such a PSW has a format error in which bit 31 is zero, but any of bits 64-96 (bits 0-32 of the instruction address) are nonzero, then an early specification exception is recognized. However, the program-old PSW that is stored when such a PSW becomes active has the short PSW format in which the invalid instruction-address bits are not stored.

Interruption Code

The six classes of interruptions (external, I/O, machine check, program, restart, and supervisor call) are distinguished by the storage locations at which the old PSW is stored and from which the new PSW is fetched. For most classes, the causes are further identified by an interruption code and, for some classes, by additional information placed in permanently assigned real storage locations during the interruption. (See “Assigned Storage Locations” on page 3-73.) For external, program, and supervisor-call interruptions, the interruption code consists of 16 bits.

For external interruptions, the interruption code is stored at real locations 134-135. A parameter may be stored at real locations 128-131, or a CPU address may be stored at real locations 132-133.

For I/O interruptions, the I/O-interruption code is stored at real locations 184-195. The I/O-interruption code consists of a 32-bit subsystem-identification word, a 32-bit interruption parameter, and a 32-bit I/O-interruption identification word.

For machine-check interruptions, the interruption code consists of 64 bits and is stored at real locations 232-239. Additional information for identifying the cause of the interruption and for recovering the state of the machine may be provided by the contents of the machine-check failing-storage address and the contents of the fixed-logout and machine-check-save areas. (See Chapter 11, “Machine-Check Handling.”)

For supervisor-call interruptions, the interruption code comprises eight binary zeros concatenated with the 8-bit immediate operand of the SUPERVISOR CALL instruction (as modified by an execute-type instruction, if applicable). The interruption code is stored at real locations 138-139, and the instruction-length code is stored in bit positions 5 and 6 of real location 137.

For program interruptions, the interruption code is stored at real locations 142-143, and the instruction-length code is stored in bit positions 5 and 6 of real location 141. In the z/Architecture architectural mode, further information may be provided in the

form of the data-exception code (DXC) or vector-exception code (VXC), monitor-class number, PER code, addressing-and-translation-mode identification, PER address, exception access identification, PER access identification, operand-access identification, translation-exception identification, and monitor code, which are stored at real locations 144-162 and 168-183. In the ESA/390-compatibility mode, the DXC, translation-exception identification, monitor-class number, PER code, addressing-and-translation-mode identification, PER address, monitor code, exception access identification, and PER access identification, may be stored at real locations 144-161.

Enabling and Disabling

By means of mask bits in the current PSW, floating-point-control (FPC) register, and control registers, the CPU may be enabled or disabled for all external, I/O, and machine-check interruptions and for some program interruptions. When a mask bit is one, the CPU is enabled for the corresponding class of interruptions, and those interruptions can occur.

When a mask bit is zero, the CPU is disabled for the corresponding interruptions. The conditions that cause I/O interruptions remain pending. External-interruption conditions either remain pending or persist until the cause is removed. Machine-check-interruption conditions, depending on the type, are ignored, remain pending, or cause the CPU to enter the check-stop state. The disallowed program-interruption conditions are ignored, except that some causes are indicated also by the setting of the condition code, and IEEE exceptions set flags in the FPC register. The setting of the HFP-significance and HFP-exponent-underflow program-mask bits affects the manner in which HFP operations are completed when the corresponding condition occurs. Similarly, the setting of the IEEE mask bits in the FPC register affects the manner in which IEEE computational operations are completed when the corresponding condition occurs.

Programming Notes:

1. Mask bits in the PSW provide a means of disallowing most maskable interruptions; thus, subsequent interruptions can be disallowed by the new PSW introduced by an interruption. Furthermore, the mask bits can be used to establish a hierarchy of interruption priorities, where a condition in one class can interrupt the program handling a condition in another class but not vice versa. To prevent an interruption-handling routine from being interrupted before the necessary housekeeping steps are performed, the new PSW must disable the CPU for further interruptions within the same class or within a class of lower priority.

2. Because the mask bits in control registers are not changed as part of the interruption procedure, these masks cannot be used to prevent an interruption immediately after a previous interruption in the same class. The mask bits in control registers provide a means for selectively enabling the CPU for some sources and disabling it for others within the same class.

3. Controlling bits exist for several program interruptions, but with no mask bit in the PSW. Such bits include the IEEE mask bits in the FPC register, the monitor masks in bit positions 48-63 of control register 8, and the primary space-switch-event-control bits, bit 57 of control registers 1 and 13, respectively. A bit of this nature is somewhat arbitrarily considered to be a “mask” bit only if the polarity is such that interruption is enabled when the bit is one.

   Thus, for example, the SSM-suppression-control bit, bit 33 of control register 0, is considered to be a mask bit, while the AFP-register-control bit, bit 45 of control register 0, is not. Regardless of the polarity of such control bits, to avoid another program interruption, an interruption-handling routine must avoid issuing instructions subject to these bits until they have been set appropriately.

   In the z/Architecture architectural mode, the enhanced-monitor masks in bit positions 16-31 of control register 8 are an exception to the above definition of a mask in that they work in conjunction with the monitor masks in bit positions 48-63 of control register 8. When both a monitor mask bit and its corresponding enhanced-monitor mask bit are one, a monitor-event counting operation occurs rather than an interruption.

Handling of Floating Interruption Conditions

An interruption condition which can be presented to any CPU in the configuration is called a floating interruption condition. The condition is presented to the

first CPU in the configuration which is enabled for the corresponding interruption and which can perform the interruption, and then the condition is cleared and not presented to any other CPU in the configuration. A CPU cannot perform the interruption when it is in the check-stop state, has an invalid prefix, is in a string of program interruptions due to a specification exception of the type which is recognized early or is in the stopped state. However, a CPU with the rate control set to instruction step can perform the interruption when the start key is activated.

Service signal, I/O, and certain machine-check conditions are floating interruption conditions. Additionally, when the STP-floating-interrupt facility is installed and STP timing-alert floating interruptions are enabled, STP timing-alert external-interruption conditions are floating interruption conditions.

Instruction-Length Code

The instruction-length code (ILC) occupies two bit positions and provides the length of the last instruction executed. It permits identifying the instruction causing the interruption when the instruction address in the old PSW designates the next sequential instruction. The ILC is provided also by the BRANCH AND LINK instructions in the 24-bit addressing mode.

The ILC for program and supervisor-call interruptions is stored in bit positions 5 and 6 of the bytes at real locations 141 and 137, respectively. For external, I/O, machine-check, and restart interruptions, the ILC is not stored since it cannot be related to the length of the last-executed instruction.

For supervisor-call and program interruptions, a nonzero ILC identifies in halfwords the length of the instruction that was last executed. That instruction may be one for which a specification exception was recognized due to an odd instruction address or for which an access exception (addressing, ASCE-type, page-translation, protection, region-translation, segment-translation, or translation-specification) was recognized during the fetching of the instruction. Whenever an instruction is executed by means of EXECUTE, instruction-length code 2 is set to indicate the length of EXECUTE and not that of the target instruction. Similarly, when an instruction is executed by means of EXECUTE RELATIVE LONG, instruction length code 3 is set.

The value of a nonzero instruction-length code is related to the leftmost two bits of the instruction. The value does not depend on whether the operation code is assigned or on whether the instruction is installed. The following table summarizes the meaning of the instruction-length code:

ILC		Instruction Bits 0-1	Instruction Length
Decimal	Binary		
0	00		Not available
1	01	00	One halfword
2	10	01	Two halfwords
2	10	10	Two halfwords
3	11	11	Three halfwords

Except as noted below, when a transaction is aborted due to a program interruption, the ILC stored in bit positions 5-6 of real location 141 is respective to the instruction identified by the aborted-transaction instruction address (ATIA) in the program-interruption TDB. Similarly, when a nonconstrained transaction is aborted due to a program interruption, and the transaction-diagnostic-block address is valid, the ILC stored in bit positions 5-6 of byte 37 in the TBEGIN-specified-TDB is respective to the instruction identified by the ATIA in the TDB. However, when a transaction is aborted due to a program interruption and a null TDB is stored, the ILC is unpredictable.

Programming Note: When a transaction is aborted due to a program interruption, the ILC is not meaningful with respect to the instruction address in the program-old PSW. If a constrained transaction is aborted due to a program interruption, the instruction address in the program-old PSW points directly to the TBEGINC instruction that initiated constrained transactional execution. If a nonconstrained transaction is aborted due to an unfiltered program interruption, the instruction address in the program-old PSW points six bytes past the outermost TBEGIN instruction that initiated nonconstrained transactional execution. The program can determine whether a constrained or nonconstrained transaction was aborted by examining the constrained-transaction indication in the program-interruption TDB (bit 1 of the flags byte, stored at real location 6,145).

Zero ILC

Instruction-length code 0, after a program interruption (other than for an instruction-fetching nullification event), indicates that the instruction address stored in the old PSW does not identify the instruction caus-

ing the interruption. In the case of a program interruption due to an instruction-fetching nullification event, the ILC is set to zero. The remainder of this section discusses zero ILC due to cases other than for instruction-fetching nullification events.

An ILC of 0 occurs when a specification exception due to a PSW-format error is recognized as part of early exception recognition and the PSW has been introduced by LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, or an interruption. (See "Exceptions Associated with the PSW" on page 6-9.) In the case of LOAD PSW, LOAD PSW EXTENDED, or PROGRAM RETURN, the instruction address of the instruction or of an execute-type instruction (EXECUTE or EXECUTE RELATIVE LONG) has been replaced by the instruction address in the new PSW. When the invalid PSW is introduced by an interruption, the PSW-format error cannot be attributed to an instruction.

In the case of LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, and the supervisor-call interruption, a PER event may be indicated concurrently with a specification exception for which the ILC is 0.

In the case of a PROGRAM RETURN instruction that causes both a space-switch event and a PSW-format error, the space-switch event is recognized, but it is unpredictable whether the ILC is 0 or 1, or 0 or 2 if EXECUTE was used, or 0 or 3 if EXECUTE RELATIVE LONG was used.

ILC on Instruction-Fetching Exceptions

When a program interruption occurs because of an exception that prohibits access to the instruction, the instruction is considered to have been executed without being fetched, and the instruction-length code cannot be set on the basis of the first two bits of the instruction. As far as the significance of the ILC for this case is concerned, the following two situations are distinguished:

1. When an odd instruction address causes a specification exception to be recognized or when an addressing, protection, or translation-specification exception is encountered on fetching an instruction, the ILC is set to 1, 2, or 3, indicating the multiple of 2 by which the instruction address has been incremented. It is unpredictable whether the instruction address is incremented by 2, 4, or 6. By reducing the instruction address in the old PSW by the number of halfword locations indicated in the ILC, the instruction address originally appearing in the PSW may be obtained.
2. When an ASCE-type, region-translation, segment-translation, or page-translation exception is recognized while fetching an instruction, the ILC is arbitrarily set to 1, 2, or 3. In this case, the operation is nullified, and the instruction address is not incremented.

The ILC is not necessarily related to the first two bits of the instruction when the first halfword of an instruction can be fetched but an access exception is recognized on fetching the second or third halfword. The ILC may be arbitrarily set to 1, 2, or 3 in these cases. The instruction address is or is not updated, as described in situations 1 and 2 above.

When any exceptions or any PER events other than an instruction-fetching-nullification event are encountered on fetching the target instruction of EXECUTE, the ILC is 2. When any exceptions or any PER events other than an instruction-fetching-nullification event are encountered on fetching the target instruction of EXECUTE RELATIVE LONG, the ILC is 3. When a PER instruction-fetching-nullification event is encountered on fetching the target of an execute-type instruction, and no other exception is recognized, the ILC is 0.

Programming Notes:

1. A nonzero instruction-length code for a program interruption indicates the number of halfword locations by which the instruction address in the program old PSW must be reduced to obtain the instruction address of the last instruction executed, unless one of the following situations exists:
   a. The interruption is caused by an exception resulting in nullification.
   b. An interruption for a PER event occurs before the execution of an interruptible instruction is completed, and no other program-interruption condition is indicated concurrently.
   c. The interruption is caused by a PER event or space-switch event due to LOAD PSW, LOAD PSW EXTENDED, or a branch or linkage instruction, including SUPERVISOR CALL (but not including MONITOR CALL).

d. The interruption is caused by an addressing exception or protection exception for the storage operand of a LOAD CONTROL instruction that loads the control register (1 or 13) containing the address-space-control element that specifies the address space from which instructions are fetched.

For situations a and b above, the instruction address in the PSW is not incremented, and the instruction designated by the instruction address is the same as the last one executed. These situations are the only ones in which the instruction address in the old PSW identifies the instruction causing the exception. Situation b can be distinguished from a PER event indicated after completion of an interruptible or noninterruptible instruction in that, for situation b, the instruction address in the PSW is the same as the PER address at real location 152.

For situation c, the instruction address has been replaced as part of the operation, and the address of the last instruction executed cannot be calculated using the one appearing in the program old PSW.

For situation d, the effective address of the last instruction executed can be calculated, but, since the address-space-control element for the instruction address space is unpredictable, the corresponding real address is unknown.

2. The instruction-length code (ILC) is redundant when a PER event is indicated since the PER address in the doubleword at real location 152 identifies the instruction causing the interruption (or the execute-type instruction, as appropriate). Similarly, the ILC is may be unpredictable when the operation is nullified; in this case the instruction address in the PSW is not incremented. If the ILC value is required in this case, it can be derived from the operation code of the instruction identified by the old PSW.

3. The address of the last instruction executed before a program interruption is insufficient to locate the program problem if one of the following situations exists:

   a. The interruption is caused by an access exception encountered in fetching an instruction, and the instruction address was introduced into the PSW by a means other than sequential operation (by a branch or linkage instruction, LOAD PSW, LOAD PSW EXTENDED, an interruption, or conclusion of an IPL sequence).

   b. The interruption is caused by a specification exception due to an odd instruction address, which necessarily also results from introduction of an instruction address into the PSW.

   c. The interruption is caused by an early specification exception due to a STORE THEN OR SYSTEM MASK or SET SYSTEM MASK instruction that switches to or from the real mode while introducing invalid values in bit positions 0-7 of the PSW.

   For situations a and b, the instruction address was replaced by the operation preceding the last instruction execution, and the address of the program location related to that preceding operation is unavailable.

   For situation c, the address of the last instruction executed is available, but the corresponding real address is unknown.

4. The address of the last instruction executed is not available when an interruption is caused by an early specification exception due to a LOAD PSW, LOAD PSW EXTENDED, or PROGRAM RETURN instruction or an interruption.

Exceptions Associated with the PSW

Exceptions associated with erroneous information in the current PSW may be recognized as follows:

- When the information is introduced into the PSW (called early exception recognition)
- As part of the execution of the next instruction (called late exception recognition)

Errors in the PSW which are specification-exception conditions are called PSW-format errors.

Early Exception Recognition

For the following error conditions, a program interruption for a specification exception occurs immediately after the PSW becomes active:

- Any of the unassigned bits (0, 2-4, 25-30, or 33-63) is a one.
- Bit 12 is a one.

- Bit 24 is one (recognition of this condition is optional)
- Bits 31 and 32 are zero and one, respectively, and bits 64-96 are not all zeros.
- Bits 31 and 32 are both zero, and bits 64-103 are not all zeros.
- Bits 31 and 32 are one and zero, respectively.

The interruption occurs regardless of whether the wait state is specified. If the invalid PSW causes the CPU to become enabled for a pending I/O, external, or machine-check interruption, the program interruption occurs instead, and the pending interruption is subject to the mask bits of the new PSW introduced by the program interruption.

When an interruption or the execution of LOAD PSW, LOAD PSW EXTENDED, or PROGRAM RETURN introduces a PSW with one of the above error conditions, the instruction-length code is set to 0, and the newly introduced PSW is stored unmodified as the old PSW. Except as noted below, when one of the above error conditions is introduced by execution of SET SYSTEM MASK or STORE THEN OR SYSTEM MASK, the instruction-length code is set to 2, and the instruction address is incremented by 4. When one of the above error conditions is introduced by execution of SET SYSTEM MASK or STORE THEN OR SYSTEM MASK that is the target of EXECUTE RELATIVE LONG, the instruction-length code is set to 3, and the instruction address is incremented by 6. The PSW containing the invalid value introduced into the system-mask field is stored as the old PSW.

Late Exception Recognition

For the following conditions, the exception is recognized as part of the execution of the next instruction:

- A specification exception is recognized due to an odd instruction address in the PSW (PSW bit 127 is one).
- An access exception (addressing, ASCE-type, page-translation, protection, region-translation, segment-translation, or translation-specification) is associated with the location designated by the instruction address or with the location of the second or third halfword of the instruction starting at the designated instruction address.

The instruction-length code and instruction address stored in the program old PSW under these conditions are discussed in "ILC on Instruction-Fetching Exceptions" on page 6-8, and an example is given in Figure 4-11 on page 4-42.

If an I/O, external, or machine-check-interruption condition is pending and the PSW causes the CPU to be enabled for that condition, the corresponding interruption occurs, and the PSW is not inspected for exceptions which are recognized late. Similarly, a PSW specifying the wait state is not inspected for exceptions which are recognized late.

Programming Notes:

1. The execution of BRANCH AND SET AUTHORITY, LOAD ADDRESS SPACE PARAMETERS, LOAD PSW, LOAD PSW EXTENDED, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, RESUME PROGRAM, SET PREFIX, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, SET SYSTEM MASK, STORE THEN AND SYSTEM MASK, and STORE THEN OR SYSTEM MASK is suppressed on an addressing or protection exception, and hence the program old PSW provides information concerning the program causing the exception.
2. When the first halfword of an instruction can be fetched but an access exception is recognized on fetching the second or third halfword, the ILC is not necessarily related to the operation code.
3. If the new PSW introduced by an interruption contains a PSW-format error, a string of interruptions may occur. (See "Priority of Interruptions" on page 6-57.)

External Interruption

The external interruption provides a means by which the CPU responds to various signals originating from either inside or outside the configuration.

In the z/Architecture architectural mode, an external interruption causes the old PSW to be stored at real locations 304-319 and a new PSW to be fetched from real locations 432-447. In the ESA/390-compatibility mode, an external interruption causes the short-format old PSW to be stored at real locations 24-31 and

a short-format new PSW to be fetched from real locations 88-95.

The source of the interruption is identified in the interruption code which is stored at real locations 134-135. The instruction-length code is not stored.

Additionally, for the malfunction-alert, emergency-signal, and external-call conditions, a 16-bit CPU address is associated with the source of the interruption and is stored at real locations 132-133. When the CPU address is stored, bit 6 of the interruption code is set to one. For all other conditions, no CPU address is stored, bit 6 of the interruption code is set to zero, and zeros are stored at real locations 132-133.

For the timing-alert, measurement-alert, and service-signal interruptions, a 32-bit parameter is associated with the interruption and is stored at real locations 128-131. Bit 5 of the external-interruption code indicates that a parameter has been stored. When bit 5 is zero, the contents of real locations 128-131 remain unchanged.

External-interruption conditions are presented as one of three types of interruption requests: pendable, direct-condition, and floating as described below:

- A pendable interruption request is preserved and remains pending in each CPU until it is honored at the CPU. The pendable interruption condition is cleared when the interruption occurs.
- A direct-condition interruption request is pending only during the time that the condition exists. If the condition is removed before the request is honored, then no interruption occurs. If the condition persists, then more than one interruption may result from a single occurrence of the condition.
- A floating interruption request is held pending external to the CPU. The interruption can be accepted by any CPU in the configuration. The information associated with the interruption is not transferred to a particular CPU until the interruption occurs. The floating interruption condition is cleared when the interruption is accepted at any CPU in the configuration.

The clock comparator and CPU timer external-interruption conditions result in direct-condition interruption requests. The interrupt-key, malfunction-alert, emergency-signal, and external-call external-interruption conditions result in pendable interruption requests. The service-signal external-interruption condition results in a floating interruption request. The timing-alert external-interruption condition results in either a floating interruption request or a pendable interruption request. See the individual description of each of the external-interruption conditions for additional details.

When several interruption requests for a single source are generated before the interruption occurs, and the interruption condition is a pendable or floating interruption request, only one request for that source is preserved and remains pending.

An external interruption for a particular source can occur only when the CPU is enabled for interruption by that source. The external interruption occurs at the completion of a unit of operation. The external mask, PSW bit 7, and external subclass-mask bits in control register 0 control whether the CPU is enabled for a particular source. Each source for an external interruption has a subclass-mask bit assigned to it, and the source can cause an interruption only when the external-mask bit is one and the corresponding subclass-mask bit is one.

When the CPU becomes enabled for a pendable or floating external-interruption condition, the interruption occurs at the completion of the instruction execution or interruption that causes the enabling.

More than one source may present a request for an external interruption at the same time. When the CPU becomes enabled for more than one concurrently pending request, the interruption occurs for the pending condition or conditions having the highest priority.

The priorities for external-interruption requests in descending order are as follows:

- Interrupt key
- Malfunction alert
- Emergency signal
- External call
- Clock comparator
- CPU timer
- Timing alert
- Warning track
- Service signal or measurement alert

All requests are honored one at a time. When more than one emergency-signal request exists at a time or when more than one malfunction-alert request exists at a time, the request associated with the smallest CPU address is honored first.

Clock Comparator

When the TOD-clock-steering facility is not installed, an interruption request for the clock comparator exists whenever either of the following conditions is met:

1. The TOD clock is in the set or not-set state, and the value of the clock comparator is less than the value in the compared portion of the TOD clock, both compare values being considered unsigned binary integers.
2. The TOD clock is in the error or not-operational state.

When the TOD-clock-steering facility is installed, an interruption request for the clock comparator exists whenever the physical clock is in the set state and the value of the clock comparator is less than the value in the compared portion of the logical TOD clock, both compare values being considered unsigned binary integers.

When the TOD-clock-steering facility is installed and a CPU remains in the wait state for a long period of time, clock-comparator interruptions may be proportionally late by the maximum steering rate.

If the condition responsible for the request is removed before the request is honored, the request does not remain pending, and no interruption occurs. Conversely, the request is not cleared by the interruption, and, if the condition persists, more than one interruption may result from a single occurrence of the condition.

When the TOD-clock-steering facility is not installed, and the TOD clock is set or changes state, interruption conditions, if any, that are due to the clock comparator may or may not be recognized for up to 1.048576 seconds after the change.

When the TOD-clock-steering facility is installed and the physical clock is set or changes state, or the logical TOD clock is changed by PTFF-ATO or PTFF-STO, the CPUs in the configuration do not necessarily recognize this change for purposes of clock comparator interruptions until one of the following instructions is issued: SET CLOCK COMPARATOR (SCKC), STORE CLOCK (STCK), or STORE CLOCK EXTENDED (STCKE). If a CPU is in the wait state when this change occurs, it may not recognize the change until it leaves the wait state and one of the aforementioned instructions is executed.

The subclass-mask bit is in bit position 52 of control register 0. This bit is initialized to zero.

The clock-comparator condition is indicated by an external-interruption code of 1004 hex.

CPU Timer

An interruption request for the CPU timer exists whenever the CPU-timer value is negative (bit 0 of the CPU timer is one). If the value is made positive before the request is honored, the request does not remain pending, and no interruption occurs. Conversely, the request is not cleared by the interruption, and, if the condition persists, more than one interruption may occur from a single occurrence of the condition.

When the TOD-clock-steering facility is not installed, and the TOD clock is set or changes state, interruption conditions, if any, that are due to the CPU timer may or may not be recognized for up to 1.048576 seconds after the change.

When the TOD-clock-steering facility is installed, CPU timer interruptions may or may not be recognized while the physical clock is in the stopped state. After the physical clock enters the set state, interruption conditions for the CPU timer are not necessarily recognized until SET CPU TIMER (SPT) is issued. CPU timer interruptions are not affected by changes to the logical TOD clock.

The subclass-mask bit is in bit position 53 of control register 0. This bit is initialized to zero.

The CPU-timer condition is indicated by an external-interruption code of 1005 hex.

Emergency Signal

An interruption request for an emergency signal is generated when the CPU accepts the emergency-

signal order specified by a SIGNAL PROCESSOR instruction addressing this CPU. The instruction may have been executed by this CPU or by another CPU in the configuration. The request is preserved and remains pending in the receiving CPU until it is cleared. The pending request is cleared when it causes an interruption and by CPU reset.

Facilities are provided for holding a separate emergency-signal request pending in the receiving CPU for each CPU in the configuration, including the receiving CPU itself.

The subclass-mask bit is in bit position 49 of control register 0. This bit is initialized to zero.

The emergency-signal condition is indicated by an external-interruption code of 1201 hex. The address of the CPU that executed the SIGNAL PROCESSOR instruction is stored at real locations 132-133.

External Call

An interruption request for an external call is generated when the CPU accepts the external-call order specified by a SIGNAL PROCESSOR instruction addressing this CPU. The instruction may have been executed by this CPU or by another CPU in the configuration. The request is preserved and remains pending in the receiving CPU until it is cleared. The pending request is cleared when it causes an interruption and by CPU reset.

Only one external-call request, along with the processor address, may be held pending in a CPU at a time.

The subclass-mask bit is in bit position 50 of control register 0. This bit is initialized to zero.

The external-call condition is indicated by an external-interruption code of 1202 hex. The address of the CPU that executed the SIGNAL PROCESSOR instruction is stored at real locations 132-133.

Interrupt Key

An interruption request for the interrupt key is generated when the operator activates that key. The request is preserved and remains pending in the CPU until it is cleared. The pending request is cleared when it causes an interruption and by CPU reset.

When the interrupt key is activated while the CPU is in the load state, it depends on the model whether an interruption request is generated or the condition is lost.

The subclass-mask bit is in bit position 57 of control register 0. This bit is initialized to one.

The interrupt-key condition is indicated by an external-interruption code of 0040 hex.

Malfunction Alert

An interruption request for a malfunction alert is generated when another CPU in the configuration enters the check-stop state or loses power. The request is preserved and remains pending in the receiving CPU until it is cleared. The pending request is cleared when it causes an interruption and by CPU reset.

Facilities are provided for holding a separate malfunction-alert request pending in the receiving CPU for each of the other CPUs in the configuration. Removal of a CPU from the configuration does not generate a malfunction-alert condition.

The subclass-mask bit is in bit position 48 of control register 0. This bit is initialized to zero.

The malfunction-alert condition is indicated by an external-interruption code of 1200 hex. The address of the CPU that generated the condition is stored at real locations 132-133.

Measurement Alert

The measurement-alert external interruption is described in the publication *The Set-Program-Parameter and CPU-Measurement Facilities* (SA23-2260).

The measurement-alert subclass-mask bit is in bit position 58 of control register 0. This bit is initialized to one, which enables the interruption.

The measurement-alert condition is indicated by an external interruption code of 1407 hex. The interrup-

tion parameter stored at real locations 128-131 indicate the cause or causes for the interruption.

Service Signal

An interruption request for a service signal is generated upon the completion of certain configuration-control and maintenance functions, such as those initiated by means of the model-dependent DIAGNOSE instruction. A 32-bit parameter is provided with the interruption to assist the program in determining the operation for which the interruption is reported.

Service signal is a floating interruption condition and is presented to the first CPU in the configuration which can perform the interruption. The interruption condition is cleared when it causes an interruption in any one of the CPUs and also by subsystem reset.

The subclass-mask bit is in bit position 54 of control register 0. This bit is initialized to zero.

The service-signal condition is indicated by an external-interruption code of 2401 hex. A 32-bit parameter is stored at real locations 128-131.

Timing Alert

An interruption request for a timing alert is generated when an ETR-timing-alert or STP-timing-alert condition is detected.

If the same timing-alert condition occurs more than once before the interruption occurs, the request is generated only once.

When the STP-floating-interrupt facility is not installed or is not enabled for STP timing-alert floating interruptions, the timing-alert condition causes a pendable interruption request to be generated for each CPU in the configuration and to remain pending at each CPU until it is cleared. The pending request is cleared when it causes an interruption at the CPU and by CPU reset.

When the STP-floating-interrupt facility is installed and is enabled for STP timing-alert floating interruptions, an STP timing-alert condition causes a floating interruption request to be generated.

The subclass-mask bit is in bit position 59 of control register 0. This bit is initialized to zero.

The timing-alert condition is indicated by an external interruption code of 1406 hex. The interruption parameter stored at real locations 128-131 indicate the reason or reasons for the signal.

ETR-Timing-Alert Condition

An ETR-timing-alert condition is detected when a port-availability change occurs at any port in the current CPC-port group or when an ETR alert occurs. The terms specific to the ETR are not defined in this publication.

STP-Timing-Alert Condition

An STP-timing-alert condition is detected when any of the following occur:

Timing Status Change: A timing-status-change condition is detected when a timing-status change has occurred for the configuration. Timing status for a configuration includes the timing mode, timing state, the STP-clock-source state, and certain external time source conditions.

A timing-state change from the synchronized state to the unsynchronized or stopped state is reported as an ETR- or STP-sync-check machine-check condition rather than a timing-alert external interruption condition.

An STP-clock-source-state change from the usable state to the not-usable is reported as a clock-source-error machine-check condition rather than a timing-alert external interruption condition.

Link Availability Change: A link-availability-change condition is detected when the availability of an STP link has changed. The condition is detected whenever an STP link state that was unavailable becomes available for STP communication, or when a link that was available becomes unavailable for STP communication.

A timing-alert condition is not generated by a link-availability-change condition if a CTN configuration-change machine check condition is detected concurrently with the link-availability change.

Time Control Parameter Change: The time control-parameter-change condition is detected when any of the time-control parameters change for the configuration, including when a time-control-parameter change is scheduled. Time-control parameters for a configuration include the following:

- Total-time offset (the combination of time-zone offset and DST offset)
- Leap seconds offset
- Time-zone offset
- Daylight-savings-time (DST) offset
- Scheduled changes to any of the above

A 32-bit parameter is associated with the interruption and is stored at real locations 128-131. The field is defined as shown below:

Bit	Meaning
0-13	Reserved
14	Timing-status change
15	Link-availability change
16	Time-control-parameter change
17-31	Reserved

Multiple alert conditions may be indicated concurrently in the external-interruption parameter field.

Warning Track

An interruption request for a warning-track event may be generated to inform the control program it is nearing the end of the current execution interval on a shared CPU. The interruption request is a pending-condition type which may be generated when the configuration is registered and is enabled for warning-track interruptions.

When the configuration is enabled for multithreading, a warning-track interruption may be made pending for each CPU of a core.

The subclass-mask bit is in bit position 30 of control register 0. This bit is initialized to zero.

The warning-track condition is indicated by an external-interruption code of 1007 hex.

I/O Interruption

The input/output (I/O) interruption provides a means by which the CPU responds to conditions originating in I/O devices and the channel subsystem.

A request for an I/O interruption may occur at any time, and more than one request may occur at the same time. The requests are preserved and remain pending until accepted by a CPU, or until cleared by some other means, such as subsystem reset.

The I/O interruption occurs at the completion of a unit of operation. Priority is established among requests so that in each CPU only one interruption request is processed at a time. Priority among requests for interruptions of differing I/O-interruption subclasses is according to the numerical value of the I/O-interruption subclass (with zero having the highest priority), in conjunction with the I/O-interruption subclass-mask settings in control register 6. For more details, see “I/O Interruptions” on page 16-1.

When a CPU becomes enabled for I/O interruptions and the channel subsystem has established priority for a pending I/O-interruption condition, the interruption occurs at the completion of the instruction execution or interruption that causes the enabling.

In the z/Architecture architectural mode, an I/O interruption causes the old PSW to be stored at real locations 368-383 and a new PSW to be fetched from real locations 496-511. In the ESA/390-compatibility mode, an I/O interruption causes the short-format old PSW to be stored at real locations 56-63 and a short-format new PSW to be fetched from real locations 120-127.

Additional information, in the form of an eight-byte I/O-interruption code, is stored at real locations 184-195. The I/O-interruption code consists of a 32-bit subsystem-identification word, a 32-bit interruption parameter, and a 32-bit I/O-interruption identification word.

An I/O interruption can occur only while a CPU is enabled for the interruption subclass presenting the request. The I/O-mask bit, bit 6 of the PSW, and the I/O-interruption subclass mask in control register 6 determine whether the CPU is enabled for a particular I/O interruption.

I/O interruptions are grouped into eight I/O-interruption subclasses, numbered from 0-7. Each I/O-interruption subclass has an associated I/O-interruption subclass-mask bit in bit positions 32-39 of control register 6. Each subchannel has an I/O-interruption subclass value associated with it. The CPU is enabled for I/O interruptions of a particular I/O-interruption subclass only when PSW bit 6 is one and the associated I/O-interruption subclass-mask bit in control register 6 is also one. If the corresponding I/O-

interruption subclass-mask bit is zero, then the CPU is disabled for I/O interruptions with that subclass value. I/O interruptions for all subclasses are disallowed when PSW bit 6 is zero.

Machine-Check Interruption

The machine-check interruption is a means for reporting to the program the occurrence of equipment malfunctions. Information is provided to assist the program in determining the source of the fault and extent of the damage.

In the z/Architecture architectural mode, a machine-check interruption causes the old PSW to be stored at real locations 352-367 and a new PSW to be fetched from real locations 480-495. In the ESA/390-compatibility mode, a machine-check interruption causes the short-format old PSW to be stored at real locations 48-55 and a short-format new PSW to be fetched from real locations 112-119.

The cause and severity of the malfunction are identified by a 64-bit machine-check-interruption code stored at real locations 232-239 and an indication of the architectural mode to be stored at real location 163. In the z/Architecture architectural mode, further information identifying the cause of the interruption and the location of the fault may be stored at real locations 244-255 and 4608-5119. In the ESA/390-compatibility mode, further information identifying the cause of the interruption and the location of the fault may be stored at real locations 216-231, 244-251, 288-511, and in the machine-check extended save area pointed to by real locations 212-215.

The interruption action and the storing of the associated information are under the control of PSW bit 13 and bits in control register 14. See Chapter 11, “Machine-Check Handling” for more detailed information.

Program Interruption

Program interruptions are used to report exceptions and events which occur during execution of the program.

In the z/Architecture architectural mode, a program interruption causes the old PSW to be stored at real locations 336-351 and a new PSW to be fetched from real locations 464-479. In the ESA/390-compatibility mode, a program interruption causes the short-format old PSW to be stored at real locations 40-47 and a short-format new PSW to be fetched from real locations 104-111.

The cause of the interruption is identified by the interruption code. The interruption code is placed at real locations 142-143, the instruction-length code is placed in bit positions 5 and 6 of the byte at real location 141 with the rest of the bits set to zeros, and zeros are stored at real location 140. When a transaction is aborted due to a program interruption, the instruction-length code is respective to the instruction at which the exception condition was detected. For some causes, additional information identifying the reason for the interruption is stored at real locations 144-183.

If the PER-3 facility is installed, then, as part of the program interruption action, the contents of the breaking-event-address register are placed in real storage locations 272-279.

Except for PER events and the crypto-operation exception, the condition causing the interruption is indicated by a coded value placed in the rightmost seven bit positions of the interruption code. Only one condition at a time can be indicated.

PER events are indicated by setting bit 8 of the interruption code to one. When this is the only condition, bits 0-7 and 9-15 are also set to zeros. When a PER event is indicated concurrently with another program-interruption condition, bit 8 is one, and bits 0-7 and 9-15 are set as for the other condition.

The crypto-operation exception is indicated by an interruption code of 0119 hex, (or 0199, 0319, or 0399 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

When a program interruption is recognized during transactional execution, bit 6 of the interruption code is set to one. Bits 0-5 of the interruption code are set to zero.

When there is a corresponding mask bit, a program interruption can occur only when that mask bit is one. The program mask in the PSW controls four of the exceptions, the IEEE masks in the FPC register control the IEEE exceptions, bit 33 in control register 0

controls whether SET SYSTEM MASK causes a special-operation exception, bits 48-63 in control register 8 control interruptions due to monitor events, and a hierarchy of masks control interruptions due to PER events. When any controlling mask bit is zero, the condition is ignored; the condition does not remain pending.

Programming Notes:

1. When the new PSW for a program interruption has a PSW-format error or causes an exception to be recognized in the process of instruction fetching, a string of program interruptions may occur. See "Priority of Interruptions" on page 6-57 for a description of how such strings are terminated.

2. Some of the conditions indicated as program exceptions may be recognized also by the channel subsystem, in which case the exception is indicated in the subchannel-status word or extended-status word.

Data-Exception Code (DXC)

When a data exception causes a program interruption, a data-exception code (DXC) is stored at location 147, and zeros are stored at locations 144-146. The DXC distinguishes between the various types of data-exception conditions. When the AFP-register (additional floating-point register) control bit, bit 45 of control register 0, is one, the DXC is also placed in the DXC field of the floating-point-control (FPC) register. The DXC field in the FPC register remains unchanged when any other program exception is reported. The DXC is an 8-bit code indicating the specific cause of a data exception. The data exceptions and data-exception codes are shown in Figure 6-2 and Figure 6-3 on page 6-18.

Priority of Program Interruptions for Data Exceptions

DXC 2 and 3 are mutually exclusive and are of higher priority than any other DXC. Thus, for example, DXC 2 (BFP instruction) takes precedence over any IEEE exception; and DXC 3 (DFP instruction) takes precedence over any IEEE exception or simulated IEEE exception. As another example, if the conditions for both DXC 3 (DFP instruction) and DXC 1 (AFP register) exist, DXC 3 is reported.

When both a specification exception and an AFP-register data exception or a vector-instruction data exception apply, it is unpredictable which one is reported.

DXC (Hex)	Data Exception
00	General operand
01	AFP register
02	BFP instruction
03	DFP instruction
04	Quantum Exception
07	Simulated Quantum Exception
08	IEEE inexact and truncated
0B	Simulated IEEE inexact
0C	IEEE inexact and incremented
10	IEEE underflow, exact
13	Simulated IEEE underflow, exact
18	IEEE underflow, inexact and truncated
1B	Simulated IEEE underflow, inexact
1C	IEEE underflow, inexact and incremented
20	IEEE overflow, exact
23	Simulated IEEE overflow, exact
28	IEEE overflow, inexact and truncated
2B	Simulated IEEE overflow, inexact
2C	IEEE overflow, inexact and incremented
40	IEEE division by zero
43	Simulated IEEE division by zero
80	IEEE invalid operation
83	Simulated IEEE invalid operation
FE	Vector instruction
FF	Compare-and-trap instruction

Figure 6-2. Data-exception codes (DXC)

Exception	Applicable Instruction Types	CR0.45	FPC Mask	FPC Flag	DXC (Binary)	Instruction Action	DXC Placed in Real Loc. 147	DXC Placed in FPC Byte 2
General operand	Various[1]	0	none	none	0000 0000	Suppress or Terminate	Yes	No
		1					Yes	Yes
AFP register	FPS & HFP	0*	none	none	0000 0001	Suppress	Yes	No
BFP instruction	BFP	0*	none	none	0000 0010	Suppress	Yes	No
DFP instruction	DFP	0*	none	none	0000 0011	Suppress	Yes	No
IEEE invalid operation	ICMP	1*	0.0	1.0	1000 0000	Suppress[2]	Yes	Yes
IEEE division by zero	ICMP	1*	0.1	1.1	0100 0000	Suppress[2]	Yes	Yes
IEEE overflow	ICMP	1*	0.2	1.2	0010 xy00	Complete	Yes	Yes
IEEE underflow	ICMP	1*	0.3	1.3	0001 xy00	Complete	Yes	Yes
IEEE inexact	ICMP	1*	0.4	1.4	0000 1y00	Complete	Yes	Yes
Quantum Exception	ICMP	1*	0.5	1.5	0000 0100	Complete	Yes	Yes
Simulated IEEE invalid operation	IXS	1*	0.0	1.0	1000 0011	Complete	Yes	Yes
Simulated IEEE division by zero	IXS	1*	0.1	1.1	0100 0011	Complete	Yes	Yes
Simulated IEEE overflow	IXS	1*	0.2	1.2	0010 w011	Complete	Yes	Yes
Simulated IEEE underflow	IXS	1*	0.3	1.3	0001 w011	Complete	Yes	Yes
Simulated IEEE inexact	IXS	1*	0.4	1.4	0000 1011	Complete	Yes	Yes
Simulated Quantum Exception	IXS	1*	0.5	1.5	0000 0111	Complete	Yes	Yes
Vector instruction	VEC	0†	none	none	1111 1110	Suppress	Yes	Unp
		1‡					Yes	Yes

Figure 6-3. Data Exceptions

Exception	Applicable Instruction Types	CR0.45	FPC Mask	FPC Flag	DXC (Binary)	Instruction Action	DXC Placed in Real Loc. 147	DXC Placed in FPC Byte 2
Compare-and-trap instruction	CT & LT	0	none	none	1111 1111	Complete	Yes	No
		1					Yes	Yes

Explanation:

[1] General-operand data exception applies to the decimal instructions (Chapter 8), the general instructions COMPRESSION CALL, CONVERT TO BINARY, and PERFORM RANDOM NUMBER OPERATION (Chapter 7), and the DFP instructions CONVERT FROM PACKED, CONVERT FROM SIGNED PACKED, CONVERT FROM UNSIGNED PACKED, and CONVERT FROM ZONED (Chapter 20).

[2] When the FPC mask bit corresponding to the exception condition is one, the DXC is stored in the FPC register, even though the resulting data-exception program interruption is considered to be suppressing.

0* This exception is recognized only when CR0.45 is zero.

1* This exception is recognized only when CR0.45 is one.

0† This exception may be recognized if CR0.46 is zero or one

1‡ This exception is recognized only when CR0.46 is zero.

w For simulated IEEE overflow and simulated IEEE underflow, bit 4 of the DXC is set to bit 4 of the signaling flags in the FPC register.

xy For IEEE overflow and IEEE underflow, bits 4 and 5 of the DXC are set to 00, 10, or 11 binary, indicating that the result is exact, inexact and truncated, or inexact and incremented, respectively.

y For IEEE inexact, bit 5 of the DXC is set to zero or one, indicating that the result is inexact and truncated or inexact and incremented, respectively.

BFP Binary-floating-point instructions (Chapter 19).

CT COMPARE AND TRAP, COMPARE IMMEDIATE AND TRAP, COMPARE LOGICAL AND TRAP, or COMPARE LOGICAL IMMEDIATE AND TRAP instructions.

DFP Decimal-floating-point instructions (Chapter 20).

FPS Floating-point-support instructions (Chapter 9).

HFP Hexadecimal-floating-point instructions (Chapter 18).

ICMP IEEE computational instructions.

IXS IEEE-exception-simulation instructions (LOAD FPC AND SIGNAL and SET FPC AND SIGNAL).

LT Load-and-trap facility instructions.

UNP Unpredictable if FPC byte 2 is updated if CR0.46 is one. Otherwise, FPC Byte 2 is not updated.

VEC Vector instructions (Chapters 21, 22, 23, 24, and 25).

Figure 6-3. Data Exceptions (Continued)

Programming Note: The data-exception code (DXC) in bits 16-23 of the floating-point control register (FPCR) is primarily intended for use by floating-point applications that rely on the enablement of the advanced-floating-point (AFP) control, bit 45 of control register 0. When a data-exception program interruption occurs as a result of the execution of a compare-and-trap or a load-and-trap facility instruction (DXC 255), the DXC in the FPCR may contain an unpredictable value under the following conditions.

- For a control program, an instruction examining the FPCR (for example, EXTRACT FPC) is the first instruction that is subject to the AFP control to be executed since the last initial-CPU reset.
- For a task running under a control program such as z/OS, an instruction examining the FPCR is the first instruction that is subject to the AFP control to be executed by the task.

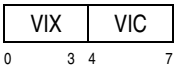
This unpredictability is the result of the control program or task being dispatched with the AFP control initially set to zero, and only applies to DXC 255 in the FPCR. Subsequent inspections of the DXC in the FPCR will yield predictable values.

If the program always needs to observe a predictable compare-and-trap DXC value in the FPCR, it should first issue any instruction that is subject to the AFP control (for example EXTRACT FPC) before issuing an instruction that causes the data-exception program interruption for the compare-and-trap condition. Alternatively, the program can inspect the DXC stored in real location 147, as copied into a control-program-supplied diagnostic work area; the DXC stored at location 147 is predictable in all cases.

Vector-Exception Code

When a vector-processing exception causes a program interruption, a vector-exception code (VXC) is stored at location 147, and zeros are stored at locations 144-146. The VXC is also placed in the DXC field of the floating-point-control (FPC) register if bit 45 of control register 0 is one. When bit 45 of control register 0 is zero and bit 46 of control register 0 is one, the DXC field of the FPC register and the contents of storage at location 147 are unpredictable. The VXC distinguishes between various types of vector floating point exceptions and which element caused the exception.

The VXC has the following format:



Bits 0-3 of the VXC are the vector index (VIX). The index in the VXC is always the index of the source element that caused the trapping exception, except for one special case in VECTOR LOAD LENGTHENED (see the programming note on page 24-26). If trapping vector-processing exception conditions exist for multiple elements, the exception of the lowest-indexed source element is recognized.

Bits 4-7 of the VXC are the vector interrupt code (VIC). This field can have the following values:

0001 - IEEE invalid operation
0010 - IEEE division by zero
0011 - IEEE overflow
0100 - IEEE underflow
0101 - IEEE inexact

Program-Interruption Conditions

The following is a detailed description of each program-interruption condition.

Addressing Exception

An addressing exception is recognized when the CPU attempts to reference a main-storage location that is not available in the configuration. A main-storage location is not available in the configuration when the location is not installed, when the storage unit is not in the configuration,or when power is off in the storage unit. An address designating a storage location that is not available in the configuration is referred to as invalid.

The operation is suppressed when the address of the instruction is invalid. Similarly, the operation is suppressed when the address of the target instruction of an execute-type instruction (EXECUTE or EXECUTE RELATIVE LONG) is invalid. Also, the unit of operation is suppressed when an addressing exception is encountered in accessing a table or table entry. The tables and table entries to which the rule applies are the dispatchable-unit-control table, the primary ASN-second-table entry, and entries in the access list, region first table, region second table, region third table, segment table, page table, linkage table, linkage-first table, linkage-second table, entry table, ASN first table, ASN second table, authority table, linkage stack, and trace table. Addressing exceptions result in suppression when they are encountered for references to the region first table, region second table, region third table, segment table, and page table, in both implicit references for dynamic address translation and references associated with the execution of LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, STORE REAL ADDRESS, and TEST PROTECTION. Similarly, addressing exceptions for accesses to the dispatchable-unit-control table, primary ASN-second-table entry, access list, ASN second table, or authority table result in suppression when they are encountered in access-register translation done either implicitly or as part of LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, STORE REAL ADDRESS, TEST ACCESS, or TEST PROTECTION.

Except for some specific instructions whose execution is suppressed, the operation is terminated for an operand address that can be translated but designates an unavailable location. See Figure 6-4 on page 6-21.

For termination, changes may occur only to result fields. In this context, the term “result field” includes the condition code, registers, and any storage locations that are provided and that are designated to be changed by the instruction. Therefore, if an instruction is due to change only the contents of a field in storage, and every byte of the field is in a location that is not available in the configuration, the operation is suppressed. When part of an operand location is available in the configuration and part is not, storing may be performed in the part that is available in the configuration.

When an addressing exception occurs during the fetching of an instruction or during the fetching of a DAT table entry associated with an instruction fetch, it is unpredictable whether the ILC is 1, 2, or 3. When the exception is associated with fetching the target of EXECUTE, the ILC is 2. When the exception is associated with fetching the target of EXECUTE RELATIVE LONG, the ILC is 3.

In all cases of addressing exceptions not associated with instruction fetching, the ILC is 1, 2, or 3, indicating the length of the instruction that caused the reference.

An addressing exception is indicated by a program-interruption code of 0005 hex (or 0085, 0205, or 0285 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Exception	Action on			
	Table-Entry Fetch[1]	**Table-Entry Store**[2]	**Instruction Fetch**	**Operand Reference**
Addressing exception	Suppress	Suppress	Suppress	Suppress for IPTE, LASP, LPSW, LPSWE, MSCH, PLO[6], RP, SCKC, SPT, SPX, SSCH, SSM, STCRW, STNSM, STOSM, TPI, and TPROT Terminate for all others.[4]
Protection exception for key-controlled protection	—	—	Suppress	Suppress for IPTE, LASP, LPSW, LPSWE, MSCH, PLO[6], RP, SCKC, SPT, SPX, SSCH, SSM, STCRW, STNSM, STOSM, and TPI[5] Terminate for all others.[4]
Protection exception for access-list-controlled protection	—	—	—	Suppress
Protection exception for instruction-execution protection	—	—	Suppress	—
Protection exception for DAT protection	—	Suppress[3]	—	Suppress[5]
Protection exception for low-address protection	—	Suppress	—	Suppress for IPTE, PLO[6], STCRW, STNSM, STOSM, and TPI[5]. Terminate for all others.[4]

Explanation:

— Not applicable

1 Table entries include region table, segment table, page table, linkage table, linkage-first table, linkage-second table, entry table, ASN first table, ASN second table, authority table, dispatchable-unit-control table, primary ASN-second-table entry, access list, and linkage stack.

2 Table entries include linkage stack and trace table.

3 DAT protection applies to the linkage stack but not the trace table.

4 For termination, changes may occur only to result fields. In this context, "result field" includes condition code, registers, and storage locations, if any, which are designated to be changed by the instruction. However, no change is made to a storage location or a storage key when the reference causes an access exception. Therefore, if an instruction is due to change only the contents of a field in main storage, and every byte of that field would cause an access exception, the result is the same as if the operation had been suppressed. The action may be, for key-controlled protection and low-address protection, suppression instead of termination; see "Suppression on Protection" on page 3-15.

5 When the effective address of TPI is zero, the store access is to implicit real locations 184-191, and key-controlled protection, DAT protection, and low-address protection do not apply.

6 Suppression occurs only for the compare-and-load and compare-and-swap operations.

Figure 6-4. Summary of Action for Addressing and Protection Exceptions

AFX-Translation Exception

An AFX-translation exception is recognized when, during ASN translation in the space-switching form of PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, or SET SECONDARY ASN WITH INSTANCE, or during ASN translation in PROGRAM RETURN when the restored SASN does not equal the restored PASN, bit 0 of the ASN-first-table entry used is not zero.

The ASN being translated is stored at real locations 174 and 175, and real locations 172 and 173 are set with zeros.

The operation is nullified.

The instruction-length code is 1 or 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The AFX-translation exception is indicated by a program-interruption code of 0020 hex (or 00A0 hex if a concurrent PER event is indicated).

ALEN-Translation Exception

An ALEN-translation exception is recognized during access-register translation when either:

1. The access register used contains an access-list-entry number that designates an access-list entry which is beyond the end of the access list designated by the effective access-list designation.
2. Bit 0 of the access-list entry is not zero.

The number of the access register is stored in bit positions 4-7 of real location 160, and bits 0-3 of the location are set to zeros.

The operation is nullified.

The instruction-length code is 1, 2, or 3.

The ALEN-translation exception is indicated by a program-interruption code of 0029 hex (or 00A9, 0229, or 02A9 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

ALE-Sequence Exception

An ALE-sequence exception is recognized during access-register translation when the access register used contains an access-list-entry sequence number (ALESN) which is not equal to the ALESN in the access-list entry that is designated by the access register.

The number of the access register is stored in bit positions 4-7 at real location 160, and bits 0-3 are set to zeros.

The operation is nullified.

The instruction-length code is 1, 2, or 3.

The ALE-sequence exception is indicated by a program-interruption code of 002A hex (or 00AA, 022A, or 02AA hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

ALET-Specification Exception

An ALET-specification exception is recognized during access-register translation when bit positions 0-6 of the access-list-entry token in the access register used do not contain all zeros. However, when access-register 0 is used, except in TEST ACCESS, it is treated as containing all zeros, and this exception is not recognized. TEST ACCESS uses the actual contents of access register 0.

The operation is suppressed.

The instruction-length code is 1, 2, or 3.

The ALET-specification exception is indicated by a program-interruption code of 0028 hex (or 00A8, 0228, or 02A8 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

ASCE-Type Exception

An ASCE-type exception is recognized when any of the following is true during dynamic address translation:

1. The address-space-control element being used is a region-second-table designation, and bits 0-10 of the virtual address being translated are not all zeros.
2. The address-space-control element being used is a region-third-table designation, and bits 0-21 of the virtual address being translated are not all zeros.

3. The address-space-control element being used is a segment-table designation, and bits 0-32 of the virtual address being translated are not all zeros.

The exception is recognized as part of the execution of the instruction that needs the translation of an instruction or operand address, except for the operand address in LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, and TEST PROTECTION, in which case the condition is indicated by the setting of the condition code.

When an interruption occurs, information about the virtual address causing the exception is stored at real locations 168-175 and conditionally at real locations 160 and 162. See "Assigned Storage Locations" on page 3-73 for a detailed description of this information.

The unit of operation is nullified.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during a reference to the target of EXECUTE, the ILC is 2. When the exception occurs during a reference to the target of EXECUTE RELATIVE LONG, the ILC is 3.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The ASCE-type exception is indicated by a program-interruption code of 0038 hex (or 00B8, 0238, or 02B8 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

ASTE-Instance Exception

An ASTE-instance exception is recognized when the ASN-and-LX-reuse facility is installed, and any of the following is true:

1. During ASN translation in PROGRAM TRANSFER WITH INSTANCE with space switching, the ASTEIN in bit positions 0-31 of general register R_1 is not equal to the ASTEIN in the located ASTE.
2. During PASN translation in PROGRAM RETURN with space switching, the ASN-and-LX-reuse control in control register 0 is one and the PASTEIN in the linkage-stack state entry is not equal to the ASTEIN in the located ASTE.
3. During ASN translation in SET SECONDARY ASN WITH INSTANCE with space switching, the ASTEIN in bit positions 0-31 of general register R_1 is not equal to the ASTEIN in the located ASTE.
4. During SASN translation in PROGRAM RETURN to current primary or with space switching, the ASN-and-LX-reuse control in control register 0 is one, and the SASTEIN in the linkage-stack state entry is not equal to the ASTEIN in the located ASTE.

Information is stored as follows:

- In cases 1 and 2, bit 2 of real location 160 is set to one, and bits 0, 1, and 3-7 are set to zeros.
- In cases 3 and 4, bit 3 of real location 160 is set to one, and bits 0-2 and 4-7 are set to zeros.

The operation is nullified.

The instruction-length code is 1, 2, or 3.

The ASTE-instance exception is indicated by a program-interruption code of 002F hex (or 00AF hex if a concurrent PER event is indicated).

ASTE-Sequence Exception

An ASTE-sequence exception is recognized when any of the following is true:

1. During access-register translation, except as in 2, the access-list entry used contains an ASN-second-table-entry sequence number (ASTESN) which is not equal to the ASTESN in the ASN-second-table entry that is designated by the access-list entry. The access-list entry is the one designated by the access register used.
2. During access-register translation of ALET 1 by BRANCH IN SUBSPACE GROUP, the subspace ASTESN (SSASTESN) in the dispatchable-unit control table (DUCT) is not equal to the ASTESN in the subspace ASTE designated by the subspace-ASTE origin (SSASTEO) in the DUCT.
3. During a subspace-replacement operation, the subspace ASTESN (SSASTESN) in the dispatchable-unit control table (DUCT) is not equal to the ASTESN in the subspace ASTE desig-

nated by the subspace-ASTE origin (SSASTEO) in the DUCT.

In the first and second cases, the number of the access register is stored in bit positions 4-7 at real location 160, and bits 0-3 are set to zeros. In the third case, all zeros are stored at real location 160.

The operation is nullified.

The instruction-length code is 1, 2, or 3.

The ASTE-sequence exception is indicated by a program-interruption code of 002C hex (or 00AC, 022C, or 02AC hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Programming Note: The storing of zeros at real location 160 in the case of an ASTE-sequence exception recognized during a subspace-replacement operation is a unique indication since the use of access register 0 in access-register translation cannot result in the exception.

ASTE-Validity Exception

An ASTE-validity exception is recognized when any of the following is true:

1. During access-register translation, except as in 2, the access-list entry used designates an ASN-second-table entry in which bit 0 is not zero. The access-list entry is the one designated by the access register used.
2. During access-register translation of ALET 1 by BRANCH IN SUBSPACE GROUP, the subspace-ASTE origin (SSASTEO) in the dispatchable-unit control table designates an ASN-second-table entry in which bit 0 is not zero.
3. During a subspace-replacement operation, the subspace-ASTE origin (SSASTEO) in the dispatchable-unit control table designates an ASN-second-table entry in which bit 0 is not zero.

In the first and second cases, the number of the access register is stored in bit positions 4-7 at real location 160, and bits 0-3 are set to zeros. In the third case, all zeros are stored at real location 160.

The operation is nullified.

The instruction-length code is 1, 2, or 3.

The ASTE-validity exception is indicated by a program-interruption code of 002B hex (or 00AB, 022B, or 02AB hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Programming Note: The storing of zeros at real location 160 in the case of an ASTE-validity exception recognized during a subspace-replacement operation is a unique indication since the use of access register 0 in access-register translation cannot result in the exception.

ASX-Translation Exception

An ASX-translation exception is recognized when, during execution of the space-switching form of PROGRAM CALL, during ASN translation in the space-switching form of PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, or SET SECONDARY ASN WITH INSTANCE, or during ASN translation in PROGRAM RETURN when the restored SASN does not equal the restored PASN, bit 0 of the ASN-second-table entry used is not zero.

The ASN being translated is stored at real locations 174 and 175, and real locations 172 and 173 are set with zeros.

The operation is nullified.

The instruction-length code is 1 or 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The ASX-translation exception is indicated by a program-interruption code of 0021 hex (or 00A1 hex if a concurrent PER event is indicated).

Crypto-Operation Exception

A crypto-operation exception is recognized when a crypto-facility instruction is executed while bit 61 of control register 0 is zero on a CPU which has the crypto facility installed and available. The crypto-operation exception is also recognized when a crypto-facility instruction is executed and the crypto facility is not installed or available on this CPU, but the facility can be made available to the program either on this CPU or another CPU in the configuration.

When a crypto-facility instruction is executed and the crypto facility is not installed on any CPU which is or

can be placed in the configuration, it depends on the model whether a crypto-operation exception or an operation exception is recognized.

The operation is nullified when the crypto-operation exception is recognized.

The instruction-length code is 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The crypto-operation exception is indicated by a program-interruption code of 0119 hex (or 0199, 0319, or 0399 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Data Exception

The data exceptions are shown in Figure 6-3 on page 6-18. A mask bit may or may not control whether an interruption occurs, as noted for each exception.

When a non-maskable data exception is recognized, a program interruption for a data exception always occurs.

Each of the IEEE exceptions is controlled by a mask bit in the floating-point-control (FPC) register. The handling of these exceptions is described in the section "IEEE Exceptions" on page 9-18.

A data exception is recognized for the following cases:

- **General-operand** data exception is recognized when an instruction which operates on decimal operands encounters invalid decimal digit or sign codes or has its operands specified improperly. The operation is suppressed, except that, for EDIT and EDIT AND MARK, it is model dependent whether the operation is suppressed or terminated. See the section "General-Operand Data Exception" on page 8-5 for details. A general-operand data exception is also recognized when COMPRESSION CALL encounters errors in its dictionaries, in which case it is model dependent whether the operation is suppressed or terminated. A general-operand data exception is also recognized when the reseed counter is zero for a PERFORM RANDOM NUMBER OPERATION generate operation, in which case the operation is suppressed. The general-operand data exception is reported with DXC 0.

  Note: In earlier versions of the architecture, the general-operand data exception was known as the decimal-operand data exception.

- **AFP-register** data exception is recognized when bit 45 of control register 0 is zero, and a floating-point-support (FPS) instruction or a hexadecimal-floating-point (HFP) instruction specifies a floating-point register other than 0, 2, 4, or 6. AFP-register data exception is also recognized when bit 45 of control register 0 is zero and a PFPO instruction is executed. The operation is suppressed and is reported with DXC 1.

- **BFP-instruction** data exception is recognized when bit 45 of control register 0 is zero and a BFP instruction is executed. The operation is suppressed and is reported with DXC 2.

- **Compare-and-trap-instruction** data exception is recognized when the operands compared by COMPARE AND TRAP, COMPARE IMMEDIATE AND TRAP, COMPARE LOGICAL AND TRAP, or COMPARE LOGICAL IMMEDIATE AND TRAP match the conditions specified by the M_3 field of the instruction. The operation is completed and is reported with DXC FF hex.

  When the load-and-trap facility is installed, compare-and-trap-instruction data exception is recognized when all zeros are loaded into the first operand of LOAD AND TRAP, LOAD HIGH AND TRAP, LOAD LOGICAL AND TRAP, and LOAD LOGICAL THIRTY ONE BITS AND TRAP. The operation is completed and is reported with DXC FF hex.

- **DFP-instruction** data exception is recognized when bit 45 of control register 0 is zero and a DFP instruction is executed. The operation is suppressed and is reported with DXC 3.

- **IEEE-exception** data exceptions are recognized when an IEEE computational instruction encounters an enabled exceptional condition. The operation is suppressed or completed, depending on the type of exception. See the section "IEEE Exceptions" on page 19-7 for details.

- **Simulated IEEE-exception** data exceptions are recognized when an IEEE-exception-simulation instruction (LOAD FPC AND SIGNAL or SET

FPC AND SIGNAL) encounters an enabled signaling flag. The operation is completed. See the section "IEEE Exceptions" on page 19-7 for details.

- **Vector-instruction** data exceptions are recognized when bit 46 of control register 0 is zero and a vector instruction is executed (a vector instruction any instruction defined in chapters 21, 22, 23, 24, or 25). It is unpredictable if a data exception is recognized if bit 45 of control register 0 is zero, bit 46 of control register 0 is one, and a vector instruction is executed. The operation is suppressed and is reported with DXC FE hex.

The instruction-length code is 1, 2, or 3.

The data exception is indicated by a program-interruption code of 0007 hex (or 0087, 0207, or 0287 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Decimal-Divide Exception

A decimal-divide exception is recognized when either of the following is true:

- For DIVIDE DECIMAL when the divisor is zero or the quotient exceeds the specified data-field size. the sign codes of both the divisor and dividend are valid, and the digit or digits used in establishing the exception are valid.
- For VECTOR DIVIDE DECIMAL, VECTOR REMAINDER DECIMAL, and VECTOR SHIFT AND DIVIDE DECIMAL, when the divisor is zero and the divisor sign code used is valid. The divisor sign code used is the third operand sign code when the force operand 3 positive (P3) bit is zero, and is a positive sign code when the force operand 3 positive (P3) bit is one.

The operation is suppressed.

The instruction-length code is 2 or 3.

The decimal-divide exception is indicated by a program-interruption code of 000B hex (or 008B, 020B, or 028B hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Decimal-Overflow Exception

A decimal-overflow exception is recognized when the designated destination of a decimal operation is too short to contain all nonzero digits of the result.

The interruption may be disallowed by the decimal-overflow mask (PSW bit 21).

The operation is completed. When the destination is a storage operand, the overflow digits are ignored, and condition code 3 is set. When the destination is a vector register, the overflow digits are replaced with zeros, and if the condition code set (CS) flag is one, condition code 3 is set.

The instruction-length code is 2 or 3.

The decimal-overflow exception is indicated by a program-interruption code of 000A hex (or 008A, 020A, or 028A hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Execute Exception

The execute exception is recognized when the target instruction of an execute-type instruction (EXECUTE or EXECUTE RELATIVE LONG) is one of the following instructions:

- Another execute-type instruction
- TRANSACTION ABORT
- TRANSACTION BEGIN
- TRANSACTION END

The operation is suppressed.

The instruction-length code is 2 or 3 and indicates the length of the execute-type instruction that attempted to execute the target instruction.

The execute exception is indicated by a program-interruption code of 0003 hex (or 0083, 0203, or 0283 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

EX-Translation Exception

An EX-translation exception is recognized during PC-number translation in PROGRAM CALL when the entry-table entry designated by the entry-index part of the PC number is beyond the end of the entry table

as designated by the linkage-table or linkage-second-table entry used.

When ASN-and-LX reuse is not installed or is not enabled by a one value of the ASN-and-LX-reuse control in control register 0, or if it is installed and enabled and bit 44 of the second-operand address used by PROGRAM CALL is zero, bits 44-63 of the address (a 20-bit PC number), with 12 zeros appended on the left, are stored in the word at real location 172. When ASN-and-LX reuse is installed and enabled and bit 44 of the second-operand address is one, bits 32-63 of the address (a 32-bit PC) are stored in the word at real location 172.

The operation is nullified.

The instruction-length code is 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The EX-translation exception is indicated by a program-interruption code of 0023 hex (or 00A3 hex if a concurrent PER event is indicated).

Extended-Authority Exception

An extended-authority exception is recognized during access-register translation when all of the following are true:

1. The private bit in the access-list entry used is one.
2. The access-list-entry authorization index (ALEAX) in the access-list entry is not equal to the extended authorization index (EAX) in control register 8.
3. Either of the following is true:
   a. The authority-table entry designated by the EAX is beyond the length of the authority table used. The authority table is the one designated by the ASN-second-table entry that is designated by the access-list entry used.
   b. The secondary-authority bit designated by the EAX is zero.

The access-list entry is the one designated by the access register used.

The number of the access register is stored in bit positions 4-7 at real location 160, and bits 0-3 are set to zeros.

The operation is nullified.

The instruction-length code is 1, 2, or 3.

The extended-authority exception is indicated by a program-interruption code of 002D hex (or 00AD, 022D, or 02AD hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Fixed-Point-Divide Exception

A fixed-point-divide exception is recognized when any of the following is true:

1. In signed or unsigned binary division when the result is defined to be 32 bits, the divisor is zero, or the quotient cannot be expressed as a 32-bit signed or unsigned, respectively, binary integer.
2. In signed or unsigned binary division when the result is defined to be 64 bits, the divisor is zero, or the quotient cannot be expressed as a 64-bit signed or unsigned, respectively, binary integer.
3. The result of CONVERT TO BINARY cannot be expressed as a 32-bit signed binary integer for a 32-bit result or as a 64-bit signed binary integer for a 64-bit result.

In the case of division, the operation is suppressed. The execution of CONVERT TO BINARY (CVB, CVBY) is completed by ignoring the leftmost bits that cannot be placed in the register. The execution of CONVERT TO BINARY (CVBG) is suppressed.

The instruction-length code is 1, 2, or 3.

The fixed-point-divide exception is indicated by a program-interruption code of 0009 hex (or 0089, 0209, or 0289 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Fixed-Point-Overflow Exception

A fixed-point-overflow exception is recognized when an overflow occurs during signed binary arithmetic, signed left-shift operations, or some decimal to binary conversions,

The interruption may be disallowed by the fixed-point-overflow mask (PSW bit 20).

The operation is completed. The result is obtained by ignoring the overflow information, and condition code 3 is set, except for VECTOR CONVERT TO BINARY instructions where condition code 3 is set only when the condition code set (CS) flag is one.

The instruction-length code is 1, 2, or 3.

The fixed-point-overflow exception is indicated by a program-interruption code of 0008 hex (or 0088, 0208, or 0288 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

HFP-Divide Exception

An HFP-divide exception is recognized when in HFP division the divisor has a zero fraction.

The operation is suppressed.

The instruction-length code is 1 or 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The HFP-divide exception is indicated by a program-interruption code of 000F hex (or 008F, 020F, or 028F hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

HFP-Exponent-Overflow Exception

An HFP-exponent-overflow exception is recognized when the result characteristic of an HFP operation exceeds 127 and the result fraction is not zero.

The operation is completed. The fraction is normalized, and the sign and fraction of the result remain correct. The result characteristic is made 128 smaller than the correct characteristic.

The instruction-length code is 1, 2, or 3.

The HFP-exponent-overflow exception is indicated by a program-interruption code of 000C hex (or 008C, 020C, or 028C hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

HFP-Exponent-Underflow Exception

An HFP-exponent-underflow exception is recognized when the result characteristic of an HFP operation is less than zero and the result fraction is not zero. For an extended-format HFP result, HFP-exponent underflow is indicated only when the high-order characteristic underflows.

The interruption may be disallowed by the HFP-exponent-underflow mask (PSW bit 22).

The operation is completed. The HFP-exponent-underflow mask also affects the result of the operation. When the mask bit is zero, the sign, characteristic, and fraction are set to zero, making the result a true zero. When the mask bit is one, the fraction is normalized, the characteristic is made 128 larger than the correct characteristic, and the sign and fraction remain correct.

The instruction-length code is 1, 2, or 3.

The HFP-exponent-underflow exception is indicated by a program-interruption code of 000D hex (or 008D, 020D, or 028D hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

HFP-Significance Exception

An HFP-significance exception is recognized when the result fraction in HFP addition or subtraction is zero.

The interruption may be disallowed by the HFP-significance mask (PSW bit 23).

The operation is completed. The HFP-significance mask also affects the result of the operation. When the mask bit is zero, the operation is completed by replacing the result with a true zero. When the mask bit is one, the operation is completed without further change to the characteristic of the result.

The instruction-length code is 1 or 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The HFP-significance exception is indicated by a program-interruption code of 000E hex (or 008E, 020E, or 028E hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

HFP-Square-Root Exception

An HFP-square-root exception is recognized when the second operand of an HFP SQUARE ROOT instruction is less than zero.

The operation is suppressed.

The instruction-length code is 2 or 3.

The HFP-square-root exception is indicated by a program-interruption code of 001D hex (or 009D, 021D, or 029D hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

LFX-Translation Exception

An LFX-translation exception may be recognized only when the ASN-and-LX-reuse facility is installed and the ASN-and-LX-reuse control in control register 0 is one.

An LFX-translation exception is recognized during PC-number translation in PROGRAM CALL when either:

1. The linkage-first-table entry specified by the linkage-first-index part of the PC number is beyond the end of the linkage first table as designated by the linkage-first-table designation used.
2. Bit 0 of the linkage-first-table entry is not zero.

When bit 44 of the second-operand address used by PROGRAM CALL is zero, bits 44-63 of the address (a 20-bit PC number), with 12 zeros appended on the left, are stored in the word at real location 172. When bit 44 of the second-operand address is one, bits 32-63 of the address (a 32-bit PC number) are stored in the word at real location 172.

The operation is nullified.

The instruction-length code is 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The LFX-translation exception is indicated by a program-interruption code of 0026 hex (or 00A6 hex if a concurrent PER event is indicated).

LSTE-Sequence Exception

An LSTE-sequence exception may be recognized only when the ASN-and-LX-reuse facility is installed and the ASN-and-LX-reuse control in control register 0 is one.

An LSTE-sequence exception is recognized during PC-number translation in PROGRAM CALL when the linkage-second-table-entry sequence number (LSTESN) in the linkage-second-table entry used is nonzero and not equal to the LSTESN in bit positions 0-31 of general register 15.

When bit 44 of the second-operand address used by PROGRAM CALL is zero, bits 44-63 of the address (a 20-bit PC number), with 12 zeros appended on the left, are stored in the word at real location 172. When bit 44 of the second-operand address is one, bits 32-63 of the address (a 32-bit PC number) are stored in the word at real location 172.

The operation is nullified.

The instruction-length code is 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The LSTE-sequence exception is indicated by a program-interruption code of 002E hex (or 00AE hex if a concurrent PER event is indicated).

LSX-Translation Exception

An LSX-translation exception may be recognized only when the ASN-and-LX-reuse facility is installed and the ASN-and-LX-reuse control in control register 0 is one.

An LSX-translation exception is recognized during PC-number translation in PROGRAM CALL when bit 0 of the linkage-second-table entry specified by the linkage-second-index part of the PC number is not zero.

When bit 44 of the second-operand address used by PROGRAM CALL is zero, bits 44-63 of the address (a 20-bit PC number), with 12 zeros appended on the left, are stored in the word at real location 172. When bit 44 of the second-operand address is one, bits 32-63 of the address (a 32-bit PC number) are stored in the word at real location 172.

The operation is nullified.

The instruction-length code is 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The LSX-translation exception is indicated by a program-interruption code of 0027 hex (or 00A7 hex if a concurrent PER event is indicated).

LX-Translation Exception

An LX-translation exception may be recognized only when the ASN-and-LX-reuse facility is not installed or the ASN-and-LX-reuse control in control register 0 is zero.

An LX-translation exception is recognized during PC-number translation in PROGRAM CALL when either:

1. The linkage-table entry designated by the linkage-index part of the PC number is beyond the end of the linkage table as designated by the linkage-table designation used.
2. Bit 0 of the linkage-table entry is not zero.

Bits 44-63 of the second-operand address used by PROGRAM CALL (a 20-bit PC number), with 12 zeros appended on the left, are stored in the word at real location 172.

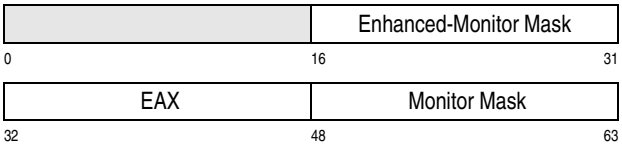
The operation is nullified.

The instruction-length code is 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The LX-translation exception is indicated by a program-interruption code of 0022 hex (or 00A2 hex if a concurrent PER event is indicated).

Monitor Event

A monitor event is recognized when MONITOR CALL is executed and the monitor-mask bit in control register 8 corresponding to the monitor class specified by instruction bits 12-15 is one. The information in control register 8 has the following format:



The monitor-mask bits, bits 48-63 of control register 8, correspond to monitor classes 0-15, respectively. Any number of monitor-mask bits may be on at a time; together they specify the classes of monitor events that are monitored at that time. The mask bits are initialized to zeros.

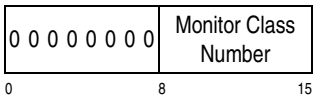
When the enhanced-monitor facility is installed, the enhanced-monitor-mask bits, bits 16-31 of control register 8, correspond to the monitor classes 0-15, respectively. Any number of enhanced-monitor-mask bits may be on at a time; together they specify the classes of monitor events that result in monitor-call program interruptions versus monitor-event counting operations.

When a monitor event occurs and either (a) the enhanced-monitor facility is not installed, or (b) the facility is installed but the enhanced-monitor-mask bit corresponding to the monitor class specified in bits 12-15 of the MONITOR CALL instruction is zero, then a monitor-event program interruption occurs. When a monitor event occurs, the enhanced-monitor facility is installed, and the enhanced-monitor-mask bit corresponding to the monitor class is one, a monitor-event-counting operation is performed, as described in “Monitor-Event Counting” on page 5-109. Note, in the ESA/390-compatibility mode, the enhanced-monitor masks are always zero; thus, a monitor-event counting operation never occurs.

When a monitor-event program interruption occurs, additional information is stored at real locations in the prefix area, as follows:

The contents of bit positions 8-15 of the MONITOR CALL instruction are stored at real location 149 and constitute the monitor-class number. Zeros are stored at real location 148. The contents of real locations 148-149 are as follows:

Real Locations 148-149



The effective address specified by the B_1 and D_1 fields of the instruction forms the monitor code. In the z/Architecture architectural mode, the monitor code is stored in the doubleword at real location 176, as shown in Figure 6-5. In the ESA/390-compatibility mode, bits 32-63 of the monitor code are stored in the word at real location 156, as shown in Figure 6-6. The value of the address is under control of the addressing mode, bits 31 and 32 of the current PSW. In the 24-bit addressing mode, bits 0-39 of the

address are zeros, while in the 31-bit addressing mode, bits 0-32 are zeros.

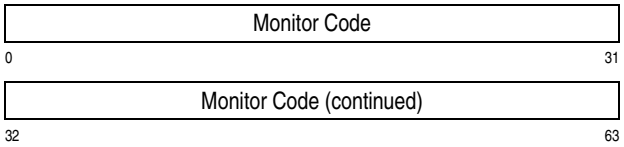


Figure 6-5. Monitor Code in Real Locations 176-183 in the z/Architecture Architectural Mode

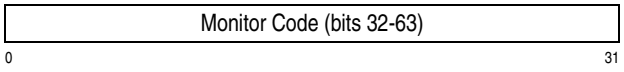


Figure 6-6. Monitor Code in Real Locations 156-159 in the ESA/390-compatibility Mode

The operation is completed.

The instruction-length code is 2, except that when the exception occurs during the execution of a MONITOR CALL instruction that is the target of EXECUTE RELATIVE LONG, the ILC is 3.

The monitor event is indicated by a program-interruption code of 0040 hex (or 00C0 hex if a concurrent PER event is indicated).

Operand Exception

An operand exception is recognized when any of the following is true:

1. Execution of CLEAR SUBCHANNEL, HALT SUBCHANNEL, MODIFY SUBCHANNEL, RESUME SUBCHANNEL, START SUBCHANNEL, STORE SUBCHANNEL, or TEST SUBCHANNEL is attempted and bit positions 32-47 of general register 1 do not contain 0001 hex. However, an exception due to ones in bit positions 32-39 of the register may or may not be recognized.
2. Execution of MODIFY SUBCHANNEL is attempted, and bits 1 and 6 of word 1 of the SCHIB operand are not zeros or bits 9-10 and 25-30 of word 6 of the SCHIB operand are not all zeros.
3. Execution of MODIFY SUBCHANNEL is attempted, and bits 9 and 10 of word 1 of the SCHIB operand are both one.
4. Execution of RESET CHANNEL PATH is attempted, and bits 40-55 of general register 1 are not all zeros.
5. Execution of SET ADDRESS LIMIT is attempted, and bits 32 and 48-63 of general register 1 are not all zeros.
6. Execution of SET CHANNEL MONITOR is attempted, bit 62 of general register 1 is one, and bits 59-63 of general register 2 are not all zeros.
7. Execution of SET CHANNEL MONITOR is attempted, and bits 36-61 of general register 1 are not all zeros.
8. Execution of START SUBCHANNEL is attempted, and bits 5, 13, and 25-28 of word 1 of the ORB operand are not all zeros.
9. Execution of START SUBCHANNEL is attempted, and bit 11 of word 1 of the ORB operand is not zero. This exception may or may not be recognized.

The operation is suppressed.

The instruction-length code is 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The operand exception is indicated by a program-interruption code of 0015 hex (or 0095 hex if a concurrent PER event is indicated).

Operation Exception

An operation exception is recognized when the CPU attempts to execute an instruction with an invalid operation code. The operation code may be unassigned, or the instruction with that operation code may not be installed on the CPU.

The operation is suppressed.

The instruction-length code is 1, 2, or 3.

The operation exception is indicated by a program-interruption code of 0001 hex (or 0081, 0201, or 0281 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

An operation exception detected while the CPU is in the nonconstrained transactional-execution mode may result in the transaction being aborted due to a restricted instruction (abort code 11). An operation exception detected while the CPU is in the constrained transactional-execution mode may be indi-

cated as a transaction-constraint exception (program-interruption code 0018 hex).

Programming Notes:

1. Some models may offer instructions not described in this publication, such as those provided for assists or as part of special or custom features. Consequently, operation codes not described in this publication do not necessarily cause an operation exception to be recognized. Furthermore, these instructions may cause modes of operation to be set up or may otherwise alter the machine so as to affect the execution of subsequent instructions. To avoid causing such an operation, an instruction with an operation code not described in this publication should be executed only when the specific function associated with the operation code is desired.

2. Operation code 00 hex will never be assigned to an instruction implemented in the CPU.

Page-Translation Exception

A page-translation exception is recognized when the page-invalid bit is one.

The exception is recognized as part of the execution of an instruction that needs the page-table entry in the translation of an instruction, operand, or side-effect address, except for the operand address in LOAD REAL ADDRESS and TEST PROTECTION, in which case the condition is indicated by the setting of the condition code, and except for an operand address in MOVE PAGE, in which case the condition is indicated by the setting of the condition code if the condition-code-option bit, bit 55 of general register 0, is one.

When an interruption occurs, information about the virtual address causing the exception is stored at real locations 168-175 and conditionally at real locations 160 and 162. See “Assigned Storage Locations” on page 3-73 for a detailed description of this information.

The unit of operation is nullified.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during a reference to the target of EXECUTE, the ILC is 2. When the exception occurs during a reference to the target of EXECUTE RELATIVE LONG, the ILC is 3.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The page-translation exception is indicated by a program-interruption code of 0011 hex (or 0091, 0211, or 0291 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

PC-Translation-Specification Exception

A PC-translation-specification exception is recognized during PC-number translation in PROGRAM CALL when either of the following is true for the entry-table entry (ETE) used:

1. The PROGRAM CALL operation is the basic operation (bit 128 of the ETE is zero) in the 24-bit or 31-bit addressing mode (bit 31 of the PSW is zero), bit 32 of the ETE is zero (specifying the 24-bit mode), and bits 33-39 of the ETE are not all zeros.

2. The PROGRAM CALL operation is the stacking operation (bit 128 of the ETE is one), bits 32 and 129 of the ETE are zeros (specifying the 24-bit mode), and bits 33-39 of the ETE are not all zeros.

The operation is suppressed.

The instruction-length code is 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The PC-translation-specification exception is indicated by a program-interruption code of 001F hex (or 009F hex if a concurrent PER event is indicated).

PER Event

A PER event is recognized when the CPU is enabled for PER and one or more of these events occur.

The PER mask, bit 1 of the PSW, controls whether the CPU is enabled for PER. When the PER mask is zero, PER events are not recognized. When the bit is one, PER events are recognized, subject to the PER-event-mask bits in control register 9.

For a PER instruction-fetching nullification event, the unit of operation is nullified. For other PER events, the unit of operation is completed, unless another

condition has caused the unit of operation to be nullified, suppressed, or terminated.

In the z/Architecture architectural mode, information identifying the event is stored at real locations 150-159 and conditionally at real location 161. In the ESA/390-compatibility mode, information identifying the event is stored at real locations 150-155 and conditionally at real location 161.

The instruction-length code is 0, 1, 2, or 3. Code 0 is set for the PER instruction-fetching basic event only if a specification exception is indicated concurrently. Code 0 is always set when the PER instruction-fetching nullification event is indicated.

The PER event is indicated by setting bit 8 of the program-interruption code to one.

See "Program-Event Recording" on page 4-26 for a detailed description of the PER event and the associated interruption information.

Primary-Authority Exception

A primary-authority exception is recognized during ASN authorization in PROGRAM TRANSFER with space switching (PT-ss) or PROGRAM TRANSFER WITH INSTANCE with space switching (PTI-ss) when either:

1. The authority-table entry indicated by the authorization index in control register 4 is beyond the end of the authority table used. The authority table is the one designated by the ASN-second-table entry for the ASN used.
2. The primary-authority bit indicated by the authorization index is zero.

The ASN used is stored at real locations 174-175, and real locations 172-173 are set to zeros.

The operation is nullified.

The instruction-length code is 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The primary-authority exception is indicated by a program-interruption code of 0024 hex (or 00A4 hex if a concurrent PER event is indicated).

Privileged-Operation Exception

A privileged-operation exception is recognized when any of the following is true:

1. Execution of a privileged instruction is attempted in the problem state.
2. The value of the rightmost bit of the general register designated by the R_2 field of the PROGRAM TRANSFER or PROGRAM TRANSFER WITH INSTANCE instruction is zero and would cause the PSW problem-state bit to change from the problem state (one) to the supervisor state (zero).
3. In the problem state, the key value specified by the second operand of the SET PSW KEY FROM ADDRESS instruction corresponds to a zero PSW-key-mask bit in control register 3.
4. In the problem state, the key value specified by the rightmost byte of the register designated by the R_3 field of the MOVE WITH KEY instruction corresponds to a zero PSW-key-mask bit in control register 3.
5. In the problem state, the key value specified by the rightmost byte of the register designated by the R_3 field for the instruction MOVE TO PRIMARY, MOVE TO SECONDARY, or MOVE WITH KEY corresponds to a zero PSW-key-mask bit in control register 3.
6. In the problem state, any of the following instructions is encountered, and the extraction-authority control, bit 4 of control register 0, is zero.
   - EXTRACT PRIMARY ASN
   - EXTRACT SECONDARY ASN
   - INSERT ADDRESS SPACE CONTROL
   - INSERT PSW KEY
   - INSERT VIRTUAL STORAGE KEY
7. In the problem state, the result of ANDing the authorization key mask (AKM) with the PSW-key mask in control register 3 during PROGRAM CALL produces a result of zero.
8. In the problem state, bits 20-23 of the second-operand address of the SET ADDRESS SPACE CONTROL or SET ADDRESS SPACE CONTROL FAST instruction have the value 0011 binary.
9. In the problem state, the key value specified by the rightmost byte of general register 1 for the instruction MOVE WITH SOURCE KEY or MOVE

WITH DESTINATION KEY corresponds to a zero PSW-key-mask bit in control register 3.

10. In the problem state, the key value specified by the rightmost byte of the register designated by the R_1 field for the instruction BRANCH AND SET AUTHORITY corresponds to a zero PSW-key-mask bit in control register 3.

11. In the problem state, bits 16 and 17 of the PSW field in the second operand of RESUME PROGRAM have the value 11 binary.

12. Execution of MOVE WITH OPTIONAL SPECIFICATIONS is attempted in the problem state, and any of the following is true:

    - The specified-access-key control for the first operand, bit 46 of general register 0, is one; and the corresponding specified-access key, bits 32-35 of the register, designates a PSW-key-mask (PKM) bit position in control register 3 that contains zero.

    - The specified-access-key control for the second operand, bit 62 of general register 0, is one; and the corresponding specified-access key, bits 48-51 of the register, designates a PKM bit position that contains zero.

    - Either or both bits 46 and 62 of general register 0 are zero, and the PSW key, bits 8-11 of the PSW, designates a PKM bit position that contains zero.

The operation is suppressed.

The instruction-length code is 1, 2, or 3.

The privileged-operation exception is indicated by a program-interruption code of 0002 hex (or 0082, 0202, or 0282 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Protection Exception

A protection exception is recognized when any of the following is true:

Access-List-Controlled Protection: The CPU attempts to store, in the access-register mode, by means of an access-list entry which has the fetch-only bit set to one.

DAT Protection: The CPU attempts to store, with DAT on, into any of the following:

- A page which has the DAT-protection bit set to one in the page-table entry used in the translation
- A segment which has the DAT-protection bit set to one in the segment-table entry used in the translation
- When EDAT-1 applies,[1] a region which has the DAT-protection bit set to one in any region-table entry used in the translation.

Instruction-Execution Protection: The instruction-execution-protection facility is installed and enabled, and the CPU attempts to fetch an instruction, with DAT on, from any of the following:

- A page which has the instruction-execution-protection control set to one in the page-table entry used in the translation.
- When EDAT-1 applies, a segment which has the instruction-execution-protection control set to one in the format-1 segment-table entry used in the translation.
- When EDAT-2 applies, a region which has the instruction-execution-protection control set to one in the format-1 region-third-table entry used in the translation.

Key-Controlled Protection: The CPU attempts to access a storage location that is protected against the type of reference, and the access key does not match the storage key.

Low-Address Protection: The CPU attempts a store that is subject to low-address protection, the effective address is in the range 0-511 or 4096-4607, and the low-address protection control, bit 35 of control register 0, is one.

The operation is suppressed when the location of the instruction is protected against fetching. Similarly, the operation is suppressed when the location of the target instruction of an execute-type instruction is protected against fetching.

For access-list-controlled protection, DAT-protection, and instruction-execution protection, the operation is suppressed. For key-controlled protection and low-

1. See "Enhanced-DAT Terminology:" on page 3-41 for an explanation of the term "enhanced-DAT applies."

address protection, the operation may be either suppressed or terminated, depending on the type of suppression-on-protection facility that is installed. See "Suppression on Protection" on page 3-15 and Figure 6-4 on page 6-21 for details.

For termination, changes may occur only to result fields. In this context, the term "result field" includes condition code, registers, and storage locations, if any, which are due to be changed by the instruction. However, no change is made to a storage location when a reference to that location causes a protection exception. Therefore, if an instruction is due to change only the contents of a field in storage, and every byte of that field would cause a protection exception, the operation is suppressed. When termination occurs on fetching, the protected information is not loaded into an addressable register nor moved to another storage location.

In the z/Architecture architectural mode, information about the address causing the exception is stored at real locations 168-175 and conditionally at real location 160. In the ESA/390-compatibility mode, information about the address causing the exception may be stored at real locations 144-147 and 160. See "Suppression on Protection" on page 3-15.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2. When the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

For a protected operand location, the instruction-length code (ILC) is 1, 2, or 3, indicating the length of the instruction that caused the reference, except that when the protected operand location is accessed by an instruction that is the target of an EXECUTE or EXECUTE RELATIVE LONG instruction, the ILC is 2 or 3, respectively.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2. When the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The protection exception is indicated by a program-interruption code of 0004 hex (or 0084, 0204, or 0284 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Region-First-Translation Exception

A region-first-translation exception is recognized when a region first table is in the translation path for translation of a virtual address and either:

1. The region-first-table entry indicated by the region-first-index portion of the virtual address is outside the region first table.
2. The region-invalid bit is one.

The exception is sometimes called simply a region-translation exception, which term applies also to a region-second-translation exception and a region-third-translation exception.

The exception is recognized as part of the execution of an instruction that needs the region-first-table entry in the translation of an instruction or operand address, except for the operand address in LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS and TEST PROTECTION, in which case the condition is indicated by the setting of the condition code.

When an interruption occurs, information about the virtual address causing the exception is stored at real locations 168-175 and conditionally at real locations 160 and 162. See "Assigned Storage Locations" on page 3-73 for a detailed description of this information.

The unit of operation is nullified.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2. When the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The region-first-translation exception is indicated by a program-interruption code of 0039 hex (or 00B9, 0239, or 02B9 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Region-Second-Translation Exception

A region-second-translation exception is recognized when a region second table is in the translation path for translation of a virtual address and either:

1. The region-second-table entry indicated by the region-second-index portion of the virtual address is outside the region second table.
2. The region-invalid bit is one.

The exception is sometimes called simply a region-translation exception, which term applies also to a region-first-translation exception and a region-third-translation exception.

The exception is recognized as part of the execution of an instruction that needs the region-second-table entry in the translation of an instruction or operand address, except for the operand address in LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS and TEST PROTECTION, in which case the condition is indicated by the setting of the condition code.

When an interruption occurs, information about the virtual address causing the exception is stored at real locations 168-175 and conditionally at real locations 160 and 162. See “Assigned Storage Locations” on page 3-73 for a detailed description of this information.

The unit of operation is nullified.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2. When the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The region-second-translation exception is indicated by a program-interruption code of 003A hex (or 00BA, 023A, or 02BA hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Region-Third-Translation Exception

A region-third-translation exception is recognized when a region third table is in the translation path for translation of a virtual address and either:

1. The region-third-table entry indicated by the region-third-index portion of the virtual address is outside the region third table.
2. The region-invalid bit is one.

The exception is sometimes called simply a region-translation exception, which term applies also to a region-first-translation exception and a region-second-translation exception.

The exception is recognized as part of the execution of an instruction that needs the region-third-table entry in the translation of an instruction or operand address, except for the operand address in LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS and TEST PROTECTION, in which case the condition is indicated by the setting of the condition code.

When an interruption occurs, information about the virtual address causing the exception is stored at real locations 168-175 and conditionally at real locations 160 and 162. See “Assigned Storage Locations” on page 3-73 for a detailed description of this information.

The unit of operation is nullified.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2. When the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The region-third-translation exception is indicated by a program-interruption code of 003B hex (or 00BB, 023B, or 02BB hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Secondary-Authority Exception

A secondary-authority exception is recognized during ASN authorization in SET SECONDARY ASN with space switching, SET SECONDARY ASN WITH INSTANCE with space switching, or during ASN authorization in PROGRAM RETURN when the restored SASN does not equal the restored PASN, when either:

1. The authority-table entry indicated by the authorization index in control register 4 is beyond the end of the authority table used. The authority table is the one designated by the ASN-second-table entry for the ASN used. For PROGRAM RETURN, the ASN is the SASN being restored from the linkage-stack state entry used.
2. The secondary-authority bit indicated by the authorization index is zero.

The ASN used is stored at real locations 174-175, and real locations 172-173 are set to zeros.

The operation is nullified.

The instruction-length code is 1 or 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The secondary-authority exception is indicated by a program-interruption code of 0025 hex (or 00A5 hex if a concurrent PER event is indicated).

Segment-Translation Exception

A segment-translation exception is recognized when either:

1. The segment-table entry indicated by the segment-index portion of a virtual address is outside the segment table.
2. The segment-invalid bit is one.

The exception is recognized as part of the execution of an instruction that needs the segment-table entry in the translation of an instruction or operand address, except for the operand address in LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS and TEST PROTECTION, in which case the condition is indicated by the setting of the condition code.

When an interruption occurs, information about the virtual address causing the exception is stored at real locations 168-175 and conditionally at real locations 160 and 162. See “Assigned Storage Locations” on page 3-73 for a detailed description of this information.

The unit of operation is nullified.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2. When the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The segment-translation exception is indicated by a program-interruption code of 0010 hex (or 0090, 0210, or 0290 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Space-Switch Event

A space-switch event is recognized at the completion of the operation in each of the following cases:

1. The space-switching form of PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, or PROGRAM TRANSFER WITH INSTANCE is executed and any of the following is true:
   a. The primary space-switch-event-control bit, bit 57 of control register 1, is one before the operation.
   b. The primary space-switch-event-control bit is one after the operation.
   c. A PER event is indicated.
2. RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST is executed, the CPU is in the home-space mode either before or after the operation, but not both before and after the operation, and any of the following is true:
   a. The primary space-switch-event-control bit, bit 57 of control register 1, is one.
   b. The home space-switch-event-control bit, bit 57 of control register 13, is one.
   c. A PER event is indicated.

For PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, and PROGRAM TRANSFER WITH INSTANCE, and for a RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST instruction that changes the translation mode to the home-space mode, the old PASN, which is in bit positions 48-63 of control register 4 before the operation, is stored at real locations 174-175, and the old primary space-switch-event-control bit is placed in bit position 0 and zeros are placed in bit positions 1-15 at real locations 172-173.

For a RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST instruction that changes the translation mode away from the home-space mode, zeros are stored at real locations 174-175, and the home space-switch-event-control bit is placed in bit position 0 and zeros are placed in bit positions 1-15 at real locations 172-173.

For a PROGRAM RETURN instruction that introduces a PSW-format error, it is unpredictable whether the instruction-length code is 0 or 1, or 0 or 2 if EXECUTE was used, or 0 or 3 if EXECUTE RELATIVE LONG was used.

The operation is completed.

The instruction-length code is 0, 1, or 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 0 or 3.

The space-switch event is indicated by a program-interruption code of 001C hex (or 009C hex if a concurrent PER event is indicated).

Programming Notes:

1. The space-switch event permits the control program to gain control whenever a program enters or leaves a particular address space. The primary space-switch-event-control bit is loaded into control register 1, along with the remaining bits of the primary address-space-control element, whenever control register 1 is loaded.

2. The space-switch event may be useful in obtaining programmed authorization checking, in causing additional trace information to be recorded, or in enabling or disabling the CPU for PER or tracing.

3. Bit 121 of the ASN-second-table entry (ASTE) is loaded into bit position 57 of control register 1 as part of the PC-ss, PR-ss, PT-ss, and PTI-ss operations. If bit 121 of the ASTE for a particular address space is set to one, then a space-switch event is recognized when a program enters or leaves the address space by means of any of PC-ss, PR-ss, PT-ss, or PTI-ss.

4. The occurrence of a space-switch event at the completion of a PC-ss, PR-ss, PT-ss, or PTI-ss operation when any PER event is indicated, or at the completion of execution of a RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST instruction that changes to or from the home-space mode when any PER event is indicated, permits the control program to determine the address space from which the instruction causing the PER event was fetched.

Special-Operation Exception

A special-operation exception is recognized when any of the following is true:

1. Execution of any of the following instructions is attempted with DAT off:

   - EXTRACT PRIMARY ASN
   - EXTRACT SECONDARY ASN
   - INSERT ADDRESS SPACE CONTROL
   - INSERT VIRTUAL STORAGE KEY
   - MOVE WITH OPTIONAL SPECIFICATIONS
   - SET ADDRESS SPACE CONTROL
   - SET SECONDARY ASN

2. Execution of BRANCH AND SET AUTHORITY is attempted, and the R_2 field is zero in the base-authority state or nonzero in the reduced-authority state.

3. Execution of BRANCH AND STACK, stacking PROGRAM CALL, PROGRAM RETURN, or TRAP is attempted, and the CPU is not in the primary-space or access-register mode.

4. Execution of BRANCH IN SUBSPACE GROUP is attempted, and any of the following is true:

   - The current primary address space is not in a subspace group associated with the current dispatchable unit, that is, the primary-ASTE origin (PASTEO) in control register 5 does not equal the base-ASTE origin (BASTEO) in the dispatchable-unit control table (DUCT).

- The access-list-entry token (ALET) in access register R_2 is ALET 1, but a subspace has not previously been entered by the dispatchable unit by means of BRANCH IN SUBSPACE GROUP, that is, the subspace-ASTE origin (SSASTEO) in the DUCT is all zeros.
- The ALET used is other than ALET 0 and ALET 1, and the destination ASTE (DASTE) does not specify the base space or a subspace of the subspace group, that is, the DASTE origin (DASTEO) obtained from an access-list entry does not equal the BASTEO in the DUCT, and either the subspace-group bit (G) in the address-space-control element in the DASTE is zero or the base-space bit (B) in the DASTE is one.

5. Execution of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE or MODIFY STACKED STATE is attempted, and the CPU is not in the primary-space, access-register, or home-space mode.
6. Execution of EXTRACT TRANSACTION NESTING DEPTH, TRANSACTION BEGIN, or TRANSACTION END is attempted, and the transactional-execution control, bit 8 of control register 0, is zero.
7. Execution of LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL with space switching (PC-ss), PROGRAM TRANSFER with space switching (PT-ss), PROGRAM TRANSFER WITH INSTANCE with space switching (PT-ss), SET SECONDARY ASN, or SET SECONDARY ASN WITH INSTANCE is attempted, or execution of a PROGRAM RETURN instruction requiring PASN or SASN translation is attempted, and the ASN-translation control, bit 44 of control register 14, is zero.
8. Execution of LOAD GUARDED STORAGE CONTROLS or STORE GUARDED STORAGE CONTROLS is attempted, and the guarded-storage-facility-enablement (GSFE) control, bit 59 of control register 2, is zero.
9. Execution of LOAD PAGE-TABLE-ENTRY ADDRESS is attempted and the effective address-space-control element specified by the M_4 field of the instruction designates the real space.
10. Execution of LOAD REAL ADDRESS (LRA) is attempted in the 24-bit or 31-bit addressing mode, and bits 0-32 of the resulting real or absolute address are not all zeros.
11. Execution of MOVE TO PRIMARY or MOVE TO SECONDARY is attempted, and the CPU is not in the primary-space or secondary-space mode.
12. Execution of MOVE TO PRIMARY, MOVE TO SECONDARY, or SET ADDRESS SPACE CONTROL is attempted, and the secondary-space control, bit 37 of control register 0, is zero. The exception may be recognized for this reason when execution of SET ADDRESS SPACE CONTROL FAST is attempted.
13. Execution of MOVE WITH OPTIONAL SPECIFICATIONS is attempted, and either of the following conditions are true:
    - Bit 47 of general register 0 is one, bits 40-41 of the register contain 11 binary, and the problem-state bit, bit 15 of the current PSW, is one.
    - The secondary-space control, bit 37 of control register 0 is zero, and any of the following conditions are true:
      – Bit 47 of general register 0 is zero, and bits 16-17 of the current PSW are 10 binary.
      – Bit 47 of general register 0 is one, and bits 40-41 of the register are 10 binary.
      – Bit 63 of general register 0 is zero, and bits 16-17 of the current PSW are 10 binary.
      – Bit 63 of general register 0 is one, and bits 56-57 of the register are 10 binary.
14. Execution of basic PROGRAM CALL, PROGRAM TRANSFER, or PROGRAM TRANSFER WITH INSTANCE is attempted, and the CPU is not in the primary-space mode.
15. Execution of basic PROGRAM CALL is attempted, and the extended-addressing-mode bit, bit 31 of the current PSW, does not equal the entry-extended-addressing-mode bit, bit 129, in the entry-table entry.
16. Execution of PROGRAM CALL, PROGRAM TRANSFER, or PROGRAM TRANSFER WITH

INSTANCE is attempted, and the subsystem-linkage control, bit 192 of the primary ASN-second-table entry, is zero.

17. Execution of the space-switching form of PROGRAM TRANSFER or SET SECONDARY ASN is attempted, the ASN-and-LX-reuse control in control register 0 is one, and the reusable-ASN bit in the located ASTE is one.

18. Execution of the space-switching form of PROGRAM TRANSFER WITH INSTANCE or SET SECONDARY ASN WITH INSTANCE is attempted, the controlled-ASN bit is one in the ASN-second-table entry used, and the CPU is in the problem state at the beginning of the operation.

19. Execution of SET SYSTEM MASK is attempted in the supervisor state, and the SSM-suppression control, bit 33 of control register 0, is one.

20. Execution of TRANSACTION ABORT is attempted, and the CPU is not in the transactional-execution mode at the beginning of the instruction.

21. Execution of TRAP is attempted, and either of the following is true:

    - The TRAP-enabled bit, bit 31 in bytes 44-47 of the dispatchable-unit control table, is zero.
    - The PSW control, bit 13 of bytes 0-3 of the trap control table, is zero and bit 31 of the current PSW, the extended-addressing-mode bit, is one.

The operation is suppressed.

The instruction-length code is 1, 2, or 3.

The special-operation exception is indicated by a program-interruption code of 0013 hex (or 0093, 0213, or 0293 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Specification Exception

A specification exception is recognized when any of the following is true:

1. A one is introduced into an unassigned bit position of the PSW (that is, any of bit positions 0, 2-4, 25-30**,** or 33-63). This is handled as an early PSW specification exception.

2. A one is introduced into bit position 12 of the PSW. This is handled as an early PSW specification exception.

3. The PSW is invalid in any of the following ways:

    a. Bit 31 of the PSW is one and bit 32 is zero.

    b. Bits 31 and 32 of the PSW are zero, indicating the 24-bit addressing mode, and bits 64-103 of the PSW are not all zeros.

    c. Bit 31 of the PSW is zero and bit 32 is one, indicating the 31-bit addressing mode, and bits 64-96 of the PSW are not all zeros.

    d. In the ESA/390-compatibility mode, bit 5 of the PSW is one.

    e. In the ESA/390-compatibility mode, bit 31 of the PSW is one. It is unpredictable whether this condition is recognized.

    This is handled as an early PSW specification exception. Note, items b and c, above, do not apply to SAM24 and SAM31, respectively. See items 47 and 48 in this list for further explanation.

4. The PSW contains an odd instruction address.

5. An operand address does not designate an integral boundary in an instruction requiring such integral-boundary designation.

6. An odd-numbered general register is designated by an R field of an instruction that requires an even-numbered register designation.

7. A floating-point register other than 0, 1, 4, 5, 8, 9, 12, or 13 is designated for an extended operand.

8. The multiplier or divisor in decimal arithmetic exceeds 15 digits and sign.

9. The length of the first-operand field is less than or equal to the length of the second-operand field in decimal multiplication or division.

10. Execution of CIPHER MESSAGE, CIPHER MESSAGE WITH AUTHENTICATION, CIPHER MESSAGE WITH CIPHER FEEDBACK, CIPHER MESSAGE WITH CHAINING, CIPHER MESSAGE WITH COUNTER, CIPHER MESSASGE WITH OUTPUT FEEDBACK, COMPUTE INTERMEDIATE MESSAGE DIGEST, COMPUTE LAST MESSAGE DIGEST, COMPUTE MESSAGE AUTHENTICATION CODE, PERFORM CRYPTOGRAPHIC KEY MANAGE-

MENT OPERATION, PERFORM CRYPTOGRAPHIC COMPUTATION, or PERFORM RANDOM NUMBER OPERATION is attempted, and the function code in bits 57-63 of general register 0 contain an unassigned or uninstalled function code.

11. Execution of CIPHER MESSAGE, CIPHER MESSAGE WITH CIPHER FEEDBACK, CIPHER MESSAGE WITH CHAINING, CIPHER MESSASGE WITH OUTPUT FEEDBACK, or PERFORM RANDOM NUMBER OPERATION is attempted, and the R_1 or R_2 field designates an odd-numbered register or general register 0.

12. Execution of CIPHER MESSAGE, CIPHER MESSAGE WITH CHAINING, CIPHER MESSAGE WITH COUNTER, CIPHER MESSASGE WITH OUTPUT FEEDBACK, COMPUTE INTERMEDIATE MESSAGE DIGEST or COMPUTE MESSAGE AUTHENTICATION CODE is attempted, and the second operand length is not a multiple of the data block size of the designated function. This specification-exception condition does not apply to the query functions.

13. Execution of CIPHER MESSAGE WITH AUTHENTICATION or CIPHER MESSAGE WITH COUNTER is attempted, and the R_1, R_2, or R_3 field designates an odd-numbered register or general register 0.

14. Execution of CIPHER MESSAGE WITH AUTHENTICATION is attempted, the function code is nonzero, the LPC flag is one (indicating that the last blocks of plaintext or ciphertext are being processed), but the LAAD flag is zero.

15. Execution of CIPHER MESSAGE WITH CIPHER FEEDBACK is attempted, and any of the following applies:

    - The second operand length is not a multiple of the length of cipher feedback. This specification-exception condition does not apply to the query functions.
    - For DEA or TDEA functions, bits 32-39 of general register 0 specify a value that is zero or greater than 8.
    - For AES functions, bits 32-39 of general register 0 specify a value that is zero or greater than 16.

16. Execution of COMPARE AND FORM CODEWORD is attempted, and general registers 1, 2, and 3 do not initially contain even values.

17. Execution of COMPARE AND REPLACE DAT TABLE ENTRY is attempted and either of the following conditions exist:

    - The DTT field, bit positions 59-61 of general register R_2, contain 001, 010, or 011 binary.
    - Bit positions 52-63 of general register $R_2 + 1$ contain nonzero values.

18. Execution of COMPARE AND SWAP AND STORE is attempted and any of the following conditions exist:

    - The function code specifies an unassigned value.
    - The store characteristic specifies an unassigned value.
    - The function code is 0, and the first operand is not designated on a word boundary.
    - The function code is 1, and the first operand is not designated on a doubleword boundary.
    - The function code is 2, and any of the following is true:
        – The compare-and-swap-and-store facility 2 is not installed.
        – The first operand is not designated on a quadword boundary.
        – The R_3 field does not designate the even-numbered register of an even-odd register pair.
    - The second operand is not designated on an integral boundary corresponding to the size of the store value.

19. Execution of COMPARE LOGICAL LONG UNICODE or MOVE LONG UNICODE is attempted, and the contents of either general register $R_1 + 1$ or $R_3 + 1$ do not specify an even number of bytes.

20. Execution of COMPARE LOGICAL STRING, MOVE STRING or SEARCH STRING is

attempted, and bits 32-55 of general register 0 are not all zeros.

21. Execution of COMPRESSION CALL is attempted, and bits 48-51 of general register 0 have any of the values 0000 and 0110-1111 binary when bit 44 is 0.

22. Execution of COMPUTE INTERMEDIATE MESSAGE DIGEST, COMPUTE LAST MESSAGE DIGEST, or COMPUTE MESSAGE AUTHENTICATION CODE is attempted, and bit 56 of general register 0 is not zero.

23. Execution of COMPUTE INTERMEDIATE MESSAGE DIGEST, COMPUTE LAST MESSAGE DIGEST, or COMPUTE MESSAGE AUTHENTICATION CODE is attempted, and the R_2 field designates an odd-numbered register or general register 0.

24. Execution of the COMPUTE LAST MESSAGE DIGEST functions KLMD-SHAKE-128 or KLMD-SHAKE-256 is attempted, and the R_1 field designates an odd-numbered register or general register 0.

25. Execution of CONVERT HFP TO BFP, CONVERT TO FIXED (BFP or HFP), or LOAD FP INTEGER (BFP) is attempted, and the M_3 field does not designate a valid modifier.

26. Execution of DIVIDE TO INTEGER is attempted, and the M_4 field does not designate a valid modifier.

27. Execution of EXTRACT STACKED STATE is attempted, and the code in bit positions 56-63 of general register R_2 is greater than 4 when the ASN-and-LX-reuse facility is not installed or is greater than 5 when the facility is installed.

28. Execution of INVALIDATE DAT TABLE ENTRY is attempted, and bits 44-51 of general register R_2 are not all zeros.

29. Execution of INVALIDATE PAGE TABLE ENTRY is attempted, the IPTE-range facility is installed, the R_3 field is nonzero, and the page index in general register R_2 plus the additional-entry count in general register R_3 is greater than 255.

30. Execution of LOAD COUNT TO BLOCK BOUNDARY is attempted, and the M_3 field specifies a reserved value.

31. Execution of LOAD FPC or LOAD FPC AND SIGNAL is attempted, and one or more bits of the second operand corresponding to unsupported bits in the FPC register are one.

32. Execution of LOAD PAGE-TABLE-ENTRY ADDRESS is attempted and the M_4 field of the instruction contains any value other than 0000-0100 binary.

33. Execution of LOAD PSW is attempted and bit 12 of the doubleword at the second-operand address is zero. It is model dependent whether or not this exception is recognized.

34. Execution of MONITOR CALL is attempted, and bit positions 8-11 of the instruction do not contain zeros.

35. Execution of MOVE PAGE is attempted, and bit positions 48-51 of general register 0 do not contain zeros or bits 52 and 53 of the register are both one.

36. Execution of PACK ASCII is attempted, and the L_2 field is greater than 31.

37. Execution of PACK UNICODE is attempted, and the L_2 field is greater than 63 or is even.

38. Execution of PERFORM FLOATING POINT OPERATION is attempted, bit 32 of general register 0 is zero, and one or more fields in bits 33-63 are invalid or designate an uninstalled function.

39. Execution of PERFORM FRAME MANAGEMENT FUNCTION is attempted, and either or both of the following are true:

    - Bits 32-45, 52, 55, or 63 of general register R_1 are not zero.
    - The frame-size code in general register R_1 specifies a reserved value.

40. Execution of PERFORM LOCKED OPERATION is attempted, and any of the following is true:

    - The T bit, bit 55 of general register 0 is zero, and the function code in bits 56-63 of the register is invalid.
    - Bits 32-54 of general register 0 are not all zeros.

- In the access-register mode, for function codes that cause use of a parameter list containing an ALET, the R_3 field is zero.

41. Execution of PERFORM RANDOM NUMBER OPERATION is attempted, bit 56 of general register 0 is one, and the length in general register $R_2 + 1$ is greater than 512.

42. Execution of PERFORM TIMING FACILITY FUNCTION is attempted, and either of the following is true:

    - Bit 56 of general register 0 is not zero.
    - Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.

43. Execution of PROGRAM TRANSFER or PROGRAM TRANSFER WITH INSTANCE is attempted, and all of the following are true:

    - The extended-addressing-mode bit in the PSW is zero.
    - The basic-addressing-mode bit, bit 32, in the general register designated by the R_2 field of the instruction is zero.
    - Bits 33-39 of the instruction address in the same register are not all zeros.

44. Execution of RESUME PROGRAM is attempted, and any of the following is true:

    - Bits 31, 32, and 64-127 of the PSW field in the second operand are not valid for placement in the current PSW. The exception is recognized if any of the following is true:
        - Bits 31 and 32 are both zero and bits 64-103 are not all zeros.
        - Bits 31 and 32 are zero and one, respectively, and bits 64-96 are not all zeros.
        - Bits 31 and 32 are one and zero, respectively.
        - Bit 127 is one.
    - Bits 0-12 of the parameter list are not all zeros.

45. Execution of SEARCH STRING UNICODE is attempted, and bits 32-47 of general register 0 are not all zeros.

46. Execution of SET ADDRESS SPACE CONTROL or SET ADDRESS SPACE CONTROL FAST is attempted, and bits 52 and 53 of the second-operand address are not both zeros.

47. Execution of SET ADDRESSING MODE (SAM24) is attempted, and bits 0-39 of the unupdated instruction address in the PSW, bits 64-103 of the PSW, are not all zeros.

48. Execution of SET ADDRESSING MODE (SAM31) is attempted, and bits 0-32 of the unupdated instruction address in the PSW, bits 64-96 of the PSW, are not all zeros.

49. Execution of SET CLOCK PROGRAMMABLE FIELD is attempted, and bits 32-47 of general register 0 are not all zeros.

50. Execution of SET BFP ROUNDING MODE (SRNMB) is attempted, and either bits 56-60 of the second-operand address are not all zeros, or bits 61-63 of the second-operand address do not designate a valid rounding mode.

51. Execution of SET FPC or SET FPC AND SIGNAL is attempted, and one or more bits of the first operand corresponding to unsupported bits in the FPC register are one.

52. Execution of STORE SYSTEM INFORMATION is attempted, the function code in general register 0 is valid, and bits 36-55 of general register 0 and bits 32-47 of general register 1 are not all zeros.

53. Execution of TRANSACTION ABORT is attempted, and the second-operand address is between 0 and 255

54. Execution of TRANSACTION BEGIN (TBEGIN) is attempted, and the program-interruption-filtering control, bits 14-15 of I_2 field of the instruction, contains the value 3.

55. Execution of TRANSACTION BEGIN (TBEGINC) is attempted, and the B_1 field is not zero.

56. Execution of TRANSLATE AND TEST EXTENDED or TRANSLATE AND TEST REVERSE EXTENDED is attempted, the A bit, bit 0 of the M_3 field, is one, and the argument-character length in general register $R_1 + 1$ is not an even number.

57. Execution of TRANSLATE TWO TO ONE or TRANSLATE TWO TO TWO is attempted, and the length in general register $R_1 + 1$ does not specify an even number of bytes.

58. Execution of UNPACK ASCII is attempted, and the L_1 field is greater than 31.

59. Execution of UNPACK UNICODE is attempted, and the L_1 field is greater than 63 or is even.

60. Execution of UPDATE TREE is attempted, and the initial contents of general registers 4 and 5 are not a multiple of 8 in the 24-bit or 31-bit addressing mode or are not a multiple of 16 in the 64-bit addressing mode.

61. Execution of a vector instruction is attempted, and an instruction-specified reason was recognized. See instruction descriptions in Chapters 21-25 for detailed reasons for specification exceptions.

The execution of the instruction identified by the old PSW is suppressed. However, for early PSW specification exceptions (causes 1-3) the operation that introduces the new PSW is completed, but an interruption occurs immediately thereafter.

Except as noted below, the instruction-length code (ILC) is 1, 2, or 3, indicating the length of the instruction causing the exception.

When the instruction address is odd (cause 4 on page 6-40), it is unpredictable whether the ILC is 1, 2, or 3.

When the exception is recognized because of an early PSW specification exception (causes 1-3) and the exception has been introduced by LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, or an interruption, the ILC is 0. When the exception is introduced by SET ADDRESSING MODE (SAM24, SAM31), the ILC is 1, or it is 2 if SET ADDRESSING MODE was the target of EXECUTE, or 3 if the SET ADDRESSING MODE was the target of EXECUTE RELATIVE LONG. When the exception is introduced by SET SYSTEM MASK or by STORE THEN OR SYSTEM MASK, the ILC is 2, except that when the SSM or STOSM instruction is the target of EXECUTE RELATIVE LONG, the ILC is 3.

The specification exception is indicated by a program-interruption code of 0006 hex (or 0086, 0206, or 0286 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Programming Note: See "Exceptions Associated with the PSW" on page 6-9 for a definition of when the exceptions associated with the PSW are recognized.

Stack-Empty Exception

A stack-empty exception is recognized during the unstacking process in EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, MODIFY STACKED STATE, or PROGRAM RETURN when the current linkage-stack entry is a header entry and the backward stack-entry validity bit in the header entry is zero.

The operation is nullified.

The instruction-length code is 1 or 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The stack-empty exception is indicated by a program-interruption code of 0031 hex (or 00B1 hex if a concurrent PER event is indicated).

Stack-Full Exception

A stack-full exception is recognized during the stacking process in BRANCH AND STACK or stacking PROGRAM CALL when there is not enough remaining free space in the current linkage-stack section and the forward-section validity bit in the trailer entry of the section is zero.

The operation is nullified.

The instruction-length code is 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The stack-full exception is indicated by a program-interruption code of 0030 hex (or 00B0 hex if a concurrent PER event is indicated).

Stack-Operation Exception

A stack-operation exception is recognized during the unstacking process in PROGRAM RETURN when the unstack-suppression bit is one in any linkage-stack state entry or header entry encountered during the process.

The operation is nullified.

The instruction-length code is 1 or 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The stack-operation exception is indicated by a program-interruption code of 0034 hex (or 00B4 hex if a concurrent PER event is indicated).

Stack-Specification Exception

A stack-specification exception is recognized in each of the following cases:

1. During the stacking process in BRANCH AND STACK or stacking PROGRAM CALL when there is not enough remaining free space in the current linkage-stack section and either of the following is true:

   a. The remaining-free-space value used to locate the trailer entry of the current section is not a multiple of 8.

   b. There is not enough remaining free space in the next section.

2. During the unstacking process in EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, MODIFY STACKED STATE, or PROGRAM RETURN when the current linkage-stack entry is a header entry in which the backward stack-entry address designates another header entry.

The operation is nullified.

The instruction-length code is 1 or 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The stack-specification exception is indicated by a program-interruption code of 0032 hex (or 00B2 hex if a concurrent PER event is indicated).

Stack-Type Exception

A stack-type exception is recognized during the unstacking process in EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, MODIFY STACKED STATE, or PROGRAM RETURN in each of the following cases:

1. The current linkage-stack entry is not a header entry or a state entry.

2. When the current linkage-stack entry is a header entry, the preceding entry, designated by the backward stack-entry address in the header entry, is not a header entry or a state entry. (A stack-specification exception is recognized if the preceding entry is a header entry.)

The operation is nullified.

The instruction-length code is 1 or 2, except that when the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

The stack-type exception is indicated by a program-interruption code of 0033 hex (or 00B3 hex if a concurrent PER event is indicated).

Trace-Table Exception

A trace-table exception is recognized when the CPU attempts to store a trace-table entry which would reach or cross the next 4 K-byte block boundary. For the purpose of recognizing this exception in the TRACE instruction, the explicit trace entry is treated as being 76 bytes long for TRACE (TRACE) and as 144 bytes long for TRACE (TRACG). For a PROGRAM CALL instruction that would cause storing of both a PROGRAM CALL trace entry and a mode-switch trace entry, the exception is recognized for the first entry when either the first or the second entry would reach or cross the boundary.

The operation is nullified.

The instruction-length code is 1, 2, or 3, indicating the length of the instruction causing the exception.

The trace-table exception is indicated by a program-interruption code of 0016 hex (or 0096 hex if a concurrent PER event is indicated).

Transaction-Constraint Exception

A transaction-constraint exception is recognized when a constrained transaction violates one or more of its constraints. See “Constrained Transaction” on page 5-107 for details on the constraints.

The unit of operation is suppressed.

When the exception occurs as a result of the fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. Otherwise, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction at which the exception was detected.

The transaction-constraint exception is indicated by a program-interruption code of 0218 hex (or 0298 hex if a concurrent PER event is indicated).

Programming Note: The instruction designated by the aborted-transaction instruction address is not necessarily the instruction that caused the transaction-constraint exception.

Transactional-Execution-Aborted Event

A transactional-execution-aborted event is recognized when a transaction is aborted due to a program-interruption condition that results in an interruption.

The instruction-length code stored in bits 5-6 of real location 141, and, when a TBEGIN-specified transaction diagnostic block (TDB) is stored, the instruction-length code stored in bits 5-6 of byte 37 of the TDB, is 1, 2, or 3, reflecting the length of the instruction that caused the abort, except that when the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3.

The transactional-execution-aborted event is indicated by setting bit 6 of the program-interruption code at real locations 142-143 to one. The remainder of the program-interruption code at real locations 142-143 identifies the program-interruption condition that caused the abort. When a TBEGIN-specified transaction diagnostic block is stored, bit 6 is not set in the program-interruption code in bytes 38-39 of the TDB.

See “Transaction Abort Processing” on page 5-102 for a detailed description of transaction abort processing due to program interruptions.

Translation-Specification Exception

A translation-specification exception is recognized when translation of a virtual address is attempted and any of the following is true:

1. In the lookup in the table designated by the address-space-control element used for the translation, the table-type bits in the selected table entry do not equal the designation-type bits in the address-space-control element.
2. In a lookup in a table designated by an entry in a region first table, region second table, or region third table, the value of the table-type bits in the selected table entry is not one less than the value of the same bits in the designating table entry.
3. The private-space control, bit 55 in the address-space-control element used for the translation, is one, and either of the following is true:
   - The segment-table entry used for the translation is valid, and the common-segment bit, bit 59, in the segment-table entry is one.
   - EDAT-2 applies, the region-third-table entry used for the translation is valid, and the common-region bit, bit 59, in the region-third-table entry is one.
4. The page-table entry used for the translation is valid, and bit position 52 does not contain zero.
5. The page-table entry used for the translation is valid, EDAT-1 does not apply, the instruction-execution-protection facility is not installed, and bit position 55 does not contain zero. It is model dependent whether this condition is recognized.

Any of the above reasons is referred to by saying that the DAT-table entry has a format error.

The exception is recognized only as part of the execution of an instruction using address translation, that is, when DAT is on and a logical address, instruction address, or virtual address must be translated, or when LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS or STORE REAL ADDRESS is executed.

The unit of operation is suppressed.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2. When the exception occurs during the fetching of the target of EXECUTE RELATIVE LONG, the ILC is 3.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The translation-specification exception is indicated by a program-interruption code of 0012 hex (or 0092, 0212, or 0292 hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Programming Note: When a translation-specification exception is recognized in the process of translating an instruction address, the operation is suppressed. In this case, the instruction-length code (ILC) is needed to derive the address of the instruction, as the instruction address in the old PSW has been incremented by the amount indicated by the ILC. In the case of region-first-translation, region-second-translation, region-third-translation, segment-translation, and page-translation exceptions, the operation is nullified, the instruction address in the old PSW identifies the instruction, and the ILC may be arbitrarily set to 1, 2, or 3.

Vector Processing Exception

A vector-processing exception is recognized when there is an exception while executing a vector floating point instruction. See the section "IEEE Exceptions" on page 19-7 for details.

The operation is suppressed.

The instruction-length code is 3, except that when the instruction is target of EXECUTE, the ILC is 2.

Each of the IEEE exceptions is controlled by a mask bit in the floating-point-control (FPC) register. The handling of these exceptions is described in the section "IEEE Exceptions" on page 9-18.

The vector-processing exception is indicated by a program-interruption code of 001B hex (or 009B, 021B, or 029B hex, if a concurrent PER event, a concurrent transactional-execution-aborted event, or both are indicated, respectively).

Collective Program-Interruption Names

For the sake of convenience, certain program exceptions are grouped together under a single collective name. These collective names are used when it is necessary to refer to the complete set of exceptions, such as in instruction definitions. Four collective names are used:

- Access exceptions
- ASN-translation exceptions
- Subspace-replacement exceptions
- Trace exceptions

The individual exceptions and their priorities are listed in "Multiple Program-Interruption Conditions" on page 6-50.

Recognition of Access Exceptions

Figure 6-7 on page 6-47 summarizes the conditions that can cause access exceptions and the action taken when they are encountered.

Condition[2]	Translation for Virtual Address of LPTEA		Translation for Virtual Address of LOAD REAL ADDRESS		Translation for TAR and TPROT, and Access for Logical Address of TPROT[1]		Translation and Access for Any Other Address	
	Indication	Action	Indication	Action	Indication	Action	Indication	Action
Access register[3]								
Bits 0-6 not all zeros	cc3	Complete	cc3	Complete	cc3	Complete	AS	Suppress
Effective access-list designation								
Invalid address of designation	A	Suppress	A	Suppress	A	Suppress	A	Suppress
Access-list entry[3]								
Access-list-length violation	cc3	Complete	cc3	Complete	cc3	Complete	AT	Nullify
Invalid address of entry	A	Suppress	A	Suppress	A	Suppress	A	Suppress
I bit on	cc3	Complete	cc3	Complete	cc3	Complete	AT	Nullify
Sequence number in access register not equal to sequence number in entry	cc3	Complete	cc3	Complete	cc3	Complete	ALQ	Nullify

Figure 6-7. Handling of Access Exceptions (Part 1 of 4)

Condition[2]	Translation for Virtual Address of LPTEA		Translation for Virtual Address of LOAD REAL ADDRESS		Translation for TAR and TPROT, and Access for Logical Address of TPROT[1]		Translation and Access for Any Other Address	
	Indi-cation	Action	Indi-cation	Action	Indi-cation	Action	Indi-cation	Action
ASN-second-table entry[3]								
Invalid address of entry	A	Suppress	A	Suppress	A	Suppress	A	Suppress
I bit on	cc3	Complete	cc3	Complete	cc3	Complete	AV	Nullify
Sequence number in access-list entry not equal to sequence number in entry	cc3	Complete	cc3	Complete	cc3	Complete	ASQ	Nullify
Authority-table entry[3,4]								
Authority-table-length violation	cc3	Complete	cc3	Complete	cc3	Complete	EA	Nullify
Invalid address of entry	A	Suppress	A	Suppress	A	Suppress	A	Suppress
Second-authority bit not one	cc3	Complete	cc3	Complete	cc3	Complete	EA	Nullify
Address-space-control element								
Bits 0-10, 0-21, or 0-32 of instruction or operand address not all zeros when address-space-control element is a region-second-table designation, region-third-table designation, or segment-table designation, respectively	cc3	Complete	cc3	Complete	cc3	Complete	ATY	Nullify
Region-table-entry designation by address-space-control element or region-table-entry								
Entry outside of table	cc3	Complete	cc3	Complete	cc3	Complete	RT	Nullify
Invalid address of entry	A	Suppress	A	Suppress	A	Suppress	A	Suppress
I bit on	cc2	Complete	cc3	Complete	cc3	Complete	RT	Nullify
TT in entry not equal DT in address-space-control element or not one less than TT in next-higher-level entry	TS	Suppress	TS	Suppress	TS	Suppress	TS	Suppress
Segment-table entry designation by address-space-control element or region-table-entry								
Entry outside of table	cc3	Complete	cc3	Complete	cc3	Complete	ST	Nullify
Invalid address of entry	A	Suppress	A	Suppress	A	Suppress	A	Suppress
I bit on (except as follows)	cc2	Complete	cc1	Complete	cc3	Complete	ST	Nullify
I bit on (LRA in 24-bit or 31-bit mode when bits 0-32 of entry address not all zeros)	-	-	cc3	Complete	-	-	-	-
One in a bit position which is checked for zero[5]	TS	Suppress	TS	Suppress	TS	Suppress	TS	Suppress
TT in entry not equal DT in address-space-control element or not one less than TT in next-higher-level entry (TT not zero)	TS	Suppress	TS	Suppress	TS	Suppress	TS	Suppress

Figure 6-7. Handling of Access Exceptions (Part 2 of 4)

Condition[2]	Translation for Virtual Address of LPTEA		Translation for Virtual Address of LOAD REAL ADDRESS		Translation for TAR and TPROT, and Access for Logical Address of TPROT[1]		Translation and Access for Any Other Address	
	Indi-cation	Action	Indi-cation	Action	Indi-cation	Action	Indi-cation	Action
Page-table entry								
Invalid address of entry	-	-	A	Suppress	A	Suppress	A	Suppress
I bit on (except as follow)	-	-	cc2	Complete	cc3	Complete	PT	Nullify
I bit on (LRA in 24-bit or 31-bit mode when bits 0-32 of entry address not all zeros)	-	-	cc3	Complete	cc3	Complete	PT	Nullify
One in a bit position which is checked for zeros[5]	-	-	TS	Suppress	TS	Suppress	TS	Suppress
Access for instruction fetch								
Location protected (key-controlled or instruction-execution protection)	-	-	-	-	-	-	P	Suppress
Invalid address	-	-	-	-	-	-	A	Suppress
Access for operand								
Location protected (low-address, DAT, or key-controlled protection)	-	-	-	-	cc set[6]	Complete	P	Term.*
Invalid address	-	-	-	-	A	Suppress	A	Term.*

Explanation

- The condition does not apply.

* Action is to terminate except where otherwise specified in this publication. For access-list-controlled protection, DAT protection, and instruction-execution protection, the action is always to suppress.

1 TAR does not have a logical address. The rows “Address-space-control element” through “Access for operands” apply only to TPROT, not to TAR.

2 Protection applies only to accesses for instruction fetch and for operands. It does not apply to the fetching of the effective access-list designation or any of the listed entries.

3 Exceptions related to an access register, effective access-list designation, access-list entry, ASN-second-table entry, or authority-table entry are recognized only in the access-register mode except that, for LOAD REAL ADDRESS and STORE REAL ADDRESS, they are recognized when PSW bits 16 and 17 are 01 binary, and, for TEST ACCESS, they are recognized regardless of the translation mode.

4 Authority table is not accessed and secondary-authority bit is not checked if the private bit in the access-list entry is zero or the access-list- entry authorization index in the access-list entry is equal to the extended authorization index in control register 8.

5 A translation-specification exception for a format error in a table entry is recognized only when the execution of an instruction requires the entry for translation of an address.

6 The condition code is set as follows:
0 Operand location not protected.
1 Fetches permitted, but stores not permitted.
2 Neither fetches nor stores permitted.

A Addressing exception.

ALQ ALE-sequence exception.

AS ALET-specification exception.

ASQ ASTE-sequence exception.

AT ALEN-translation exception.

ATY ASCE-type exception.

Figure 6-7. Handling of Access Exceptions (Part 3 of 4)

Condition[2]		Translation for Virtual Address of LPTEA		Translation for Virtual Address of LOAD REAL ADDRESS		Translation for TAR and TPROT, and Access for Logical Address of TPROT[1]		Translation and Access for Any Other Address	
		Indication	Action	Indication	Action	Indication	Action	Indication	Action
AV	ASTE-validity exception.								
cc1	Condition code 1 set.								
cc2	Condition code 2 set.								
cc3	Condition code 3 set.								
EA	Extended-authority exception.								
P	Protection exception.								
PT	Page-translation exception.								
RT	Region-first-translation, region-second-translation, or region-third translation exception, depending on the level of the table.								
ST	Segment-translation exception.								
TS	Translation-specification exception.								

Figure 6-7. Handling of Access Exceptions (Part 4 of 4)

Any access exception is recognized as part of the execution of the instruction with which the exception is associated. An access exception is not recognized when the CPU attempts to prefetch from an unavailable location or detects some other access-exception condition, but a branch instruction or an interruption changes the instruction sequence such that the instruction is not executed.

Every instruction can cause an access exception to be recognized because of instruction fetch. Additionally, access exceptions associated with instruction execution may occur because of an access to an operand in storage.

An access exception due to fetching an instruction is indicated when the first instruction halfword cannot be fetched without encountering the exception. When the first halfword of the instruction has no access exceptions, access exceptions may be indicated for additional halfwords according to the instruction length specified by the first two bits of the instruction; however, when the operation can be performed without accessing the second or third halfwords of the instruction, it is unpredictable whether the access exception is indicated for the unused part. Since the indication of access exceptions for instruction fetch is common to all instructions, it is not covered in the individual instruction definitions.

Except where otherwise indicated in the individual instruction description, the following rules apply for exceptions associated with an access to an operand location. For a fetch-type operand, access exceptions are necessarily indicated only for that portion of the operand which is required for completing the operation. It is unpredictable whether access exceptions are indicated for those portions of a fetch-type operand which are not required for completing the operation. For a store-type operand, access exceptions are recognized for the entire operand even if the operation could be completed without the use of the inaccessible part of the operand. In situations where the value of a store-type operand is defined to be unpredictable, it is unpredictable whether an access exception is indicated.

Whenever an access to an operand location can cause an access exception to be recognized, the word “access” is included in the list of program exceptions in the description of the instruction. This entry also indicates which operand can cause the exception to be recognized and whether the exception is recognized on a fetch or store access to that operand location. Access exceptions are recognized only for the portion of the operand as defined for each particular instruction.

Multiple Program-Interruption Conditions

Except for PER basic events, only one program-interruption condition is indicated with a program interrup-

tion. The existence of one condition, however, does not preclude the existence of other conditions. When more than one program-interruption condition exists, only the condition having the highest priority is identified in the interruption code.

When multiple conditions of the same priority apply, it is unpredictable which is indicated.

When multiple parts of the same storage operand are subject to separate access controls, the priority of access exceptions associated with the parts is unpredictable and is not necessarily related to the sequence specified for the access of bytes within the operand. For example, when (a) the first operand of a MOVE (MVC) instruction crosses a segment boundary, (b) the invalid bit is one in the segment-table entry used to translate the leftmost part of the operand, and (c) the DAT-protection bit is one in a valid page-table entry used to translate the rightmost part of the operand, then it is unpredictable whether a segment-translation exception or protection exception is recognized.

When an instruction has two storage operands and access-exception conditions exist for both operands, it is unpredictable which condition is recognized. A subsequent execution of the same instruction (with the same exception conditions) may result in the exception condition being recognized for the same operand as the first execution, or for the other operand.

The type of ending which occurs (nullification, suppression, or termination) is that which is defined for the type of exception that is indicated in the interruption code. However, if a condition is indicated which permits termination, and another condition also exists which would cause either nullification or suppression, then the unit of operation is suppressed.

Figure 6-8 on page 6-52 lists the priorities of all program-interruption conditions other than PER basic events and exceptions associated with some of the more complex control instructions. All exceptions associated with references to storage for a particular instruction halfword or a particular operand byte are grouped as a single entry called “access.” Figure 6-9 on page 6-54 lists the priority of access exceptions for a single access. Thus, the second figure specifies which of several exceptions, encountered either in the access of a particular portion of an instruction or in any particular access associated with an operand, has highest priority, and the first figure specifies the priority of this condition in relation to other conditions detected in the operation. Similarly, the priorities for exceptions occurring as part of ASN translation and tracing are covered in Figure 6-10 on page 6-56 and Figure 6-12 on page 6-56, respectively.

For some instructions, the priority is shown in the individual instruction description.

The relative priorities of any two conditions listed in the figure can be found by comparing the priority numbers, as found in the figure, from left to right until a mismatch is found. If the first inequality is between numeric characters, either the two conditions are mutually exclusive or, if both can occur, the condition with the smaller number is indicated. If the first inequality is between alphabetic characters, then the two conditions are not exclusive, and it is unpredictable which is indicated when both occur.

To understand the use of the table, consider an example involving the instruction ADD DECIMAL, which is a six-byte instruction. Assume that the first four bytes of the instruction can be accessed but that the instruction crosses a boundary so that an addressing exception exists for the last two bytes. Additionally, assume that the first operand addressed by the instruction contains invalid decimal digits and is in a location that can be fetched from, but not stored into, because of key-controlled protection. The three exceptions which could result from attempted execution of the ADD DECIMAL are:

Priority Number	**Exception**
6.2.B	Access exceptions for third instruction halfword
8.B	Access exceptions (operand 1).
8.D	Data exception

Since the first inequality (6≠8) is between numeric characters, the addressing exception would be indicated. If, however, the entire ADD DECIMAL instruction can be fetched, and only the second two exceptions listed above exist, then the inequality (B ≠D) is between alphabetic characters, and it is unpre-

dictable whether the protection exception or the data exception would be indicated.

Any	Transaction constraint exception, except for one caused by a restricted instruction[8]
1.	Specification exception due to any PSW error of the type that causes an immediate interruption.[1]
2.	Specification exception due to an odd instruction address in the PSW.
3.	Access exceptions for first halfword of an execute-type instruction.[2]
4.A	Access exceptions for second halfword of an execute-type instruction.[2]
4.B	Access exceptions for third halfword of EXECUTE RELATIVE LONG.[2]
5.1	PER event due to instruction-fetching nullification on an execute-type instruction. (With PER-3)
5.2	Specification exception due to target instruction of EXECUTE not being specified on halfword boundary.[2]
6.1	Access exceptions for first instruction halfword.
6.2.A	Access exceptions for second instruction halfword.[3] (With PER-3)
6.2.B	Access exceptions for third instruction halfword.[3] (With PER-3)
6.3	PER event due to instruction-fetching nullification. (PER-3)
7.A	Access exceptions for second instruction halfword.[3] (Without PER-3)
7.B	Access exceptions for third instruction halfword. [3] (Without PER-3)
7.C.1	Operation exception.
7.C.2	Privileged-operation exception for privileged instructions.
7.C.3	Execute exception.
7.C.4	Special-operation exception.[4]
7.D	Transaction-constraint exception due to any restricted instruction.
8.A	Specification exception due to conditions other than those included in 1, 2, and 5.2 above.
8.B[5]	Access exceptions for an access to an operand in storage.[6]
8.C[5]	Access exceptions for any other access to an operand in storage.[6]
8.D.1	Data exception.[7]
8.D.2	Vector-processing exception.
8.E	Decimal-divide exception.[7]
8.F	Trace exceptions.
9.	Events other than PER events, exceptions which result in completion, and the following exceptions: • Fixed-point divide, • Floating-point divide, • Operand, • Special-operation recognized by any of the following: – LOAD PAGE-TABLE-ENTRY ADDRESS due to a real-space designation – LOAD REAL ADDRESS See the definition of the TRAP instruction for other priorities of the special-operation exception. • Square root. Either these exceptions and events are mutually exclusive or their priority is specified in the corresponding definitions.

Explanation:

Numbers indicate priority, with “1” being the highest priority; letters indicate no priority.

Figure 6-8. Priority of Program-Interruption Conditions

[1] PSW errors which cause an immediate interruption may be introduced by a new PSW loaded as a result of an interruption or by the instructions BRANCH AND SET AUTHORITY, LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, SET SYSTEM MASK, and STORE THEN OR SYSTEM MASK. The priority shown in the chart is for a PSW error introduced by an interruption and may also be considered as the priority for a PSW error introduced by the previous instruction. The error is introduced only if the instruction introducing the error encounters no other exceptions. The resulting interruption has a higher priority than any interruption caused by the instruction which would have been executed next; it has lower priority, however, than any interruption caused by the instruction which introduced the erroneous PSW.

[2] Priorities 3, 4.A, and 5.1 are for both EXECUTE and EXECUTE RELATIVE LONG, priority 4.B is for EXECUTE RELATIVE LONG, priority 5.2 is for EXECUTE; priorities starting with 6 are for the target instruction. When no execute-type instruction is encountered, priorities 3-5 do not apply.

[3] Separate accesses may occur for each halfword of an instruction. The second instruction halfword is accessed only if bits 0-1 of the instruction are not both zeros. The third instruction halfword is accessed only if bits 0-1 of the instruction are both ones. Access exceptions for one of these halfwords are not necessarily recognized if the instruction can be completed without use of the contents of the halfword or if an exception of lower priority can be determined without the use of the halfword.

[4] Except for the special-operation-exception conditions listed in priority 9.

[5] As in instruction fetching, separate accesses may occur for each portion of an operand. Each of these accesses, and also accesses for different operands, are of equal priority, and the two entries 8.B and 8.C are listed to represent the relative priorities of exceptions associated with any two of these accesses. Access exceptions for INSERT STORAGE KEY EXTENDED, INSERT VIRTUAL STORAGE KEY, INVALIDATE PAGE TABLE ENTRY, LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, PERFORM FRAME MANAGEMENT FUNCTION, RESET REFERENCE BIT EXTENDED, SET STORAGE KEY EXTENDED, STORE REAL ADDRESS, and TEST PROTECTION are also included in 8.B.

[6] For MOVE LONG, MOVE LONG EXTENDED, COMPARE LOGICAL LONG, and COMPARE LOGICAL LONG EXTENDED, an access exception for a particular operand can be indicated only if the R field for that operand designates an even-numbered register.

[7] The exception can be indicated only if any operand data fetched that is responsible for the exception were fetched without encountering an access exception.

[8] Except as noted, the transaction-constraint exception may occur at any priority.

Figure 6-8. Priority of Program-Interruption Conditions (Continued)

Access Exceptions

The access exceptions consist of those exceptions which can be encountered while using an absolute, instruction, logical, real, or virtual address to access storage. Thus, in the access-register mode, the exceptions are:

1. ALET specification
2. ALEN translation
3. ALE sequence
4. ASTE validity
5. ASTE sequence
6. Extended authority
7. Addressing (the ART tables)
8. ASCE type
9. Region first translation
10. Region second translation
11. Region third translation
12. Segment translation
13. Page translation
14. Translation specification
15. Addressing (the DAT tables)
16. Addressing (the operand or instruction)
17. Protection (access-list-controlled, DAT, instruction-execution, key-controlled, and low-address)

With DAT on but in other than the access-register mode, exceptions 8-17 in the above list, except for access-list-controlled protection, can be encountered.

With DAT off, the exceptions are:

18. Addressing (the operand or instruction)
19. Protection (key-controlled and low-address)

Additionally, even with DAT off, the instruction STORE REAL ADDRESS can encounter exceptions 1-17, the instructions LOAD PAGE-TABLE-ENTRY ADDRESS and LOAD REAL ADDRESS can encoun-

ter exceptions 7, 14, and 15, and the instructions COMPARE AND REPLACE DAT TABLE ENTRY, INVALIDATE DAT TABLE ENTRY, and INVALIDATE PAGE TABLE ENTRY can encounter exception 15.

The access exceptions are listed in more detail in Figure 6-9 on page 6-54.

Programming Note: The priorities in Figure 6-9 on page 6-54 could be renumbered, but they are kept as they are to allow easier comparison to the corresponding ESA/390 priorities. Specifically, B.1.A.1-B.1.A.9 could be changed to B.1-B.9, but ESA/390 contains "B.1.B Translation-specification exception due to invalid encoding of bits 8-12 of control register 0."

A.	Protection exception (low-address protection) due to a store-type operand reference with an effective address in the range 0-511 or 4096-4607. Not recognized if DAT is on and the address-space-control element to be used in the translation cannot be obtained because of another exception.
B.1.A.1	ALET-specification exception due to bits 0-6 of access register not being all zeros.[1]
B.1.A.2	Addressing exception for access to effective access-list designation.[2]
B.1.A.3	ALEN-translation exception due to access-list entry being outside the list.[1]
B.1.A.4	Addressing exception for access to access-list entry.[2]
B.1.A.5	ALEN-translation exception due to I bit in access-list entry having the value one.[1]
B.1.A.6	ALE-sequence exception due to access-list-entry sequence number (ALESN) in access register not being equal to ALESN in access-list entry.[1]
B.1.A.7	Addressing exception for access to ASN-second-table entry.[2]
B.1.A.8	ASTE-validity exception due to I bit in ASN-second-table entry having the value one.[1]
B.1.A.9	ASTE-sequence exception due to ASN-second-table-entry sequence number (ASTESN) in access-list entry not being equal to ASTESN in ASN-second-table entry.[1] **Note:** Exceptions B.1.A.10 through B.1.A.12 are recognized only when the private bit in the access-list entry is one and the ALEAX in the entry is not equal to the EAX in control register 8.
B.1.A.10	Extended-authority exception due to authority-table entry being outside table.[1]
B.1.A.11	Addressing exception for access to authority-table entry.[2]
B.1.A.12	Extended-authority exception due to (1) private bit in access-list entry not being zero, (2) access-list-entry authorization index in access-list entry not being equal to extended authorization index in control register 8, and (3) secondary-authority bit selected by extended authorization index not being one.[1]
B.2.A	Protection exception (access-list-controlled protection) due to store-type operand reference to a virtual address which is protected against stores.[1]
B.2.B.1	ASCE-type exception due to bits 0-10, 0-21, or 0-32 of instruction or operand address not being zeros when address-space-control element is a region-second-table designation, region-third-table designation, or segment-table designation, respectively.[3]
B.2.B.2	Region-first-, region-second-, region-third-, or segment-translation exception due to required entry in table designated by address-space-control element being outside of table.[3] **Note:** Exceptions B.2.B.3 through B.2.B.6 are recognized for a region-first-table, region-second- table, region-third-table, and segment-table entry in the order in which the entries are used.
B.2.B.3	Addressing exception for access to table entry.[4]
B.2.B.4	Region-first-, region-second-, region-third-, or segment-translation exception due to I bit in table entry having the value one.[3]

Figure 6-9. Priority of Access Exceptions (Part 1 of 2)

B.2.B.5	Translation-specification exception due to (1) TT in table entry not equal to DT in designating address-space-control element or not one less than TT in designating next-higher-level table entry, (2) invalid one in segment-table entry if this entry is a segment-table entry (common-segment bit if private- space bit in address-space-control element is one), or (3) invalid one in region-third-table entry if EDAT-2 applies and this entry is a region-third-table entry (common-region bit if private-space bit in address-space-control element is one).
B.2.B.6	Region-second-, region-third-, or segment-translation exception due to required entry in next-lower-level table entry, if any, being outside of table.[3]
B.2.B.7	Addressing exception for access to page-table entry.[5]
B.2.B.8	Page-translation exception due to I bit in page-table entry having the value one.[3, 7]
B.2.B.9	Translation-specification exception due to invalid ones in page-table entry in which I bit is zero.[8] A one in bit position 52 of a valid PTE always causes the exception. When EDAT-1 does not apply and the instruction-execution-protection facility is not installed, it is model dependent whether a one in bit position 55 of a valid PTE causes the exception.
B.3.A	Protection exception (instruction-execution protection) due to an instruction fetch from a virtual address which is protected from instruction execution.[6]
B.3.B	Protection exception (DAT protection) due to a store-type or update-type reference to a virtual address which is protected against stores.[6]
B.3.C	Addressing exception for access to instruction or operand.
B.4	Protection exception (key-controlled protection) due to attempt to access a protected instruction or operand location.

Explanation:

1 Not applicable when not in the access-register mode; not applicable for execution of TEST ACCESS and for translation of operand address of LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS, and TEST PROTECTION.

2 Not applicable when not in the access-register mode, except applicable for execution of TEST ACCESS and, when PSW bits 16 and 17 are 01 binary, for translation of operand address of LOAD REAL ADDRESS and second-operand address of STORE REAL ADDRESS; also applicable for LOAD PAGE-TABLE-ENTRY ADDRESS when the M_4 field is 0001 binary or when the M_4 field is 0100 binary and PSW bits 16-17 are 01 binary.

3 Not applicable when DAT is off except for translation of second-operand address of STORE REAL ADDRESS; not applicable to operand addresses of LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS and TEST PROTECTION.

4 Not applicable when DAT is off except for translation of operand address of LOAD PAGE-TABLE-ENTRY ADDRESS, LOAD REAL ADDRESS and second-operand address of STORE REAL ADDRESS.

5 Not applicable when DAT is off, except for execution of INVALIDATE PAGE TABLE ENTRY and for translation of operand address of LOAD PAGE-TABLE-ENTRY ADDRESS and LOAD REAL ADDRESS and second-operand address of STORE REAL ADDRESS.

6 Not applicable when DAT is off.

7 For MOVE PAGE, if the condition is true for both operands, the exception is recognized for the second operand. Also, if the condition-code-option bit is one, the exception is not recognized. Instead, condition code 1 is set if the condition is true for only the first operand, or condition code 2 is set if the condition is true for the second operand or both operands.

8 Not applicable when DAT is off except for translation of operand address of LOAD REAL ADDRESS and second-operand address of STORE REAL ADDRESS.

Figure 6-9. Priority of Access Exceptions (Part 2 of 2)

ASN-Translation Exceptions

The ASN-translation exceptions are those exceptions which are common to the process of translating an ASN in the instructions PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, and SET SECONDARY ASN WITH INSTANCE. The exceptions and the priority in which they are detected are shown in Figure 6-10.

1.	Addressing exception for access to ASN- first-table entry.
2.	AFX-translation exception due to I bit (bit 0) in ASN-first-table entry being one.
3.	Addressing exception for access to ASN-second-table entry.
4.	ASX-translation exception due to I bit (bit 0) in ASN-second-table entry being one.

Figure 6-10. Priority of ASN-Translation Exceptions

Subspace-Replacement Exceptions

The subspace-replacement exceptions are those exceptions which can be recognized during a subspace-replacement operation in PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, or SET SECONDARY ASN WITH INSTANCE. The exceptions and their priority are shown in Figure 6-11 on page 6-56.

1.	Addressing exception for access to dispatchable-unit control table.
2.	Addressing exception for access to subspace ASN-second-table entry.
3.	ASTE-validity exception due to bit 0 being one in subspace ASN-second-table entry.
4.	ASTE-sequence exception due to subspace ASN-second-table-entry sequence number in dispatchable-unit control table not being equal to ASN-second-table-entry sequence number in subspace ASN-second-table entry.

Figure 6-11. Priority of Subspace-Replacement Exceptions

Trace Exceptions

The trace exceptions are those exceptions which can be encountered while forming a trace-table entry. The exceptions and their priority are shown in Figure 6-12 on page 6-56.

A.	Protection exception (low-address protection) due to entry address being in the range 0-511 or 4096-4607.
B.1	Trace-table exception due to new entry reaching or crossing next 4 K-byte boundary.
B.2	Addressing exception for access to trace-table entry.

Figure 6-12. Priority of Trace Exceptions

Restart Interruption

The restart interruption provides a means for the operator or another CPU to invoke the execution of a specified program. The CPU cannot be disabled for this interruption.

In the z/Architecture architectural mode, a restart interruption causes the old PSW to be stored at real locations 288-303 and a new PSW, designating the start of the program to be executed, to be fetched from real locations 416-431. In the ESA/390-compatibility mode, a restart interruption causes the short-format old PSW to be stored at real locations 8-15 and a short-format new PSW to be fetched from real locations 0-7. The instruction-length code and interruption code are not stored.

If the CPU is in the operating state, the exchange of the PSWs occurs at the completion of the current unit of operation and after all other pending interruption conditions for which the CPU is enabled have been honored. If the CPU is in the stopped state, the CPU enters the operating state and exchanges the PSWs without first honoring any other pending interruptions.

The restart interruption is initiated by activating the restart key. The operation can also be initiated at the addressed CPU by executing a SIGNAL PROCESSOR instruction which specifies the restart order.

When the rate control is set to the instruction-step position, it is unpredictable whether restart causes a unit of operation or additional interruptions to be performed after the PSWs have been exchanged.

Programming Note: To perform a restart when the CPU is in the check-stop state, the CPU has to be reset. Resetting with loss of the least amount of information can be accomplished by means of the sys-

tem-reset-normal key, which does not clear the contents of program-addressable registers, including the control registers, but causes the channel subsystem to be reset. The CPU-reset SIGNAL PROCESSOR order can be used to clear the CPU without affecting the channel subsystem.

Supervisor-Call Interruption

The supervisor-call interruption occurs when the instruction SUPERVISOR CALL is executed. The CPU cannot be disabled for the interruption, and the interruption occurs immediately upon the execution of the instruction.

In the z/Architecture architectural mode, the supervisor-call interruption causes the old PSW to be stored at real locations 320-335 and a new PSW to be fetched from real locations 448-463. In the ESA/390-compatibility mode, the supervisor-call interruption causes the short-format old PSW to be stored at real locations 32-39 and a short-format new PSW to be fetched from real locations 96-103.

The contents of bit positions 8-15 of the SUPERVISOR CALL instruction are placed in the rightmost byte of the interruption code. The leftmost byte of the interruption code is set to zero. The instruction-length code is 1, unless the instruction was executed by means of an execute-type instruction. When the SUPERVISOR CALL instruction was executed by means of EXECUTE, the instruction-length code is 2; when it is executed by means of EXECUTE RELATIVE LONG, the instruction-length code is 3.

The interruption code is placed at real locations 138-139; the instruction-length code is placed in bit positions 5 and 6 of the byte at real location 137, with the other bits set to zeros; and zeros are stored at real location 136.

Priority of Interruptions

During the execution of an instruction, several interruption-causing events may occur simultaneously. The instruction may give rise to a program interruption, a request for an external interruption may be received, equipment malfunctioning may be detected, an I/O-interruption request may be made, and the restart key may be activated. Instead of the program interruption, a supervisor-call interruption might occur; or both can occur if PER is active. Simultaneous interruption requests are honored in a predetermined order.

An exigent machine-check condition has the highest priority. When it occurs, the current operation is terminated or nullified. Program and supervisor-call interruptions that would have occurred as a result of the current operation may be eliminated. Any pending repressible machine-check conditions may be indicated with the exigent machine-check interruption. Every reasonable attempt is made to limit the side effects of an exigent machine-check condition, and requests for external, I/O, and restart interruptions normally remain unaffected.

In the absence of an exigent machine-check condition, interruption requests existing concurrently at the end of a unit of operation are honored, in descending order of priority, as follows:

- Supervisor call
- Program
- Repressible machine check
- External
- Input/output
- Restart

The processing of multiple simultaneous interruption requests consists in storing the old PSW and fetching the new PSW belonging to the interruption first honored. This new PSW is subsequently stored without the execution of any instructions, and the new PSW associated with the next interruption is fetched. Storing and fetching of PSWs continues until no more interruptions are to be serviced. The priority is reevaluated after each new PSW is loaded. Each evaluation takes into consideration any additional interruptions which may have become pending. Additionally, external and I/O interruptions, as well as machine-check interruptions due to repressible conditions, occur only if the current PSW at the instant of evaluation indicates that the CPU is interruptible for the cause.

Instruction execution is resumed using the last-fetched PSW. The order of executing interruption subroutines is, therefore, the reverse of the order in which the PSWs are fetched.

If the new PSW for a program interruption does not specify the wait state and has an odd instruction address, or causes an access exception to be recog-

nized, another program interruption occurs. Since this second interruption introduces the same unacceptable PSW, a string of interruptions is established. These program exceptions are recognized as part of the execution of the following instruction, and the string may be broken by an external, I/O, machine-check, or restart interruption or by the stop function.

If the new PSW for a program interruption contains a one in bit position 12 or in an unassigned bit position, if the leftmost 40 bits of the instruction address are not zeros when bit 31 and 32 indicate 24-bit addressing, or the leftmost 33 bits are not zeros when bits 31 and 32 indicate 31-bit addressing, or if bit 32 is zero when bit 31 is one, another program interruption occurs. This condition is of higher priority than restart, I/O, external, or repressible machine-check conditions, or the stop function, and CPU reset has to be used to break the string of interruptions.

A string of interruptions for other interruption classes can also exist if the new PSW allows the interruption which has just occurred. These include machine-check interruptions, external interruptions, and I/O interruptions due to program-controlled-interruption (PCI) conditions generated because of CCWs which form a loop. Furthermore, a string of interruptions involving more than one interruption class can exist. For example, assume that the CPU timer is negative and the CPU-timer subclass mask is one. If the external new PSW has a one in an unassigned bit position, and the program new PSW is enabled for external interruptions, then a string of interruptions occurs, alternating between external and program. Even more complex strings of interruptions are possible. As long as more interruptions must be serviced, the string of interruptions cannot be broken by employing the stop function; CPU reset is required.

Similarly, CPU reset has to be invoked to terminate the condition that exists when an interruption is attempted with a prefix value designating a storage location that is not available to the CPU.

Interruptions for all requests for which the CPU is enabled occur before the CPU is placed in the stopped state. When the CPU is in the stopped state, restart has the highest priority.

Programming Note: The order in which concurrent interruption requests are honored can be changed to some extent by masking.

Chapter 7. General Instructions

Data Format. 7-3
Binary-Integer Representation. 7-4
Binary Arithmetic 7-5
Signed Binary Arithmetic 7-5
Addition and Subtraction 7-5
Fixed-Point Overflow 7-5
Unsigned Binary Arithmetic 7-6
Signed and Logical Comparison 7-7
Instructions 7-7
ADD 7-26
ADD IMMEDIATE 7-27
ADD HALFWORD 7-28
ADD HALFWORD IMMEDIATE. 7-28
ADD HIGH 7-29
ADD IMMEDIATE HIGH 7-29
ADD LOGICAL. 7-30
ADD LOGICAL IMMEDIATE 7-30
ADD LOGICAL HIGH. 7-31
ADD LOGICAL WITH CARRY. 7-31
ADD LOGICAL WITH SIGNED IMMEDIATE. 7-32
ADD LOGICAL WITH SIGNED IMMEDIATE HIGH 7-33
AND 7-33
AND IMMEDIATE 7-34
BRANCH AND LINK 7-35
BRANCH AND SAVE 7-36
BRANCH AND SAVE AND SET MODE 7-37
BRANCH AND SET MODE 7-38
BRANCH INDIRECT ON CONDITION 7-39
BRANCH ON CONDITION 7-40
BRANCH ON COUNT 7-41
BRANCH ON INDEX HIGH 7-42
BRANCH ON INDEX LOW OR EQUAL 7-42
BRANCH PREDICTION PRELOAD 7-43
BRANCH PREDICTION RELATIVE PRELOAD 7-43
BRANCH RELATIVE AND SAVE 7-46
BRANCH RELATIVE AND SAVE LONG. . . . 7-46
BRANCH RELATIVE ON CONDITION 7-46
BRANCH RELATIVE ON CONDITION LONG 7-46
BRANCH RELATIVE ON COUNT. 7-47
BRANCH RELATIVE ON COUNT HIGH. . . . 7-47
BRANCH RELATIVE ON INDEX HIGH. 7-48
BRANCH RELATIVE ON INDEX LOW OR EQUAL 7-48
CHECKSUM 7-49
CIPHER MESSAGE 7-53
CIPHER MESSAGE WITH CHAINING 7-53
CIPHER MESSAGE WITH AUTHENTICATION 7-78
CIPHER MESSAGE WITH CIPHER FEEDBACK. 7-92
CIPHER MESSAGE WITH COUNTER. 7-107
CIPHER MESSAGE WITH OUTPUT FEEDBACK. 7-120
COMPARE 7-134
COMPARE IMMEDIATE. 7-134
COMPARE RELATIVE LONG 7-135
COMPARE AND BRANCH. 7-135
COMPARE AND BRANCH RELATIVE. 7-135
COMPARE IMMEDIATE AND BRANCH . . . 7-136
COMPARE IMMEDIATE AND BRANCH RELATIVE. 7-136
COMPARE AND FORM CODEWORD. 7-137
COMPARE AND SWAP 7-144
COMPARE DOUBLE AND SWAP 7-144
COMPARE AND SWAP AND STORE 7-146
COMPARE AND TRAP. 7-149
COMPARE IMMEDIATE AND TRAP 7-149
COMPARE HALFWORD 7-150
COMPARE HALFWORD IMMEDIATE 7-150
COMPARE HALFWORD RELATIVE LONG. 7-150
COMPARE HIGH 7-151
COMPARE IMMEDIATE HIGH. 7-151
COMPARE LOGICAL 7-152
COMPARE LOGICAL IMMEDIATE 7-152
COMPARE LOGICAL RELATIVE LONG . . . 7-153
COMPARE LOGICAL AND BRANCH 7-154
COMPARE LOGICAL AND BRANCH RELATIVE. 7-154
COMPARE LOGICAL IMMEDIATE AND BRANCH. 7-154
COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE 7-154
COMPARE LOGICAL AND TRAP 7-155
COMPARE LOGICAL IMMEDIATE AND TRAP 7-156
COMPARE LOGICAL CHARACTERS UNDER MASK 7-157
COMPARE LOGICAL HIGH 7-157
COMPARE LOGICAL IMMEDIATE HIGH . . 7-158
COMPARE LOGICAL LONG 7-158
COMPARE LOGICAL LONG EXTENDED . . 7-160
COMPARE LOGICAL LONG UNICODE . . . 7-163
COMPARE LOGICAL STRING. 7-166
COMPARE UNTIL SUBSTRING EQUAL . . . 7-167
COMPRESSION CALL. 7-170
COMPUTE INTERMEDIATE MESSAGE DIGEST. 7-188
COMPUTE LAST MESSAGE DIGEST. 7-201

© Copyright IBM Corp. 2000, 2017

COMPUTE MESSAGE AUTHENTICATION CODE. 7-219
CONVERT TO BINARY 7-230
CONVERT TO DECIMAL. 7-231
CONVERT UTF-16 TO UTF-32 7-231
CONVERT UTF-16 TO UTF-8 7-234
CONVERT UNICODE TO UTF-8. 7-234
CONVERT UTF-32 TO UTF-16 7-238
CONVERT UTF-32 TO UTF-8 7-241
CONVERT UTF-8 TO UTF-16 7-244
CONVERT UTF-8 TO UNICODE. 7-244
CONVERT UTF-8 TO UTF-32 7-248
COPY ACCESS 7-252
DIVIDE 7-252
DIVIDE LOGICAL. 7-253
DIVIDE SINGLE 7-254
EXCLUSIVE OR. 7-254
EXCLUSIVE OR IMMEDIATE 7-256
EXECUTE 7-256
EXECUTE RELATIVE LONG. 7-256
EXTRACT ACCESS. 7-257
EXTRACT CPU ATTRIBUTE. 7-257
EXTRACT CPU TIME 7-260
EXTRACT PSW 7-261
EXTRACT TRANSACTION NESTING DEPTH. 7-261
FIND LEFTMOST ONE 7-262
INSERT CHARACTER. 7-262
INSERT CHARACTERS UNDER MASK . . . 7-262
INSERT IMMEDIATE 7-263
INSERT PROGRAM MASK 7-264
LOAD 7-264
LOAD IMMEDIATE 7-264
LOAD RELATIVE LONG 7-264
LOAD ACCESS MULTIPLE. 7-265
LOAD ADDRESS. 7-266
LOAD ADDRESS EXTENDED 7-266
LOAD ADDRESS RELATIVE LONG 7-267
LOAD AND ADD 7-268
LOAD AND ADD LOGICAL 7-268
LOAD AND AND 7-269
LOAD AND EXCLUSIVE OR 7-269
LOAD AND OR 7-270
LOAD AND TEST. 7-270
LOAD AND TRAP 7-271
LOAD AND ZERO RIGHTMOST BYTE. . . . 7-271
LOAD BYTE. 7-272
LOAD BYTE HIGH. 7-272
LOAD COMPLEMENT. 7-272
LOAD COUNT TO BLOCK BOUNDARY . . . 7-273
LOAD GUARDED 7-274
LOAD LOGICAL AND SHIFT GUARDED . . 7-274
LOAD GUARDED STORAGE CONTROLS. 7-275
LOAD HALFWORD 7-276
LOAD HALFWORD IMMEDIATE 7-276
LOAD HALFWORD RELATIVE LONG 7-276
LOAD HALFWORD HIGH. 7-277
LOAD HALFWORD IMMEDIATE ON CONDITION 7-277
LOAD HALFWORD HIGH IMMEDIATE ON CONDITION 7-277
LOAD HIGH. 7-278
LOAD HIGH AND TRAP 7-278
LOAD LOGICAL 7-278
LOAD LOGICAL RELATIVE LONG. 7-278
LOAD LOGICAL AND TRAP. 7-279
LOAD LOGICAL AND ZERO RIGHTMOST BYTE. 7-279
LOAD LOGICAL CHARACTER. 7-279
LOAD LOGICAL CHARACTER HIGH. 7-280
LOAD LOGICAL HALFWORD. 7-280
LOAD LOGICAL HALFWORD RELATIVE LONG 7-280
LOAD LOGICAL HALFWORD HIGH. 7-281
LOAD LOGICAL IMMEDIATE 7-281
LOAD LOGICAL THIRTY ONE BITS 7-281
LOAD LOGICAL THIRTY ONE BITS AND TRAP. 7-282
LOAD MULTIPLE 7-282
LOAD MULTIPLE DISJOINT. 7-283
LOAD MULTIPLE HIGH 7-283
LOAD NEGATIVE 7-283
LOAD ON CONDITION 7-284
LOAD HIGH ON CONDITION 7-284
LOAD PAIR DISJOINT 7-285
LOAD PAIR FROM QUADWORD. 7-286
LOAD POSITIVE 7-287
LOAD REVERSED 7-287
MONITOR CALL 7-288
MOVE 7-289
MOVE INVERSE 7-290
MOVE LONG. 7-290
MOVE LONG EXTENDED 7-294
MOVE LONG UNICODE 7-297
MOVE NUMERICS 7-301
MOVE STRING 7-301
MOVE WITH OFFSET. 7-302
MOVE ZONES 7-303
MULTIPLY. 7-304
MULTIPLY HALFWORD 7-305
MULTIPLY HALFWORD IMMEDIATE 7-305
MULTIPLY LOGICAL 7-306
MULTIPLY SINGLE. 7-307
MULTIPLY SINGLE IMMEDIATE 7-307
NEXT INSTRUCTION ACCESS INTENT . . .7-308
NONTRANSACTIONAL STORE. 7-310
OR 7-310
OR IMMEDIATE 7-312

PACK 7-312
PACK ASCII 7-313
PACK UNICODE 7-314
PERFORM CRYPTOGRAPHIC COMPUTATION 7-315
PERFORM LOCKED OPERATION 7-331
PERFORM PROCESSOR ASSIST 7-345
PERFORM RANDOM NUMBER OPERATION 7-346
POPULATION COUNT 7-359
PREFETCH DATA 7-359
PREFETCH DATA RELATIVE LONG 7-359
ROTATE LEFT SINGLE LOGICAL 7-361
ROTATE THEN AND SELECTED BITS . . . 7-361
ROTATE THEN EXCLUSIVE OR SELECTED BITS 7-362
ROTATE THEN OR SELECTED BITS 7-362
ROTATE THEN INSERT SELECTED BITS 7-363
ROTATE THEN INSERT SELECTED BITS HIGH 7-364
ROTATE THEN INSERT SELECTED BITS LOW 7-365
SEARCH STRING 7-366
SEARCH STRING UNICODE 7-368
SET ACCESS 7-370
SET ADDRESSING MODE 7-370
SET PROGRAM MASK 7-371
SHIFT LEFT DOUBLE 7-371
SHIFT LEFT DOUBLE LOGICAL 7-372
SHIFT LEFT SINGLE 7-372
SHIFT LEFT SINGLE LOGICAL 7-373
SHIFT RIGHT DOUBLE 7-374
SHIFT RIGHT DOUBLE LOGICAL 7-374
SHIFT RIGHT SINGLE 7-375
SHIFT RIGHT SINGLE LOGICAL 7-376
STORE 7-376
STORE RELATIVE LONG 7-377
STORE ACCESS MULTIPLE 7-377
STORE CHARACTER 7-378
STORE CHARACTER HIGH 7-378
STORE CHARACTERS UNDER MASK . . . 7-378
STORE CLOCK 7-379
STORE CLOCK FAST 7-379
STORE CLOCK EXTENDED 7-380
STORE FACILITY LIST EXTENDED 7-382
STORE GUARDED STORAGE CONTROLS 7-383
STORE HALFWORD 7-383
STORE HALFWORD RELATIVE LONG . . . 7-384
STORE HALFWORD HIGH 7-384
STORE HIGH 7-384
STORE MULTIPLE 7-385
STORE MULTIPLE HIGH 7-385
STORE ON CONDITION 7-385
STORE HIGH ON CONDITION 7-386
STORE PAIR TO QUADWORD 7-386
STORE REVERSED 7-387
SUBTRACT 7-387
SUBTRACT HALFWORD 7-388
SUBTRACT HIGH 7-389
SUBTRACT LOGICAL 7-389
SUBTRACT LOGICAL IMMEDIATE 7-390
SUBTRACT LOGICAL HIGH 7-391
SUBTRACT LOGICAL WITH BORROW . . . 7-391
SUPERVISOR CALL 7-392
TEST ADDRESSING MODE 7-392
TEST AND SET 7-392
TEST UNDER MASK (TEST UNDER MASK HIGH, TEST UNDER MASK LOW) 7-393
TRANSACTION ABORT 7-394
TRANSACTION BEGIN (TBEGIN) 7-395
TRANSACTION BEGIN (TBEGINC) 7-399
TRANSACTION END 7-401
TRANSLATE 7-402
TRANSLATE AND TEST 7-403
TRANSLATE AND TEST EXTENDED 7-403
TRANSLATE AND TEST REVERSE EXTENDED 7-404
TRANSLATE AND TEST REVERSE 7-408
TRANSLATE EXTENDED 7-409
TRANSLATE ONE TO ONE 7-411
TRANSLATE ONE TO TWO 7-411
TRANSLATE TWO TO ONE 7-411
TRANSLATE TWO TO TWO 7-411
UNPACK 7-416
UNPACK ASCII 7-417
UNPACK UNICODE 7-418
UPDATE TREE 7-419
Protection of Cryptographic Keys 7-424
Protection of DES Keys 7-425
Protection of AES Keys 7-427

This chapter includes most of the unprivileged instructions described in this publication. Other unprivileged instructions are described in chapters 8-9 and 18-25.

Data Format

The general instructions treat data as being of four types: signed binary integers, unsigned binary integers, unstructured logical data, and decimal data.

Data is treated as decimal by the conversion, packing, and unpacking instructions. Decimal data is described in Chapter 8, “Decimal Instructions.”

The general instructions manipulate data which resides in general registers or in storage or is introduced from the instruction stream. Some general instructions operate on data which resides in the PSW or the TOD clock.

In a storage-and-storage operation the operand fields may be defined in such a way that they overlap. The effect of this overlap depends upon the operation. When the operands remain unchanged, as in COMPARE or TRANSLATE AND TEST, overlapping does not affect the execution of the operation. For instructions such as MOVE and TRANSLATE, one operand is replaced by new data, and the execution of the operation may be affected by the amount of overlap and the manner in which data is fetched or stored. For purposes of evaluating the effect of overlapped operands, data is considered to be handled one eight-bit byte at a time. Special rules apply to the operands of MOVE LONG and MOVE INVERSE. See “Interlocks within a Single Instruction” on page 5-116 for how overlap is detected in the access-register mode.

Binary-Integer Representation

Binary integers are treated as signed or unsigned.

In an unsigned binary integer, all bits are used to express the absolute value of the number. When two unsigned binary integers of different lengths are added, the shorter number is considered to be extended on the left with zeros.

In some operations, the result is achieved by the use of the one’s complement of the number. The one’s complement of a number is obtained by inverting each bit of the number, including the sign.

For signed binary integers, the leftmost bit represents the sign, which is followed by the numeric field. Positive numbers are represented in true binary notation with the sign bit set to zero. When the value is zero, all bits are zeros, including the sign bit. Negative numbers are represented in two’s-complement binary notation with a one in the sign-bit position.

Specifically, a negative number is represented by the two’s complement of the positive number of the same absolute value. The two’s complement of a number is obtained by forming the one’s complement of the number, adding a value of one in the rightmost bit position, allowing a carry into the sign position, and ignoring any carry out of the sign position.

This number representation can be considered the rightmost portion of an infinitely long representation of the number. When the number is positive, all bits to the left of the most significant bit of the number are zeros. When the number is negative, these bits are ones. Therefore, when a signed operand must be extended with bits on the left, the extension is achieved by setting these bits equal to the sign bit of the operand.

The notation for signed binary integers does not include a negative zero. It has a number range in which, for a given length, the set of negative nonzero numbers is one larger than the set of positive nonzero numbers. The maximum positive number consists of a sign bit of zero followed by all ones, whereas the maximum negative number (the negative number with the greatest absolute value) consists of a sign bit of one followed by all zeros.

A signed binary integer of either sign, except for zero and the maximum negative number, can be changed to a number of the same magnitude but opposite sign by forming its two’s complement. Forming the two’s complement of a number is equivalent to subtracting the number from zero. The two’s complement of zero is zero.

The two’s complement of the maximum negative number cannot be represented in the same number of bits. When an operation, such as LOAD COMPLEMENT, attempts to produce the two’s complement of the maximum negative number, the result is the maximum negative number, and a fixed-point-overflow exception is recognized. An overflow does not result, however, when the maximum negative number is complemented as an intermediate result but the final result is within the representable range. An example of this case is a subtraction of the maximum negative number from -1. The product of two maximum negative numbers of a given length is representable as a positive number of double that length.

In discussions of signed binary integers in this publication, a signed binary integer includes the sign bit. Thus, the expression “32-bit signed binary integer”

denotes an integer with 31 numeric bits and a sign bit, and the expression "64-bit signed binary integer" denotes an integer with 63 numeric bits and a sign bit.

In an arithmetic operation, a carry out of the numeric field of a signed binary integer is carried into the sign bit. However, in algebraic left-shifting, the sign bit does not change even if significant numeric bits are shifted out.

Programming Notes:

1. An alternate way of forming the two's complement of a signed binary integer is to invert all bits to the left of the rightmost one bit, leaving the rightmost one bit and all zero bits to the right of it unchanged.
2. The numeric bits of a signed binary integer may be considered to represent a positive value, with the sign representing a value of either zero or the maximum negative number.

Binary Arithmetic

Many of the instructions that perform a register-and-storage or register-and-register binary-arithmetic operation are provided in sets of three instructions corresponding to three different combinations of operand lengths. These three instructions have the same name but different operation codes and mnemonics. For example, ADD (A) operates on 32-bit operands and produces a 32-bit result, ADD (AG) operates on 64-bit operands and produces a 64-bit result, and ADD (AGF) operates on a 64-bit operand and a 32-bit operand and produces a 64-bit result. The letter "G" alone in the mnemonic indicates a completely 64-bit operation, and the letters "GF" indicate a 32-to-64-bit operation.

In a 32-to-64-bit operation, the intermediate result is 64 bits. LOAD COMPLEMENT (LCGFR) forms the two's complement of the maximum negative 32-bit number without recognizing overflow.

Except for the instructions of the high-word facility, a 32-bit operand in a general register is in bit positions 32-63 of the register. In an operation on the operand, such as by ADD (A), bits 0-31 of the register are unused and remain unchanged. A 64-bit operand in a general register is in bit positions 0-63 of the register, and all of the bits participate in an operation on the operand, such as by ADD (AG). However, some instructions, which do not have "G" in their mnemonics, use a 64-bit operand of which the leftmost 32 bits are in bit positions 32-63 of the even register of an even-odd general-register pair, and the rightmost 32 bits are in bit positions 32-63 of the odd register of the pair.

The bits of a 32-bit operand in storage are numbered 0-31. When the operand is in bit positions 32-63 of a general register, the bits are numbered 32-63.

Signed Binary Arithmetic

Addition and Subtraction

Addition of signed binary integers is performed by adding all bits of each operand, including the sign bits. When one of the operands is shorter, the shorter operand is considered to be extended on the left to the length of the longer operand by propagating the sign-bit value.

For a 32-bit signed binary integer in a general register, the sign bit is bit 32 of the register. For a 64-bit signed binary integer in a general register, the sign bit is bit 0 of the register.

Subtraction is performed by adding the one's complement of the second operand and a value of one to the first operand.

Fixed-Point Overflow

A fixed-point-overflow condition exists for signed binary addition or subtraction when the carry out of the sign-bit position and the carry out of the leftmost numeric bit position disagree. Detection of an overflow does not affect the result produced by the addition. In mathematical terms, signed addition and subtraction produce a fixed-point overflow when the result is outside the range of representation for signed binary integers. Specifically, for ADD, LOAD AND ADD, LOAD COMPLEMENT, and LOAD POSITIVE instructions which operate on 32-bit signed binary integers, there is an overflow when the proper result would be greater than or equal to $+2^{31}$ or less than -2^{31}. The actual result placed in the general register after an overflow differs from the proper result by 2^{32}. Similarly, for ADD, LOAD AND ADD, LOAD COMPLEMENT, and LOAD POSITIVE instructions which operate on 64-bit signed binary integers, there is an overflow when the proper result would be

greater than or equal to $+2^{63}$ or less than -2^{63}, and the actual result placed in the general register after an overflow differs from the proper result by 2^{64}. ADD (AGF) and SUBTRACT (SGF) have the same 64-bit result and overflow rules as ADD (AG) and SUBTRACT (SG).

The instructions SHIFT LEFT SINGLE and SHIFT LEFT DOUBLE produce an overflow when the result is outside the range of representation for signed binary integers. The actual result differs from that for addition and subtraction in that the sign of the result remains the same as the original sign.

For MULTIPLY SINGLE (MSC and MSRKC), there is an overflow result when the proper result would be greater than or equal to $+2^{31}$ or less than -2^{31}. Similarly, for MULTIPLY SINGLE (MSGC and MSGRKC), there is an overflow result when the proper result would be greater than or equal to $+2^{63}$ or less than -2^{63}. The actual result placed in the general register after an overflow is the rightmost 32 bits (MSC, MSRKC) or 64 bits (MSGC, MSGRKC) of the product.

When an overflow result occurs, condition code 3 is set. When the fixed-point overflow mask (bit 20 of the PSW) is one, a fixed-point-overflow program-exception condition is recognized.

Unsigned Binary Arithmetic

Addition of unsigned binary integers is performed by adding all bits of each operand. Subtraction is performed by adding the one's complement of the second operand (the subtractor) and a value of one to the first operand (the subtrahend). In any case, when one of the operands is shorter, the shorter operand is considered to be extended on the left with zeros. During subtraction, this extension applies before an operand is complemented, and it applies to the value of one.

Unsigned binary arithmetic is used in address arithmetic for adding the X, B, and D fields. (See "Address Generation" on page 5-10.) It is also used to obtain the addresses of the function bytes in TRANSLATE and TRANSLATE AND TEST. Furthermore, unsigned binary arithmetic is used on 32-bit or 64-bit unsigned binary integers by ADD LOGICAL, ADD LOGICAL WITH CARRY, DIVIDE LOGICAL, MULTIPLY LOGICAL, SUBTRACT LOGICAL, and SUBTRACT LOGICAL WITH BORROW.

Given the same length operands, ADD (A, AG, AGF) and ADD LOGICAL (AL, ALG, ALGF) produce the same 32-bit or 64-bit result. The instructions differ only in the interpretation of this result. ADD interprets the result as a signed binary integer and inspects it for sign, magnitude, and overflow to set the condition code accordingly. ADD LOGICAL interprets the result as an unsigned binary integer and sets the condition code according to whether the result is zero and whether there was a carry out of bit position 32, for a 32-bit integer, or out of bit position 0 for a 64-bit integer. Such a carry is not considered an overflow, and no program interruption for overflow can occur for ADD LOGICAL.

SUBTRACT LOGICAL differs from ADD LOGICAL in that the one's complement of the second operand and a value of one are added to the first operand.

For ADD LOGICAL WITH CARRY, a carry from a previous operation is represented by a one value of bit 18 of the current PSW. Bit 18 is the leftmost bit of the two-bit condition code in the PSW. For SUBTRACT LOGICAL WITH BORROW, a borrow from a previous operation is represented by a zero value of bit 18. A borrow is equivalent to the absence of a carry.

For VECTOR ADD WITH CARRY, a carry from a previous operation is represented by a one in the rightmost bit of each element of a vector register. For VECTOR SUBTRACT WITH BORROW INDICATION, an indication of borrow from a previous operation is represented by a one in the rightmost bit of each element of a vector register. The indication of borrow is equivalent to a carry. The vector integer instructions may be found in Chapter 22.

Programming Notes:

1. Logical addition and subtraction may be used to perform arithmetic on multiple-precision binary-integer operands. Thus, for multiple-precision addition, ADD LOGICAL can be used to add the lowest-order corresponding parts of the operands, and ADD LOGICAL WITH CARRY can be used to add the other corresponding parts of the operands, moving from right to left in the operands. If the multiple-precision operands are signed, ADD should be used on the highest-order parts. The condition code then indicates any overflow or the proper sign and magnitude of the entire result; an overflow is also indicated by a program interruption for fixed-point overflow if

allowed by the program mask. When ADD is used, a value of one must be added to the sum of the highest-order parts if the condition code indicated there was a carry from the addition of the next-lower parts.

2. Another use for ADD LOGICAL is to increment values representing binary counters, which are allowed to wrap around from all ones to all zeros without indicating overflow.

Signed and Logical Comparison

Comparison operations determine whether two operands are equal or not and, for most operations, which of two unequal operands is the greater (high). Signed-binary-comparison operations are provided which treat the operands as signed binary integers, and logical-comparison operations are provided which treat the operands as unsigned binary integers or as unstructured data.

COMPARE (C, CG, CGF) and COMPARE HALFWORD are signed-binary-comparison operations. These instructions are equivalent to SUBTRACT (S, SG, SGF) and SUBTRACT HALFWORD without replacing either operand, the resulting difference being used only to set the condition code. The operations permit comparison of numbers of opposite sign which differ by 2^{63} or more. Thus, unlike SUBTRACT, COMPARE cannot cause overflow.

Logical comparison of two operands is performed byte by byte, in a left-to-right sequence. The operands are equal when all their bytes are equal. When the operands are unequal, the comparison result is determined by a left-to-right comparison of corresponding bit positions in the first unequal pair of bytes: the zero bit in the first unequal pair of bits indicates the low operand, and the one bit the high operand. Since the remaining bit and byte positions do not change the comparison, it is not necessary to continue comparing unequal operands beyond the first unequal bit pair.

Instructions

The general instructions and their mnemonics, formats, and operation codes are listed in Figure 7-1 on page 7-13. The figure also indicates which instructions are new in z/Architecture as compared to ESA/390, when the condition code is set, the instruction fields that designate access registers, and the exceptional conditions in operand designations, data, or results that cause a program interruption.

The instructions that are new in z/Architecture are indicated in Figure 7-1 by “N.” A few of the instructions that are new in z/Architecture have also been added to ESA/390, and these are indicated by “N3.”

When the operands of an instruction are 32-bit operands, the mnemonic for the instruction normally does not include a letter indicating the operand length; however some mnemonics for instructions with 32-bit operands contain the letter “F”. If there is an instruction with the same name but with 64-bit operands, its mnemonic includes the letter “G.” If there is an instruction with the same name but with a 64-bit first operand and a 32-bit second operand, its mnemonic includes the letters “GF.” Certain instruction mnemonics indicate the length of the first and second operands with combinations of G (64-bit), F (32-bit), or H (16-bit), although the F is usually omitted for the first operand.

In Figure 7-1, when there is an instruction with 32-bit operands and other instructions with the same name but with “G” or “GF” added in their mnemonics, the first instruction has “(32)” after its name, and the other instructions have “(64)” or “(64←32),” respectively, after their names. Some 32-bit operand-length instructions do not have 64-bit operand-length counterparts, and they do not have “(32)” after their names. However, all instructions for multiplication or division are marked to show, or approximately show, operand lengths.

A detailed definition of instruction formats, operand designation and length, and address generation is contained in “Instructions” on page 5-3. Exceptions to the general rules stated in that section are explicitly identified in the individual instruction descriptions.

Note: In the detailed descriptions of the individual instructions, the mnemonic and the symbolic operand designations for the assembler language are shown with each instruction. For LOAD AND TEST with 32-bit operands, for example, LTR is the mnemonic and R_1,R_2 the operand designation.

Certain fields of various instructions are considered to be optional, as indicated by the field being contained within brackets [ ] in the assembler syntax.

When an optional field is not specified, the assembler places zeros in that field of the instruction.

Programming Notes:

1. Trimodal addressing affects the general instructions only in the manner in which logical storage addresses are handled, except as follows.

   The instructions BRANCH AND LINK (BAL, BALR), BRANCH AND SAVE (BAS, BASR), BRANCH AND SAVE AND SET MODE, BRANCH AND SET MODE, BRANCH RELATIVE AND SAVE, and BRANCH RELATIVE AND SAVE LONG place information in bit positions 32-39 of general register R_1 as in ESA/390 in the 24-bit or 31-bit addressing mode or place address bits in those bit positions in the 64-bit addressing mode.

   The instruction BRANCH AND SAVE AND SET MODE places a zero in bit position 63 of general register R_1 in the 24-bit or 31-bit addressing mode or places a one in that bit position in the 64-bit addressing mode.

   The instruction BRANCH AND SET MODE leaves the contents of bit position 63 of general register R_1 unchanged in the 24-bit or 31-bit addressing mode or places a one in that bit position in the 64-bit addressing mode.

   The following instructions leave bits 0-31 of a general register unchanged in the 24-bit or 31-bit addressing mode but place or update address or length information in them in the 64-bit addressing mode. Also, the leftmost byte of the results in registers may be handled differently depending on whether the addressing mode is the 24-bit or the 31-bit mode.

   - BRANCH AND LINK (BAL, BALR)
   - BRANCH AND SAVE (BAS, BASR)
   - BRANCH AND SAVE AND SET MODE
   - BRANCH RELATIVE AND SAVE
   - BRANCH RELATIVE AND SAVE LONG
   - CHECKSUM
   - COMPARE AND FORM CODEWORD
   - COMPARE LOGICAL LONG
   - COMPARE LOGICAL LONG EXTENDED
   - COMPARE LOGICAL LONG UNICODE
   - COMPARE LOGICAL STRING
   - COMPARE UNTIL SUBSTRING EQUAL
   - COMPRESSION CALL
   - CONVERT UTF-16 TO UTF-32
   - CONVERT UTF-16 TO UTF-8
   - CONVERT UTF-32 TO UTF-16
   - CONVERT UTF-32 TO UTF-8
   - CONVERT UTF-8 TO UTF-16
   - CONVERT UTF-8 TO UTF-32
   - LOAD ADDRESS
   - LOAD ADDRESS EXTENDED
   - LOAD ADDRESS RELATIVE LONG
   - MOVE LONG
   - MOVE LONG EXTENDED
   - MOVE LONG UNICODE
   - MOVE STRING
   - SEARCH STRING
   - TRANSLATE EXTENDED
   - TRANSLATE AND TEST
   - TRANSLATE AND TEST EXTENDED
   - TRANSLATE AND TEST REVERSE
   - TRANSLATE AND TEST REVERSE EXTENDED
   - TRANSLATE ONE TO ONE
   - TRANSLATE ONE TO TWO
   - TRANSLATE TWO TO ONE
   - TRANSLATE TWO TO TWO
   - UPDATE TREE

   The instructions in the preceding list are sometimes called modal instructions.

2. Bits 0-31 of general registers are changed by two types of instructions. The first type is a modal instruction, as listed in the preceding note, when the instruction is executed in the 64-bit addressing mode. The second type is an instruction having, independent of the addressing mode, either a 64-bit result operand in a single general register or a 128-bit result operand in an even-odd general-register pair.

   Most of the instructions of the second type are indicated by a “G,” either alone or in “GF,” in their mnemonics. The other instructions that change or may change bits 0-31 of a general register regardless of the current addressing mode are:

   - AND IMMEDIATE (NIHF, NIHH, NIHL only)
   - EXCLUSIVE OR IMMEDIATE (XIHF only)
   - INSERT CHARACTERS UNDER MASK (ICMH only)
   - INSERT IMMEDIATE (IIHF, IIHH, IIHL only)
   - LOAD LOGICAL IMMEDIATE (LLIHF, LLIHH, LLIHL only)
   - LOAD MULTIPLE DISJOINT
   - LOAD MULTIPLE HIGH
   - LOAD PAIR FROM QUADWORD
   - OR IMMEDIATE (OIHF, OIHH, OIHL only)

All of the instructions of the second type are sometimes referred to as “G-type” instructions.

If a program is not executed in the 64-bit addressing mode and does not contain a G-type instruction, it cannot change bits 0-31 of any general register.

3. It is not intended or expected that old programs not containing G-type instructions will be able to be executed successfully in the 64-bit addressing mode. However, this may be possible, particularly if, by programming convention, bits 0-31 of the general registers are always all zeros when an old program is given control.

4. The following additional general instructions are available when the extended-translation facility 2 is installed:

   - COMPARE LOGICAL LONG UNICODE
   - MOVE LONG UNICODE
   - PACK ASCII
   - PACK UNICODE
   - TRANSLATE ONE TO ONE
   - TRANSLATE ONE TO TWO
   - TRANSLATE TWO TO ONE
   - TRANSLATE TWO TO TWO
   - UNPACK ASCII
   - UNPACK UNICODE

5. The long-displacement facility uses new instruction formats, named RSY, RXY, and SIY, to provide 20-bit signed displacements. In connection with the long-displacement facility, all previously existing general instructions of the RSE or RXE format are changed to be of format RSY or RXY, respectively, where the new formats differ from the old by using a previously unused byte, now named DH, in the instructions. When the long-displacement facility is installed, the displacement for an instruction operand address is formed by appending DH on the left of the previous displacement field, now named DL, of the instruction. When the long-displacement facility is not installed, eight zero bits are appended on the left of DL, and DH is ignored.

   The following additional general instruction is available when the long-displacement facility is installed.

   - LOAD BYTE

   The following additional RSY-format versions of general instructions are available when the long-displacement facility is installed.

   - COMPARE AND SWAP
   - COMPARE DOUBLE AND SWAP
   - COMPARE LOGICAL CHARACTERS UNDER MASK
   - LOAD ACCESS MULTIPLE
   - LOAD MULTIPLE
   - STORE ACCESS MULTIPLE
   - STORE CHARACTERS UNDER MASK
   - STORE MULTIPLE

   The following additional RXY-format versions of general instructions are available when the long-displacement facility is installed.

   - ADD
   - ADD HALFWORD
   - ADD LOGICAL
   - AND
   - COMPARE
   - COMPARE HALFWORD
   - COMPARE LOGICAL
   - CONVERT TO BINARY
   - CONVERT TO DECIMAL
   - EXCLUSIVE OR
   - INSERT CHARACTER
   - INSERT CHARACTER UNDER MASK
   - LOAD
   - LOAD ADDRESS
   - LOAD HALFWORD
   - MULTIPLY SINGLE
   - OR
   - STORE
   - STORE CHARACTER
   - STORE HALFWORD
   - SUBTRACT
   - SUBTRACT HALFWORD
   - SUBTRACT LOGICAL

   The following additional SIY-format versions of general instructions are available when the long-displacement facility is installed.

   - AND
   - COMPARE LOGICAL
   - EXCLUSIVE OR
   - MOVE
   - OR
   - TEST UNDER MASK

6. The following additional general instructions are available when the message-security assist is installed:

- CIPHER MESSAGE
- CIPHER MESSAGE WITH CHAINING
- COMPUTE INTERMEDIATE MESSAGE DIGEST
- COMPUTE LAST MESSAGE DIGEST
- COMPUTE MESSAGE AUTHENTICATION CODE

7. The following additional general instructions are available when the extended-translation facility 3 is installed:

   - CONVERT UTF-16 TO UTF-32
   - CONVERT UTF-32 TO UTF-16
   - CONVERT UTF-32 TO UTF-8
   - CONVERT UTF-8 TO UTF-32
   - SEARCH STRING UNICODE
   - TRANSLATE AND TEST REVERSE

   Additionally, CONVERT UNICODE TO UTF-8 (CUUTF) and CONVERT UTF-8 TO UNICODE (CUTFU) are renamed to CONVERT UTF-16 TO UTF-8 (CU21) and CONVERT UTF-8 TO UTF-16 (CU12), respectively, in order to conform to the names used by this facility. The old names continue to be recognized.

8. The following additional general instructions are available when the extended-immediate facility is installed:

   - ADD IMMEDIATE (AFI, AGFI)
   - ADD LOGICAL IMMEDIATE (ALFI, ALGFI)
   - AND IMMEDIATE (NIHF, NILF)
   - COMPARE IMMEDIATE (CFI, CGFI)
   - COMPARE LOGICAL IMMEDIATE (CLFI, CLGFI)
   - EXCLUSIVE OR IMMEDIATE (XIHF, XILF)
   - FIND LEFTMOST ONE (FLOGR)
   - INSERT IMMEDIATE (IIHF, IILF)
   - LOAD AND TEST (LT, LTG)
   - LOAD BYTE (LBR, LGBR)
   - LOAD HALFWORD (LHR, LGHR)
   - LOAD IMMEDIATE (LGFI)
   - LOAD LOGICAL CHARACTER (LLC, LLCR, LLGCR)
   - LOAD LOGICAL HALFWORD (LLGHR, LLH, LLHR)
   - LOAD LOGICAL IMMEDIATE (LLIHF, LLILF)
   - OR IMMEDIATE (OIHF, OILF)
   - SUBTRACT LOGICAL IMMEDIATE (SLFI, SLGFI)

   The instructions that are part of the extended-immediate facility are indicated in Figure 7-1 by "EI."

9. The STORE CLOCK FAST general instruction is available when the store-clock-fast facility is installed.

10. The STORE FACILITY LIST EXTENDED general instruction is available when the store-facility-list-extended facility is installed.

11. When the ETF2-enhancement facility is installed, new functions are provided for the TRANSLATE ONE TO ONE, TRANSLATE ONE TO TWO, TRANSLATE TWO TO ONE, and TRANSLATE TWO TO TWO instructions.

12. When the ETF3-enhancement facility is installed, improved well-formedness checking is provided for the CONVERT UTF-16 TO UTF-32, CONVERT UTF-16 TO UTF-8, CONVERT UTF-8 TO UTF-16, and CONVERT UTF-8 TO UTF-32 instructions.

13. The COMPARE AND SWAP AND STORE general instruction is available if the compare-and-swap-and-store facility is installed.

14. The EXTRACT CPU TIME general instruction is available if the extract-CPU-time facility is installed.

15. The following additional general instructions are available when the parsing-enhancement facility is installed:

    - TRANSLATE AND TEST EXTENDED
    - TRANSLATE AND TEST REVERSE EXTENDED

16. The EXECUTE RELATIVE LONG general instruction is available when the execute-extension facility is installed.

17. The following additional general instructions are available when the general-instructions-extension facility is installed:

    - ADD IMMEDIATE (AGSI, ASI)
    - ADD LOGICAL WITH SIGNED IMMEDIATE (ALGSI, ALSI)
    - COMPARE AND BRANCH (CGRB, CRB)
    - COMPARE AND BRANCH RELATIVE (CGRJ, CRJ)
    - COMPARE AND TRAP (CGRT, CRT)
    - COMPARE HALFWORD (CGH)
    - COMPARE HALFWORD IMMEDIATE (CGHSI, CHHSI, CHSI)

- COMPARE HALFWORD RELATIVE LONG (CGHRL, CHRL)
- COMPARE IMMEDIATE AND BRANCH (CGIB, CIB)
- COMPARE IMMEDIATE AND BRANCH RELATIVE (CGIJ, CIJ)
- COMPARE IMMEDIATE AND TRAP (CIT, CGIT)
- COMPARE LOGICAL AND BRANCH (CLGRB, CLRB)
- COMPARE LOGICAL AND BRANCH RELATIVE (CLGRJ, CLRJ)
- COMPARE LOGICAL AND TRAP (CLGRT, CLRT)
- COMPARE LOGICAL IMMEDIATE (CLFHSI, CLGHSI, CLHHSI)
- COMPARE LOGICAL IMMEDIATE AND BRANCH (CLIB, CLGIB)
- COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (CLGIJ, CLIJ)
- COMPARE LOGICAL IMMEDIATE AND TRAP (CLFIT, CLGIT)
- COMPARE LOGICAL RELATIVE LONG (CLGFRL, CLGHRL, CLGRL, CLHRL, CLRL)
- COMPARE RELATIVE LONG (CGFRL, CGRL, CRL)
- EXTRACT CPU ATTRIBUTE (ECAG)
- LOAD ADDRESS EXTENDED (LAEY)
- LOAD AND TEST (LTGF)
- LOAD HALFWORD RELATIVE LONG (LGHRL, LHRL)
- LOAD LOGICAL HALFWORD RELATIVE LONG (LLGHRL, LLHRL)
- LOAD LOGICAL RELATIVE LONG (LLGFRL)
- LOAD RELATIVE LONG (LGFRL, LGRL, LRL)
- MOVE (MVHI, MVGHI, MVHHI)
- MULTIPLY (MFY)
- MULTIPLY HALFWORD (MHY)
- MULTIPLY SINGLE IMMEDIATE (MSFI, MSGFI)
- PREFETCH DATA (PFD)
- PREFETCH DATA RELATIVE LONG (PFDRL)
- ROTATE THEN AND SELECTED BITS (RNSBG)
- ROTATE THEN EXCLUSIVE OR SELECTED BITS (RXSBG)
- ROTATE THEN INSERT SELECTED BITS (RISBG)
- ROTATE THEN OR SELECTED BITS (ROSBG)
- STORE HALFWORD RELATIVE LONG (STHRL)
- STORE RELATIVE LONG (STGRL, STRL)

18. The following additional general instructions are available when the high-word facility is installed:

    - ADD HIGH (AHHHR, AHHLR)
    - ADD IMMEDIATE HIGH (AIH)
    - ADD LOGICAL HIGH (ALHHHR, ALHHLR)
    - ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (ALSIH, ALSIHN)
    - BRANCH RELATIVE ON COUNT HIGH (BRCTH)
    - COMPARE HIGH (CHHR, CHLR, CHF)
    - COMPARE IMMEDIATE HIGH (CIH)
    - COMPARE LOGICAL HIGH (CLHHR, CLHLR, CLHF)
    - COMPARE LOGICAL IMMEDIATE HIGH (CLIH)
    - LOAD BYTE HIGH (LBH)
    - LOAD HALFWORD HIGH (LHH)
    - LOAD HIGH (LFH)
    - LOAD LOGICAL CHARACTER HIGH (LLCH)
    - LOAD LOGICAL HALFWORD HIGH (LLHH)
    - ROTATE THEN INSERT SELECTED BITS HIGH (RISBHG)
    - ROTATE THEN INSERT SELECTED BITS LOW (RISBLG)
    - STORE CHARACTER HIGH (STCH)
    - STORE HALFWORD HIGH (STHH)
    - STORE HIGH (STFH)
    - SUBTRACT HIGH (SHHHR, SHHLR)
    - SUBTRACT LOGICAL HIGH (SLHHHR, SLHHLR)

19. The following additional general instructions are available when the interlocked-access facility 1 is installed:

    - LOAD AND ADD (LAA, LAAG)
    - LOAD AND ADD LOGICAL (LAAL, LAALG)
    - LOAD AND AND (LAN, LANG)
    - LOAD AND EXCLUSIVE OR (LAX, LAXG)
    - LOAD AND OR (LAO, LAOG)
    - LOAD PAIR DISJOINT (LPD, LPDG)

20. The following additional general instructions are available when the load/store-on-condition facility 1 is installed:

    - LOAD ON CONDITION (LOC, LOCG, LOCGR, LOCR)
    - STORE ON CONDITION (STOC, STOCG)

21. The following additional general instructions are available when the distinct-operands facility is installed:

    - ADD (ARK, AGRK)
    - ADD IMMEDIATE (AHIK, AGHIK)
    - ADD LOGICAL (ALRK, ALGRK)
    - ADD LOGICAL WITH SIGNED IMMEDIATE (ALHSIK, ALGHSIK)
    - AND (NRK, NGRK)
    - EXCLUSIVE OR (XRK, XGRK)
    - OR (ORK, OGRK)
    - SHIFT LEFT SINGLE (SLAK)
    - SHIFT LEFT SINGLE LOGICAL (SLLK)
    - SHIFT RIGHT SINGLE (SRAK)
    - SHIFT RIGHT SINGLE LOGICAL (SRLK)
    - SUBTRACT (SRK, SGRK)
    - SUBTRACT LOGICAL (SLRK, SLGRK)

22. The POPULATION COUNT general instruction is available when the population-count facility is installed.

23. The following additional general instructions are available when the message-security-assist extension 4 is installed:

    - CIPHER MESSAGE WITH CIPHER FEEDBACK
    - CIPHER MESSAGE WITH COUNTER
    - CIPHER MESSAGE WITH OUTPUT FEEDBACK
    - PERFORM CRYPTOGRAPHIC COMPUTATION

24. The following additional general instructions are available when the execution-hint facility is installed:

    - BRANCH PREDICTION PRELOAD (BPP)
    - BRANCH PREDICTION RELATIVE PRELOAD (BPRP)
    - NEXT INSTRUCTION ACCESS INTENT (NIAI)

25. The following additional general instructions are available when the load-and-trap facility is installed:

    - LOAD AND TRAP (LAT and LGAT)
    - LOAD HIGH AND TRAP (LFHAT)
    - LOAD LOGICAL AND TRAP (LLGFAT)
    - LOAD LOGICAL THIRTY ONE BITS AND TRAP (LLGTAT)

26. The following additional general instructions are available when the miscellaneous-instruction-extensions facility 1 is installed:

    - COMPARE LOGICAL AND TRAP (CLT, CLGT)
    - ROTATE THEN INSERT SELECTED BITS (RISBGN)

27. The following additional general instructions are available when the load-and-zero-rightmost-byte facility is installed:

    - LOAD AND ZERO RIGHTMOST BYTE (LZRF, LZRG)
    - LOAD LOGICAL AND ZERO RIGHTMOST BYTE (LLZRGF)

28. The following additional general instructions are available when the load/store-on-condition facility 2 is installed:

    - LOAD HALFWORD HIGH IMMEDIATE ON CONDITION (LOCHHI)
    - LOAD HALFWORD IMMEDIATE ON CONDITION (LOCHI, LOCGHI)
    - LOAD HIGH ON CONDITION (LOCFH, LOCFHR)
    - STORE HIGH ON CONDITION (STOCFH)

29. The PERFORM RANDOM NUMBER OPERATION general instruction is available when the message-security-assist extension 5 is installed:

30. The LOAD COUNT TO BLOCK BOUNDARY general instruction is available when the vector facility for z/Architecture is installed. Other instructions of the vector facility for z/Architecture are described in Chapters 21-24.

31. The following additional general instructions are available when the miscellaneous-instruction-extensions facility 2 is installed:

    - ADD HALFWORD (AGH)
    - BRANCH INDIRECT ON CONDITION
    - MULTIPLY (MG, MGRK)
    - MULTIPLY HALFWORD (MGH)
    - MULTIPLY SINGLE (MSC, MSGC, MSGRKC, MSRKC)
    - SUBTRACT HALFWORD (SGH)

32. The following additional general instructions are available when the guarded-storage facility is installed:

    - LOAD GUARDED (LGG)
    - LOAD GUARDED STORAGE CONTROLS (LGSC)
    - LOAD LOGICAL AND SHIFT GUARDED (LLGFSG)
    - STORE GUARDED STORAGE CONTROLS (STGSC)

33. The CIPHER MESSAGE WITH AUTHENTICATION general instruction is available when the message-security-assist extension 8 is installed.

Figure 7-1 lists the instructions, mnemonics, characteristics, and operation codes of the general instructions. In the Characteristics columns of this figure, a facility code associated with the instruction may be indicated in the third column (immediately to the left of the first vertical ruling in the Characteristics columns). If the facility code is blank, E2, MS, or N3, then the instruction is supported in the ESA/390-compatibility mode, although not all operands or functions of the instruction may be valid.

Name	Mnemonic	Characteristics								Opcode	Page
ADD (32)	A	RX-a	C		A	IF			B_2	5A	7-26
ADD (32)	AR	RR	C			IF				1A	7-26
ADD (32)	ARK	RRF-a	C	DO		IF				B9F8	7-26
ADD (32)	AY	RXY-a	C	LD	A	IF			B_2	E35A	7-27
ADD (64)	AG	RXY-a	C	N	A	IF			B_2	E308	7-27
ADD (64)	AGR	RRE	C	N		IF				B908	7-26
ADD (64)	AGRK	RRF-a	C	DO		IF				B9E8	7-26
ADD (64←32)	AGF	RXY-a	C	N	A	IF			B_2	E318	7-27
ADD (64←32)	AGFR	RRE	C	N		IF				B918	7-26
ADD HALFWORD (32←16)	AH	RX-a	C		A	IF			B_2	4A	7-28
ADD HALFWORD (32←16)	AHY	RXY-a	C	LD	A	IF			B_2	E37A	7-28
ADD HALFWORD (64←16)	AGH	RXY-a	C	MI2	A	IF			B_2	E338	7-28
ADD HALFWORD IMMEDIATE (32←16)	AHI	RI-a	C			IF				A7A	7-28
ADD HALFWORD IMMEDIATE (64←16)	AGHI	RI-a	C	N		IF				A7B	7-29
ADD HIGH (32)	AHHHR	RRF-a	C	HW		IF				B9C8	7-29
ADD HIGH (32)	AHHLR	RRF-a	C	HW		IF				B9D8	7-29
ADD IMMEDIATE (32)	AFI	RIL-a	C	EI		IF				C29	7-27
ADD IMMEDIATE (32←16)	AHIK	RIE-d	C	DO		IF				ECD8	7-27
ADD IMMEDIATE (32←8)	ASI	SIY	C	GE	A	IF £[1]		ST	B_1	EB6A	7-27
ADD IMMEDIATE (64←16)	AGHIK	RIE-d	C	DO		IF				ECD9	7-27
ADD IMMEDIATE (64←32)	AGFI	RIL-a	C	EI		IF				C28	7-27
ADD IMMEDIATE (64←8)	AGSI	SIY	C	GE	A	IF £[1]		ST	B_1	EB7A	7-27
ADD IMMEDIATE HIGH (32)	AIH	RIL-a	C	HW		IF				CC8	7-29
ADD LOGICAL (32)	AL	RX-a	C		A				B_2	5E	7-30
ADD LOGICAL (32)	ALR	RR	C							1E	7-30
ADD LOGICAL (32)	ALRK	RRF-a	C	DO						B9FA	7-30
ADD LOGICAL (32)	ALY	RXY-a	C	LD	A				B_2	E35E	7-30
ADD LOGICAL (64)	ALG	RXY-a	C	N	A				B_2	E30A	7-30
ADD LOGICAL (64)	ALGR	RRE	C	N						B90A	7-30
ADD LOGICAL (64)	ALGRK	RRF-a	C	DO						B9EA	7-30
ADD LOGICAL (64←32)	ALGF	RXY-a	C	N	A				B_2	E31A	7-30
ADD LOGICAL (64←32)	ALGFR	RRE	C	N						B91A	7-30
ADD LOGICAL HIGH (32)	ALHHHR	RRF-a	C	HW						B9CA	7-31
ADD LOGICAL HIGH (32)	ALHHLR	RRF-a	C	HW						B9DA	7-31
ADD LOGICAL IMMEDIATE (32)	ALFI	RIL-a	C	EI						C2B	7-30
ADD LOGICAL IMMEDIATE (64←32)	ALGFI	RIL-a	C	EI						C2A	7-30
ADD LOGICAL WITH CARRY (32)	ALC	RXY-a	C	N3	A				B_2	E398	7-31
ADD LOGICAL WITH CARRY (32)	ALCR	RRE	C	N3						B998	7-31
ADD LOGICAL WITH CARRY (64)	ALCG	RXY-a	C	N	A				B_2	E388	7-31

Figure 7-1. Summary of General Instructions (Part 1 of 14)

Name	Mnemonic	Characteristics								Opcode	Page
ADD LOGICAL WITH CARRY (64)	ALCGR	RRE	C	N						B988	7-31
ADD LOGICAL WITH SIGNED IMMEDIATE (32←16)	ALHSIK	RIE-d	C	DO						ECDA	7-32
ADD LOGICAL WITH SIGNED IMMEDIATE (32←8)	ALSI	SIY	C	GE	A	£[1]		ST	B_1	EB6E	7-32
ADD LOGICAL WITH SIGNED IMMEDIATE (64←16)	ALGHSIK	RIE-d	C	DO						ECDB	7-32
ADD LOGICAL WITH SIGNED IMMEDIATE (64←8)	ALGSI	SIY	C	GE	A	£[1]		ST	B_1	EB7E	7-32
ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (32)	ALSIH	RIL-a	C	HW						CCA	7-33
ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (32)	ALSIHN	RIL-a		HW						CCB	7-33
AND (32)	N	RX-a	C		A				B_2	54	7-33
AND (32)	NR	RR	C							14	7-33
AND (32)	NRK	RRF-a	C	DO						B9F4	7-33
AND (32)	NY	RXY-a	C	LD	A				B_2	E354	7-33
AND (64)	NG	RXY-a	C	N	A				B_2	E380	7-33
AND (64)	NGR	RRE	C	N						B980	7-33
AND (64)	NGRK	RRF-a	C	DO						B9E4	7-33
AND (character)	NC	SS-a	C		¤[9] A			ST	B_1 B_2	D4	7-34
AND (immediate)	NI	SI	C		A	£[2]		ST	B_1	94	7-33
AND (immediate)	NIY	SIY	C	LD	A	£[2]		ST	B_1	EB54	7-34
AND IMMEDIATE (high high)	NIHH	RI-a	C	N						A54	7-34
AND IMMEDIATE (high low)	NIHL	RI-a	C	N						A55	7-34
AND IMMEDIATE (high)	NIHF	RIL-a	C	EI						C0A	7-34
AND IMMEDIATE (low high)	NILH	RI-a	C	N						A56	7-35
AND IMMEDIATE (low low)	NILL	RI-a	C	N						A57	7-35
AND IMMEDIATE (low)	NILF	RIL-a	C	EI						C0B	7-35
BRANCH AND LINK	BAL	RX-a			¤[9]			B		45	7-35
BRANCH AND LINK	BALR	RR			¤[2,9]	T		B		05	7-35
BRANCH AND SAVE	BAS	RX-a			¤[9]			B		4D	7-36
BRANCH AND SAVE	BASR	RR			¤[2,9]	T		B		0D	7-36
BRANCH AND SAVE AND SET MODE	BASSM	RR			¤[2,3,9]	T		B		0C	7-37
BRANCH AND SET MODE	BSM	RR			¤[3,9]	T		B		0B	7-38
BRANCH INDIRECT ON CONDITION	BIC	RXY-b		MI2	¤[9] A			B	B_2	E347	7-39
BRANCH ON CONDITION	BC	RX-b			¤[9]			B		47	7-40
BRANCH ON CONDITION	BCR	RR			¤[9]	¢[1]		B		07	7-40
BRANCH ON COUNT (32)	BCT	RX-a			¤[9]			B		46	7-41
BRANCH ON COUNT (32)	BCTR	RR			¤[9]			B		06	7-41
BRANCH ON COUNT (64)	BCTG	RXY-a		N	¤[9]			B		E346	7-41
BRANCH ON COUNT (64)	BCTGR	RRE		N	¤[9]			B		B946	7-41
BRANCH ON INDEX HIGH (32)	BXH	RS-a			¤[9]			B		86	7-42
BRANCH ON INDEX HIGH (64)	BXHG	RSY-a		N	¤[9]			B		EB44	7-42
BRANCH ON INDEX LOW OR EQUAL (32)	BXLE	RS-a			¤[9]			B		87	7-42
BRANCH ON INDEX LOW OR EQUAL (64)	BXLEG	RSY-a		N	¤[9]			B		EB45	7-42
BRANCH PREDICTION PRELOAD	BPP	SMI		EH	¤[9]					C7	7-43
BRANCH PREDICTION RELATIVE PRELOAD	BPRP	MII		EH	¤[9]					C5	7-43
BRANCH RELATIVE AND SAVE	BRAS	RI-b			¤[9]			B		A75	7-46
BRANCH RELATIVE AND SAVE LONG	BRASL	RIL-b		N3	¤[9]			B		C05	7-46
BRANCH RELATIVE ON CONDITION	BRC	RI-c			¤[10]			B		A74	7-46
BRANCH RELATIVE ON CONDITION LONG	BRCL	RIL-c		N3	¤[10]			B		C04	7-46
BRANCH RELATIVE ON COUNT (32)	BRCT	RI-b			¤[9]			B		A76	7-47
BRANCH RELATIVE ON COUNT (64)	BRCTG	RI-b		N	¤[9]			B		A77	7-47
BRANCH RELATIVE ON COUNT HIGH (32)	BRCTH	RIL-b		HW	¤[9]			B		CC6	7-47
BRANCH RELATIVE ON INDEX HIGH (32)	BRXH	RSI			¤[9]			B		84	7-48
BRANCH RELATIVE ON INDEX HIGH (64)	BRXHG	RIE-e		N	¤[9]			B		EC44	7-48
BRANCH RELATIVE ON INDEX LOW OR EQ. (32)	BRXLE	RSI			¤[9]			B		85	7-48

Figure 7-1. Summary of General Instructions (Part 2 of 14)

Name	Mnemonic	Characteristics																Opcode	Page
BRANCH RELATIVE ON INDEX LOW OR EQ. (64)	BRXLG	RIE-e		N	α[9]								B					EC45	7-48
CHECKSUM	CKSM	RRE	C		α[9]	A	SP	IC									R_2	B241	7-49
CIPHER MESSAGE	KM	RRE	C	MS	α[5,9]	A	SP	IC			GM	I1		ST	R_1	R_2		B92E	7-53
CIPHER MESSAGE WITH AUTHENTICATION	KMA	RRF-b	C	M8	α[5,9]	A	SP	IC			GM	I1		ST	R_1	R_2	R_3	B929	7-78
CIPHER MESSAGE WITH CHAINING	KMC	RRE	C	MS	α[5,9]	A	SP	IC			GM	I1		ST	R_1	R_2		B92F	7-53
CIPHER MESSAGE WITH CIPHER FEEDBACK	KMF	RRE	C	M4	α[5,9]	A	SP	IC			GM	I1		ST	R_1	R_2		B92A	7-92
CIPHER MESSAGE WITH COUNTER	KMCTR	RRF-b	C	M4	α[5,9]	A	SP	IC			GM	I1		ST	R_1	R_2	R_3	B92D	7-107
CIPHER MESSAGE WITH OUTPUT FEEDBACK	KMO	RRE	C	M4	α[5,9]	A	SP	IC			GM	I1		ST	R_1	R_2		B92B	7-120
COMPARE (32)	C	RX-a	C			A											B_2	59	7-134
COMPARE (32)	CR	RR	C															19	7-134
COMPARE (32)	CY	RXY-a	C	LD		A											B_2	E359	7-134
COMPARE (64)	CG	RXY-a	C	N		A											B_2	E320	7-134
COMPARE (64)	CGR	RRE	C	N														B920	7-134
COMPARE (64←32)	CGF	RXY-a	C	N		A											B_2	E330	7-134
COMPARE (64←32)	CGFR	RRE	C	N														B930	7-134
COMPARE AND BRANCH (32)	CRB	RRS		GE	α[9]								B					ECF6	7-135
COMPARE AND BRANCH (64)	CGRB	RRS		GE	α[9]								B					ECE4	7-135
COMPARE AND BRANCH RELATIVE (32)	CRJ	RIE-b		GE	α[10]								B					EC76	7-135
COMPARE AND BRANCH RELATIVE (64)	CGRJ	RIE-b		GE	α[10]								B					EC64	7-136
COMPARE AND FORM CODEWORD	CFC	S	C		α[9]	A	SP	II			GM	I1						B21A	7-137
COMPARE AND SWAP (32)	CS	RS-a	C		α[9]	A	SP		$					ST			B_2	BA	7-144
COMPARE AND SWAP (32)	CSY	RSY-a	C	LD	α[9]	A	SP		$					ST			B_2	EB14	7-144
COMPARE AND SWAP (64)	CSG	RSY-a	C	N	α[9]	A	SP		$					ST			B_2	EB30	7-144
COMPARE AND SWAP AND STORE	CSST	SSF	C	CS	α[1]	A	SP		$		GM			ST	B_1		B_2	C82	7-146
COMPARE AND TRAP (32)	CRT	RRF-c		GE				Dc										B972	7-149
COMPARE AND TRAP (64)	CGRT	RRF-c		GE				Dc										B960	7-149
COMPARE DOUBLE AND SWAP (32)	CDS	RS-a	C		α[9]	A	SP		$					ST			B_2	BB	7-144
COMPARE DOUBLE AND SWAP (32)	CDSY	RSY-a	C	LD	α[9]	A	SP		$					ST			B_2	EB31	7-144
COMPARE DOUBLE AND SWAP (64)	CDSG	RSY-a	C	N	α[9]	A	SP		$					ST			B_2	EB3E	7-144
COMPARE HALFWORD (32←16)	CH	RX-a	C			A											B_2	49	7-150
COMPARE HALFWORD (32←16)	CHY	RXY-a	C	LD		A											B_2	E379	7-150
COMPARE HALFWORD (64←16)	CGH	RXY-a	C	GE		A											B_2	E334	7-150
COMPARE HALFWORD IMMEDIATE (16←16)	CHHSI	SIL	C	GE		A									B_1			E554	7-150
COMPARE HALFWORD IMMEDIATE (32←16)	CHI	RI-a	C															A7E	7-150
COMPARE HALFWORD IMMEDIATE (32←16)	CHSI	SIL	C	GE		A									B_1			E55C	7-150
COMPARE HALFWORD IMMEDIATE (64←16)	CGHI	RI-a	C	N														A7F	7-150
COMPARE HALFWORD IMMEDIATE (64←16)	CGHSI	SIL	C	GE		A									B_1			E558	7-150
COMPARE HALFWORD RELATIVE LONG (32←16)	CHRL	RIL-b	C	GE		A*												C65	7-150
COMPARE HALFWORD RELATIVE LONG (64←16)	CGHRL	RIL-b	C	GE		A*												C64	7-150
COMPARE HIGH (32)	CHF	RXY-a	C	HW		A											B_2	E3CD	7-151
COMPARE HIGH (32)	CHHR	RRE	C	HW														B9CD	7-151
COMPARE HIGH (32)	CHLR	RRE	C	HW														B9DD	7-151
COMPARE IMMEDIATE (32)	CFI	RIL-a	C	EI														C2D	7-134
COMPARE IMMEDIATE (64←32)	CGFI	RIL-a	C	EI														C2C	7-135
COMPARE IMMEDIATE AND BRANCH (32←8)	CIB	RIS		GE	α[9]								B					ECFE	7-136
COMPARE IMMEDIATE AND BRANCH (64←8)	CGIB	RIS		GE	α[9]								B					ECFC	7-136
COMPARE IMMEDIATE AND BRANCH RELATIVE (32←8)	CIJ	RIE-c		GE	α[10]								B					EC7E	7-136
COMPARE IMMEDIATE AND BRANCH RELATIVE (64←8)	CGIJ	RIE-c		GE	α[10]								B					EC7C	7-136
COMPARE IMMEDIATE AND TRAP (32←16)	CIT	RIE-a		GE				Dc										EC72	7-149
COMPARE IMMEDIATE AND TRAP (64←16)	CGIT	RIE-a		GE				Dc										EC70	7-149
COMPARE IMMEDIATE HIGH (32)	CIH	RIL-a	C	HW														CCD	7-151
COMPARE LOGICAL (32)	CL	RX-a	C			A											B_2	55	7-152
COMPARE LOGICAL (32)	CLR	RR	C															15	7-152

Figure 7-1. Summary of General Instructions (Part 3 of 14)

Name	Mnemonic	Characteristics											Opcode	Page
COMPARE LOGICAL (32)	CLY	RXY-a	C	LD		A						B_2	E355	7-152
COMPARE LOGICAL (64)	CLG	RXY-a	C	N		A						B_2	E321	7-152
COMPARE LOGICAL (64)	CLGR	RRE	C	N									B921	7-152
COMPARE LOGICAL (64←32)	CLGF	RXY-a	C	N		A						B_2	E331	7-152
COMPARE LOGICAL (64←32)	CLGFR	RRE	C	N									B931	7-152
COMPARE LOGICAL (character)	CLC	SS-a	C		¤[9]	A					B_1	B_2	D5	7-152
COMPARE LOGICAL (immediate)	CLI	SI	C			A					B_1		95	7-152
COMPARE LOGICAL (immediate)	CLIY	SIY	C	LD		A					B_1		EB55	7-152
COMPARE LOGICAL AND BRANCH (32)	CLRB	RRS		GE	¤[9]				B				ECF7	7-154
COMPARE LOGICAL AND BRANCH (64)	CLGRB	RRS		GE	¤[9]				B				ECE5	7-154
COMPARE LOGICAL AND BRANCH RELATIVE (32)	CLRJ	RIE-b		GE	¤[10]				B				EC77	7-154
COMPARE LOGICAL AND BRANCH RELATIVE (64)	CLGRJ	RIE-b		GE	¤[10]				B				EC65	7-154
COMPARE LOGICAL AND TRAP (32)	CLRT	RRF-c		GE			Dc						B973	7-155
COMPARE LOGICAL AND TRAP (32)	CLT	RSY-b		MI1		A	Dc					B_2	EB23	7-155
COMPARE LOGICAL AND TRAP (64)	CLGRT	RRF-c		GE			Dc						B961	7-155
COMPARE LOGICAL AND TRAP (64)	CLGT	RSY-b		MI1		A	Dc					B_2	EB2B	7-155
COMPARE LOGICAL CHAR. UNDER MASK (high)	CLMH	RSY-b	C	N		A						B_2	EB20	7-157
COMPARE LOGICAL CHAR. UNDER MASK (low)	CLM	RS-b	C			A						B_2	BD	7-157
COMPARE LOGICAL CHAR. UNDER MASK (low)	CLMY	RSY-b	C	LD		A						B_2	EB21	7-157
COMPARE LOGICAL HIGH (32)	CLHF	RXY-a	C	HW		A						B_2	E3CF	7-157
COMPARE LOGICAL HIGH (32)	CLHHR	RRE	C	HW									B9CF	7-157
COMPARE LOGICAL HIGH (32)	CLHLR	RRE	C	HW									B9DF	7-157
COMPARE LOGICAL IMMEDIATE (16←16)	CLHHSI	SIL	C	GE		A					B_1		E555	7-152
COMPARE LOGICAL IMMEDIATE (32)	CLFI	RIL-a	C	EI									C2F	7-152
COMPARE LOGICAL IMMEDIATE (32←16)	CLFHSI	SIL	C	GE		A					B_1		E55D	7-152
COMPARE LOGICAL IMMEDIATE (64←16)	CLGHSI	SIL	C	GE		A					B_1		E559	7-152
COMPARE LOGICAL IMMEDIATE (64←32)	CLGFI	RIL-a	C	EI									C2E	7-152
COMPARE LOGICAL IMMEDIATE AND BRANCH (32←8)	CLIB	RIS		GE	¤[9]				B				ECFF	7-154
COMPARE LOGICAL IMMEDIATE AND BRANCH (64←8)	CLGIB	RIS		GE	¤[9]				B				ECFD	7-154
COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (32←8)	CLIJ	RIE-c		GE	¤[10]				B				EC7F	7-154
COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (64←8)	CLGIJ	RIE-c		GE	¤[10]				B				EC7D	7-154
COMPARE LOGICAL IMMEDIATE AND TRAP (32←16)	CLFIT	RIE-a		GE			Dc						EC73	7-156
COMPARE LOGICAL IMMEDIATE AND TRAP (64←16)	CLGIT	RIE-a		GE			Dc						EC71	7-156
COMPARE LOGICAL IMMEDIATE HIGH (32)	CLIH	RIL-a	C	HW									CCF	7-158
COMPARE LOGICAL LONG	CLCL	RR	C		¤[9]	A	SP	II			R_1	R_2	0F	7-158
COMPARE LOGICAL LONG EXTENDED	CLCLE	RS-a	C		¤[9]	A	SP	IC			R_1	R_3	A9	7-160
COMPARE LOGICAL LONG UNICODE	CLCLU	RSY-a	C	E2	¤[9]	A	SP	IC			R_1	R_2	EB8F	7-163
COMPARE LOGICAL RELATIVE LONG (32)	CLRL	RIL-b	C	GE		A*	SP						C6F	7-153
COMPARE LOGICAL RELATIVE LONG (32←16)	CLHRL	RIL-b	C	GE		A*							C67	7-153
COMPARE LOGICAL RELATIVE LONG (64)	CLGRL	RIL-b	C	GE		A*	SP						C6A	7-153
COMPARE LOGICAL RELATIVE LONG (64←16)	CLGHRL	RIL-b	C	GE		A*							C66	7-153
COMPARE LOGICAL RELATIVE LONG (64←32)	CLGFRL	RIL-b	C	GE		A*	SP						C6E	7-153
COMPARE LOGICAL STRING	CLST	RRE	C		¤[9]	A	SP	IC		G0	R_1	R_2	B25D	7-166
COMPARE RELATIVE LONG (32)	CRL	RIL-b	C	GE		A*	SP						C6D	7-135
COMPARE RELATIVE LONG (64)	CGRL	RIL-b	C	GE		A*	SP						C68	7-135
COMPARE RELATIVE LONG (64←32)	CGFRL	RIL-b	C	GE		A*	SP						C6C	7-135
COMPARE UNTIL SUBSTRING EQUAL	CUSE	RRE	C		¤[9]	A	SP	II		GM	R_1	R_2	B257	7-167
COMPRESSION CALL	CMPSC	RRE	C		¤[5,9]	A	SP	II	Dg	GM	R_1	R_2	B263	7-170

Figure 7-1. Summary of General Instructions (Part 4 of 14)

Name	Mnemonic	Characteristics															Opcode	Page
COMPUTE INTERMEDIATE MESSAGE DIGEST	KIMD	RRE	C	MS	α[5,9]	A	SP	IC			GM	I1	ST			R_2	B93E	7-188
COMPUTE LAST MESSAGE DIGEST	KLMD	RRE	C	MS	α[5,9]	A	SP	IC			GM	I1	ST			R_2	B93F	7-201
COMPUTE MESSAGE AUTHENTICATION CODE	KMAC	RRE	C	MS	α[5,9]	A	SP	IC			GM	I1	ST			R_2	B91E	7-219
CONVERT TO BINARY (32)	CVB	RX-a			α[9]	A			Dg	IK						B_2	4F	7-230
CONVERT TO BINARY (32)	CVBY	RXY-a		LD	α[9]	A			Dg	IK						B_2	E306	7-230
CONVERT TO BINARY (64)	CVBG	RXY-a		N	α[9]	A			Dg	IK						B_2	E30E	7-230
CONVERT TO DECIMAL (32)	CVD	RX-a			α[9]	A							ST			B_2	4E	7-231
CONVERT TO DECIMAL (32)	CVDY	RXY-a		LD	α[9]	A							ST			B_2	E326	7-231
CONVERT TO DECIMAL (64)	CVDG	RXY-a		N	α[9]	A							ST			B_2	E32E	7-231
CONVERT UNICODE TO UTF-8	CUUTF	RRF-c	C		α[5,9]	A	SP	IC					ST		R_1	R_2	B2A6	7-234
CONVERT UTF-16 TO UTF-32	CU24	RRF-c	C	E3	α[5,9]	A	SP	IC					ST		R_1	R_2	B9B1	7-231
CONVERT UTF-16 TO UTF-8	CU21	RRF-c	C		α[5,9]	A	SP	IC					ST		R_1	R_2	B2A6	7-234
CONVERT UTF-32 TO UTF-16	CU42	RRE	C	E3	α[5,9]	A	SP	IC					ST		R_1	R_2	B9B3	7-238
CONVERT UTF-32 TO UTF-8	CU41	RRE	C	E3	α[5,9]	A	SP	IC					ST		R_1	R_2	B9B2	7-241
CONVERT UTF-8 TO UNICODE	CUTFU	RRF-c	C		α[5,9]	A	SP	IC					ST		R_1	R_2	B2A7	7-244
CONVERT UTF-8 TO UTF-16	CU12	RRF-c	C		α[5,9]	A	SP	IC					ST		R_1	R_2	B2A7	7-244
CONVERT UTF-8 TO UTF-32	CU14	RRF-c	C	E3	α[5,9]	A	SP	IC					ST		R_1	R_2	B9B0	7-248
COPY ACCESS	CPYA	RRE			α[6]										U_1	U_2	B24D	7-252
DIVIDE (32←64)	D	RX-a			α[9]	A	SP			IK						B_2	5D	7-252
DIVIDE (32←64)	DR	RR			α[9]		SP			IK							1D	7-252
DIVIDE LOGICAL (32←64)	DL	RXY-a		N3	α[9]	A	SP			IK						B_2	E397	7-253
DIVIDE LOGICAL (32←64)	DLR	RRE		N3	α[9]		SP			IK							B997	7-253
DIVIDE LOGICAL (64←128)	DLG	RXY-a		N	α[9]	A	SP			IK						B_2	E387	7-253
DIVIDE LOGICAL (64←128)	DLGR	RRE		N	α[9]		SP			IK							B987	7-253
DIVIDE SINGLE (64)	DSG	RXY-a		N	α[9]	A	SP			IK						B_2	E30D	7-254
DIVIDE SINGLE (64)	DSGR	RRE		N	α[9]		SP			IK							B90D	7-254
DIVIDE SINGLE (64←32)	DSGF	RXY-a		N	α[9]	A	SP			IK						B_2	E31D	7-254
DIVIDE SINGLE (64←32)	DSGFR	RRE		N	α[9]		SP			IK							B91D	7-254
EXCLUSIVE OR (32)	X	RX-a	C			A										B_2	57	7-254
EXCLUSIVE OR (32)	XR	RR	C														17	7-254
EXCLUSIVE OR (32)	XRK	RRF-a	C	DO													B9F7	7-254
EXCLUSIVE OR (32)	XY	RXY-a	C	LD		A										B_2	E357	7-254
EXCLUSIVE OR (64)	XG	RXY-a	C	N		A										B_2	E382	7-254
EXCLUSIVE OR (64)	XGR	RRE	C	N													B982	7-254
EXCLUSIVE OR (64)	XGRK	RRF-a	C	DO													B9E7	7-254
EXCLUSIVE OR (character)	XC	SS-a	C		α[9]	A							ST		B_1	B_2	D7	7-255
EXCLUSIVE OR (immediate)	XI	SI	C			A				£[2]			ST		B_1		97	7-255
EXCLUSIVE OR (immediate)	XIY	SIY	C	LD		A				£[2]			ST		B_1		EB57	7-255
EXCLUSIVE OR IMMEDIATE (high)	XIHF	RIL-a	C	EI													C06	7-256
EXCLUSIVE OR IMMEDIATE (low)	XILF	RIL-a	C	EI													C07	7-256
EXECUTE	EX	RX-a			α[9]	AI	SP				EX						44	7-256
EXECUTE RELATIVE LONG	EXRL	RIL-b		XX	α[9]	AI*					EX						C60	7-256
EXTRACT ACCESS	EAR	RRE														U_2	B24F	7-257
EXTRACT CPU ATTRIBUTE	ECAG	RSY-a		GE	α[9]												EB4C	7-257
EXTRACT CPU TIME	ECTG	SSF		ET	α[8,9]	A					GM		R_3		B_1	B_2	C81	7-260
EXTRACT PSW	EPSW	RRE		N3	α[8,9]												B98D	7-261
EXTRACT TRANSACTION NESTING DEPTH	ETND	RRE		TX	α[9]			SO									B2EC	7-261
FIND LEFTMOST ONE	FLOGR	RRE	C	EI			SP										B983	7-262
INSERT CHARACTER	IC	RX-a				A										B_2	43	7-262
INSERT CHARACTER	ICY	RXY-a		LD		A										B_2	E373	7-262
INSERT CHARACTERS UNDER MASK (high)	ICMH	RSY-b	C	N		A										B_2	EB80	7-262
INSERT CHARACTERS UNDER MASK (low)	ICM	RS-b	C			A										B_2	BF	7-262
INSERT CHARACTERS UNDER MASK (low)	ICMY	RSY-b	C	LD		A										B_2	EB81	7-262
INSERT IMMEDIATE (high high)	IIHH	RI-a		N													A50	7-263
INSERT IMMEDIATE (high low)	IIHL	RI-a		N													A51	7-263

Figure 7-1. Summary of General Instructions (Part 5 of 14)

Name	Mnemonic	Characteristics											Opcode	Page
INSERT IMMEDIATE (high)	IIHF	RIL-a		EI									C08	7-263
INSERT IMMEDIATE (low high)	IILH	RI-a		N									A52	7-263
INSERT IMMEDIATE (low low)	IILL	RI-a		N									A53	7-263
INSERT IMMEDIATE (low)	IILF	RIL-a		EI									C09	7-263
INSERT PROGRAM MASK	IPM	RRE											B222	7-264
LOAD (32)	L	RX-a				A						B_2	58	7-264
LOAD (32)	LR	RR											18	7-264
LOAD (32)	LY	RXY-a		LD		A						B_2	E358	7-264
LOAD (64)	LG	RXY-a		N		A						B_2	E304	7-264
LOAD (64)	LGR	RRE		N									B904	7-264
LOAD (64←32)	LGF	RXY-a		N		A						B_2	E314	7-264
LOAD (64←32)	LGFR	RRE		N									B914	7-264
LOAD ACCESS MULTIPLE	LAM	RS-a			¤[6]	A	SP					UB	9A	7-265
LOAD ACCESS MULTIPLE	LAMY	RSY-a		LD	¤[6]	A	SP					UB	EB9A	7-265
LOAD ADDRESS	LA	RX-a											41	7-266
LOAD ADDRESS	LAY	RXY-a		LD									E371	7-266
LOAD ADDRESS EXTENDED	LAE	RX-a			¤[6]							U_1 BP	51	7-266
LOAD ADDRESS EXTENDED	LAEY	RXY-a		GE	¤[6]							U_1 BP	E375	7-266
LOAD ADDRESS RELATIVE LONG	LARL	RIL-b		N3									C00	7-267
LOAD AND ADD (32)	LAA	RSY-a	C	IA	¤[9]	A	SP	IF	£		ST	B_2	EBF8	7-268
LOAD AND ADD (64)	LAAG	RSY-a	C	IA	¤[9]	A	SP	IF	£		ST	B_2	EBE8	7-268
LOAD AND ADD LOGICAL (32)	LAAL	RSY-a	C	IA	¤[9]	A	SP		£		ST	B_2	EBFA	7-268
LOAD AND ADD LOGICAL (64)	LAALG	RSY-a	C	IA	¤[9]	A	SP		£		ST	B_2	EBEA	7-268
LOAD AND AND (32)	LAN	RSY-a	C	IA	¤[9]	A	SP		£		ST	B_2	EBF4	7-269
LOAD AND AND (64)	LANG	RSY-a	C	IA	¤[9]	A	SP		£		ST	B_2	EBE4	7-269
LOAD AND EXCLUSIVE OR (32)	LAX	RSY-a	C	IA	¤[9]	A	SP		£		ST	B_2	EBF7	7-269
LOAD AND EXCLUSIVE OR (64)	LAXG	RSY-a	C	IA	¤[9]	A	SP		£		ST	B_2	EBE7	7-269
LOAD AND OR (32)	LAO	RSY-a	C	IA	¤[9]	A	SP		£		ST	B_2	EBF6	7-270
LOAD AND OR (64)	LAOG	RSY-a	C	IA	¤[9]	A	SP		£		ST	B_2	EBE6	7-270
LOAD AND TEST (32)	LT	RXY-a	C	EI		A						B_2	E312	7-271
LOAD AND TEST (32)	LTR	RR	C										12	7-270
LOAD AND TEST (64)	LTG	RXY-a	C	EI		A						B_2	E302	7-271
LOAD AND TEST (64)	LTGR	RRE	C	N									B902	7-270
LOAD AND TEST (64←32)	LTGF	RXY-a	C	GE		A						B_2	E332	7-271
LOAD AND TEST (64←32)	LTGFR	RRE	C	N									B912	7-270
LOAD AND TRAP (32)	LAT	RXY-a		LT		A		Dc				B_2	E39F	7-271
LOAD AND TRAP (64)	LGAT	RXY-a		LT		A		Dc				B_2	E385	7-271
LOAD AND ZERO RIGHTMOST BYTE (32)	LZRF	RXY-a		LZ		A						B_2	E33B	7-271
LOAD AND ZERO RIGHTMOST BYTE (64)	LZRG	RXY-a		LZ		A						B_2	E32A	7-271
LOAD BYTE (32←8)	LB	RXY-a		LD		A							E376	7-272
LOAD BYTE (32←8)	LBR	RRE		EI									B926	7-272
LOAD BYTE (64←8)	LGB	RXY-a		LD		A							E377	7-272
LOAD BYTE (64←8)	LGBR	RRE		EI									B906	7-272
LOAD BYTE HIGH (32←8)	LBH	RXY-a		HW		A						B_2	E3C0	7-272
LOAD COMPLEMENT (32)	LCR	RR	C					IF					13	7-272
LOAD COMPLEMENT (64)	LCGR	RRE	C	N				IF					B903	7-273
LOAD COMPLEMENT (64←32)	LCGFR	RRE	C	N									B913	7-273
LOAD COUNT TO BLOCK BOUNDARY	LCBB	RXE	C	VF			SP						E727	7-273
LOAD GUARDED (64)	LGG	RXY-a		GF	¤[12]	A	SP				B ST	B_2	E34C	7-274
LOAD GUARDED STORAGE CONTROLS	LGSC	RXY-a		GF	¤[1]	A		SO				B_2	E34D	7-275
LOAD HALFWORD (32←16)	LH	RX-a				A						B_2	48	7-276
LOAD HALFWORD (32←16)	LHR	RRE		EI									B927	7-276
LOAD HALFWORD (32←16)	LHY	RXY-a		LD		A						B_2	E378	7-276
LOAD HALFWORD (64←16)	LGH	RXY-a		N		A						B_2	E315	7-276
LOAD HALFWORD (64←16)	LGHR	RRE		EI									B907	7-276

Figure 7-1. Summary of General Instructions (Part 6 of 14)

Name	Mne-monic	Characteristics								Op-code	Page
LOAD HALFWORD HIGH (32←16)	LHH	RXY-a	HW		A				B_2	E3C4	7-277
LOAD HALFWORD HIGH IMMEDIATE ON CONDITION (32←16)	LOCHHI	RIE-g	L2							EC4E	7-277
LOAD HALFWORD IMMEDIATE (32←16)	LHI	RI-a								A78	7-276
LOAD HALFWORD IMMEDIATE (64←16)	LGHI	RI-a	N							A79	7-276
LOAD HALFWORD IMMEDIATE ON CONDITION (32←16)	LOCHI	RIE-g	L2							EC42	7-277
LOAD HALFWORD IMMEDIATE ON CONDITION (64←16)	LOCGHI	RIE-g	L2							EC46	7-277
LOAD HALFWORD RELATIVE LONG (32←16)	LHRL	RIL-b	GE		A*					C45	7-276
LOAD HALFWORD RELATIVE LONG (64←16)	LGHRL	RIL-b	GE		A*					C44	7-276
LOAD HIGH (32)	LFH	RXY-a	HW		A				B_2	E3CA	7-278
LOAD HIGH AND TRAP (32)	LFHAT	RXY-a	LT		A		Dc		B_2	E3C8	7-278
LOAD HIGH ON CONDITION (32)	LOCFH	RSY-b	L2		A				B_2	EBE0	7-284
LOAD HIGH ON CONDITION (32)	LOCFHR	RRF-c	L2							B9E0	7-284
LOAD IMMEDIATE (64←32)	LGFI	RIL-a	EI							C01	7-264
LOAD LOGICAL (64←32)	LLGF	RXY-a	N		A				B_2	E316	7-278
LOAD LOGICAL (64←32)	LLGFR	RRE	N							B916	7-278
LOAD LOGICAL AND TRAP (64←32)	LLGFAT	RXY-a	LT		A		Dc		B_2	E39D	7-279
LOAD LOGICAL AND ZERO RIGHTMOST BYTE (64←32)	LLZRGF	RXY-a	LZ		A				B_2	E33A	7-279
LOAD LOGICAL AND SHIFT GUARDED (64←32)	LLGFSG	RXY-a	GF	[12]	A	SP		B ST	B_2	E348	7-274
LOAD LOGICAL CHARACTER (32←8)	LLC	RXY-a	EI		A				B_2	E394	7-279
LOAD LOGICAL CHARACTER (32←8)	LLCR	RRE	EI							B994	7-279
LOAD LOGICAL CHARACTER (64←8)	LLGC	RXY-a	N		A				B_2	E390	7-279
LOAD LOGICAL CHARACTER (64←8)	LLGCR	RRE	EI							B984	7-279
LOAD LOGICAL CHARACTER HIGH (32←8)	LLCH	RXY-a	HW		A				B_2	E3C2	7-280
LOAD LOGICAL HALFWORD (32←16)	LLH	RXY-a	EI		A				B_2	E395	7-280
LOAD LOGICAL HALFWORD (32←16)	LLHR	RRE	EI							B995	7-280
LOAD LOGICAL HALFWORD (64←16)	LLGH	RXY-a	N		A				B_2	E391	7-280
LOAD LOGICAL HALFWORD (64←16)	LLGHR	RRE	EI							B985	7-280
LOAD LOGICAL HALFWORD HIGH (32←16)	LLHH	RXY-a	HW		A				B_2	E3C6	7-281
LOAD LOGICAL HALFWORD RELATIVE LONG (32←16)	LLHRL	RIL-b	GE		A*					C42	7-280
LOAD LOGICAL HALFWORD RELATIVE LONG (64←16)	LLGHRL	RIL-b	GE		A*					C46	7-280
LOAD LOGICAL IMMEDIATE (high high)	LLIHH	RI-a	N							A5C	7-281
LOAD LOGICAL IMMEDIATE (high low)	LLIHL	RI-a	N							A5D	7-281
LOAD LOGICAL IMMEDIATE (high)	LLIHF	RIL-a	EI							C0E	7-281
LOAD LOGICAL IMMEDIATE (low high)	LLILH	RI-a	N							A5E	7-281
LOAD LOGICAL IMMEDIATE (low low)	LLILL	RI-a	N							A5F	7-281
LOAD LOGICAL IMMEDIATE (low)	LLILF	RIL-a	EI							C0F	7-281
LOAD LOGICAL RELATIVE LONG (64←32)	LLGFRL	RIL-b	GE		A*	SP				C4E	7-278
LOAD LOGICAL THIRTY ONE BITS (64←31)	LLGT	RXY-a	N		A				B_2	E317	7-282
LOAD LOGICAL THIRTY ONE BITS (64←31)	LLGTR	RRE	N							B917	7-281
LOAD LOGICAL THIRTY ONE BITS AND TRAP (64←31)	LLGTAT	RXY-a	LT		A		Dc		B_2	E39C	7-282
LOAD MULTIPLE (32)	LM	RS-a			A				B_2	98	7-282
LOAD MULTIPLE (32)	LMY	RSY-a	LD		A				B_2	EB98	7-282
LOAD MULTIPLE (64)	LMG	RSY-a	N		A				B_2	EB04	7-282
LOAD MULTIPLE DISJOINT (64←32&32)	LMD	SS-e	N	[9]	A				B_2 B_4	EF	7-283
LOAD MULTIPLE HIGH (32)	LMH	RSY-a	N		A				B_2	EB96	7-283
LOAD NEGATIVE (32)	LNR	RR	C							11	7-283
LOAD NEGATIVE (64)	LNGR	RRE	C	N						B901	7-283
LOAD NEGATIVE (64←32)	LNGFR	RRE	C	N						B911	7-284
LOAD ON CONDITION (32)	LOC	RSY-b	L1		A				B_2	EBF2	7-284

Figure 7-1. Summary of General Instructions (Part 7 of 14)

Name	Mnemonic	Characteristics											Opcode	Page
LOAD ON CONDITION (32)	LOCR	RRF-c		L1		A							B9F2	7-284
LOAD ON CONDITION (64)	LOCG	RSY-b		L1		A						B_2	EBE2	7-284
LOAD ON CONDITION (64)	LOCGR	RRF-c		L1		A							B9E2	7-284
LOAD PAIR DISJOINT (32)	LPD	SSF	C	IA	¤[9]	A	SP				B_1	B_2	C84	7-285
LOAD PAIR DISJOINT (64)	LPDG	SSF	C	IA	¤[9]	A	SP				B_1	B_2	C85	7-285
LOAD PAIR FROM QUADWORD (64&64←128)	LPQ	RXY-a		N	¤[9]	A	SP					B_2	E38F	7-286
LOAD POSITIVE (32)	LPR	RR	C					IF					10	7-287
LOAD POSITIVE (64)	LPGR	RRE	C	N				IF					B900	7-287
LOAD POSITIVE (64←32)	LPGFR	RRE	C	N									B910	7-287
LOAD RELATIVE LONG (32)	LRL	RIL-b		GE		A	SP						C4D	7-264
LOAD RELATIVE LONG (64)	LGRL	RIL-b		GE		A*	SP						C48	7-264
LOAD RELATIVE LONG (64←32)	LGFRL	RIL-b		GE		A*	SP						C4C	7-264
LOAD REVERSED (16)	LRVH	RXY-a		N3		A						B_2	E31F	7-287
LOAD REVERSED (32)	LRV	RXY-a		N3		A						B_2	E31E	7-287
LOAD REVERSED (32)	LRVR	RRE		N3									B91F	7-287
LOAD REVERSED (64)	LRVG	RXY-a		N		A						B_2	E30F	7-287
LOAD REVERSED (64)	LRVGR	RRE		N									B90F	7-287
MONITOR CALL	MC	SI			¤[4,8,9]		SP	ME		ST			AF	7-288
MOVE (16←16)	MVHHI	SIL		GE		A				ST	B_1		E544	7-289
MOVE (32←16)	MVHI	SIL		GE		A				ST	B_1		E54C	7-289
MOVE (64←16)	MVGHI	SIL		GE		A				ST	B_1		E548	7-289
MOVE (character)	MVC	SS-a			¤[9]	A				ST	B_1	B_2	D2	7-289
MOVE (immediate)	MVI	SI				A				ST	B_1		92	7-289
MOVE (immediate)	MVIY	SIY		LD		A				ST	B_1		EB52	7-289
MOVE INVERSE	MVCIN	SS-a			¤[9]	A				ST	B_1	B_2	E8	7-290
MOVE LONG	MVCL	RR	C		¤[9]	A	SP	II		ST	R_1	R_2	0E	7-290
MOVE LONG EXTENDED	MVCLE	RS-a	C		¤[9]	A	SP	IC		ST	R_1	R_3	A8	7-294
MOVE LONG UNICODE	MVCLU	RSY-a	C	E2	¤[9]	A	SP	IC		ST	R_1	R_3	EB8E	7-297
MOVE NUMERICS	MVN	SS-a			¤[9]	A				ST	B_1	B_2	D1	7-301
MOVE STRING	MVST	RRE	C		¤[9]	A	SP	IC	G0	ST	R_1	R_2	B255	7-301
MOVE WITH OFFSET	MVO	SS-b			¤[9]	A				ST	B_1	B_2	F1	7-302
MOVE ZONES	MVZ	SS-a			¤[9]	A				ST	B_1	B_2	D3	7-303
MULTIPLY (128←64)	MG	RXY-a		MI2		A	SP					B_2	E384	7-304
MULTIPLY (128←64)	MGRK	RRF-a		MI2			SP						B9EC	7-304
MULTIPLY (64←32)	M	RX-a				A	SP					B_2	5C	7-304
MULTIPLY (64←32)	MFY	RXY-a		GE		A	SP					B_2	E35C	7-304
MULTIPLY (64←32)	MR	RR					SP						1C	7-304
MULTIPLY HALFWORD (32←16)	MH	RX-a				A						B_2	4C	7-305
MULTIPLY HALFWORD (32←16)	MHY	RXY-a		GE		A						B_2	E37C	7-305
MULTIPLY HALFWORD (64←16)	MGH	RXY-a		MI2		A						B_2	E33C	7-305
MULTIPLY HALFWORD IMMEDIATE (32←16)	MHI	RI-a											A7C	7-305
MULTIPLY HALFWORD IMMEDIATE (64←16)	MGHI	RI-a		N									A7D	7-305
MULTIPLY LOGICAL (128←64)	MLG	RXY-a		N		A	SP					B_2	E386	7-306
MULTIPLY LOGICAL (128←64)	MLGR	RRE		N			SP						B986	7-306
MULTIPLY LOGICAL (64←32)	ML	RXY-a		N3		A	SP					B_2	E396	7-306
MULTIPLY LOGICAL (64←32)	MLR	RRE		N3			SP						B996	7-305
MULTIPLY SINGLE (32)	MS	RX-a				A						B_2	71	7-307
MULTIPLY SINGLE (32)	MSC	RXY-a	C	MI2		A		IF				B_2	E353	7-307
MULTIPLY SINGLE (32)	MSR	RRE											B252	7-307
MULTIPLY SINGLE (32)	MSRKC	RRF-a	C	MI2				IF					B9FD	7-307
MULTIPLY SINGLE (32)	MSY	RXY-a		LD		A						B_2	E351	7-307
MULTIPLY SINGLE (64)	MSG	RXY-a		N		A						B_2	E30C	7-307
MULTIPLY SINGLE (64)	MSGC	RXY-a	C	MI2		A		IF				B_2	E383	7-307
MULTIPLY SINGLE (64)	MSGR	RRE		N									B90C	7-307
MULTIPLY SINGLE (64)	MSGRKC	RRF-a	C	MI2				IF					B9ED	7-307

Figure 7-1. Summary of General Instructions (Part 8 of 14)

Name	Mne-monic	Characteristics												Op-code	Page
MULTIPLY SINGLE (64←32)	MSGF	RXY-a		N		A							B_2	E31C	7-307
MULTIPLY SINGLE (64←32)	MSGFR	RRE		N										B91C	7-307
MULTIPLY SINGLE IMMEDIATE (32)	MSFI	RIL-a		GE										C21	7-307
MULTIPLY SINGLE IMMEDIATE (64←32)	MSGFI	RIL-a		GE										C20	7-307
NEXT INSTRUCTION ACCESS INTENT	NIAI	IE		EH										B2FA	7-308
NONTRANSACTIONAL STORE (64)	NTSTG	RXY-a		TX	¤[9]	A	SP				ST		B_2	E325	7-310
OR (32)	O	RX-a	C			A							B_2	56	7-310
OR (32)	OR	RR	C											16	7-310
OR (32)	ORK	RRF-a	C	DO										B9F6	7-310
OR (32)	OY	RXY-a	C	LD		A							B_2	E356	7-310
OR (64)	OG	RXY-a	C	N		A							B_2	E381	7-311
OR (64)	OGR	RRE	C	N										B981	7-310
OR (64)	OGRK	RRF-a	C	DO										B9E6	7-310
OR (character)	OC	SS-a	C		¤[9]	A					ST	B_1	B_2	D6	7-311
OR (immediate)	OI	SI	C			A		£[2]			ST	B_1		96	7-311
OR (immediate)	OIY	SIY	C	LD		A		£[2]			ST	B_1		EB56	7-311
OR IMMEDIATE (high high)	OIHH	RI-a	C	N										A58	7-312
OR IMMEDIATE (high low)	OIHL	RI-a	C	N										A59	7-312
OR IMMEDIATE (high)	OIHF	RIL-a	C	EI										C0C	7-312
OR IMMEDIATE (low high)	OILH	RI-a	C	N										A5A	7-312
OR IMMEDIATE (low low)	OILL	RI-a	C	N										A5B	7-312
OR IMMEDIATE (low)	OILF	RIL-a	C	EI										C0D	7-312
PACK	PACK	SS-b			¤[9]	A					ST	B_1	B_2	F2	7-312
PACK ASCII	PKA	SS-f		E2	¤[9]	A	SP				ST	B_1	B_2	E9	7-313
PACK UNICODE	PKU	SS-f		E2	¤[9]	A	SP				ST	B_1	B_2	E1	7-314
PERFORM CRYPTOGRAPHIC COMPUTATION	PCC	RRE	C	M4	¤[5,9]	A	SP	IC		GM I1	ST			B92C	7-315
PERFORM PROCESSOR ASSIST	PPA	RRF-c		PA	¤[1]									B2E8	7-345
PERFORM LOCKED OPERATION	PLO	SS-e	C		¤[1]	A	SP		$	GM	ST		FC	EE	7-331
PERFORM RANDOM NUMBER OPERATION	PRNO	RRE	C	M5	¤[5,9]	A	SP	IC	Dg	GM I1	ST	R_1	R_2	B93C	7-346
POPULATION COUNT	POPCNT	RRE	C	PK										B9E1	7-359
PREFETCH DATA	PFD	RXY-b		GE	¤[9,11]								B_2	E336	7-359
PREFETCH DATA RELATIVE LONG	PFDRL	RIL-c		GE	¤[9,11]									C62	7-359
ROTATE LEFT SINGLE LOGICAL (32)	RLL	RSY-a		N3										EB1D	7-361
ROTATE LEFT SINGLE LOGICAL (64)	RLLG	RSY-a		N										EB1C	7-361
ROTATE THEN AND SELECTED BITS (64)	RNSBG	RIE-f	C	GE										EC54	7-361
ROTATE THEN EXCLUSIVE OR SEL. BITS (64)	RXSBG	RIE-f	C	GE										EC57	7-362
ROTATE THEN INSERT SELECTED BITS (64)	RISBG	RIE-f	C	GE										EC55	7-363
ROTATE THEN INSERT SELECTED BITS (64)	RISBGN	RIE-f		MI1										EC59	7-363
ROTATE THEN INSERT SELECTED BITS HIGH (32)	RISBHG	RIE-f		HW										EC5D	7-364
ROTATE THEN INSERT SELECTED BITS LOW (32)	RISBLG	RIE-f		HW										EC51	7-365
ROTATE THEN OR SELECTED BITS (64)	ROSBG	RIE-f	C	GE										EC56	7-362
SEARCH STRING	SRST	RRE	C		¤[9]	A	SP	IC		G0			R_2	B25E	7-366
SEARCH STRING UNICODE	SRSTU	RRE	C	E3	¤[9]	A	SP	IC		G0		R_1	R_2	B9BE	7-368
SET ACCESS	SAR	RRE			¤[6]							U_1		B24E	7-370
SET ADDRESSING MODE (24)	SAM24	E		N3	¤[3,9]		SP		T					010C	7-370
SET ADDRESSING MODE (31)	SAM31	E		N3	¤[3,9]		SP		T					010D	7-370
SET ADDRESSING MODE (64)	SAM64	E		N	¤[3,9]				T					010E	7-370
SET PROGRAM MASK	SPM	RR	L											04	7-371
SHIFT LEFT DOUBLE (64)	SLDA	RS-a	C				SP		IF					8F	7-371
SHIFT LEFT DOUBLE LOGICAL (64)	SLDL	RS-a					SP							8D	7-372
SHIFT LEFT SINGLE (32)	SLA	RS-a	C						IF					8B	7-372
SHIFT LEFT SINGLE (32)	SLAK	RSY-a	C	DO					IF					EBDD	7-372
SHIFT LEFT SINGLE (64)	SLAG	RSY-a	C	N					IF					EB0B	7-372
SHIFT LEFT SINGLE LOGICAL (32)	SLL	RS-a												89	7-373

Figure 7-1. Summary of General Instructions (Part 9 of 14)

Name	Mnemonic	Characteristics										Opcode	Page
SHIFT LEFT SINGLE LOGICAL (32)	SLLK	RSY-a		DO								EBDF	7-373
SHIFT LEFT SINGLE LOGICAL (64)	SLLG	RSY-a		N								EB0D	7-373
SHIFT RIGHT DOUBLE (64)	SRDA	RS-a	C				SP					8E	7-374
SHIFT RIGHT DOUBLE LOGICAL (64)	SRDL	RS-a					SP					8C	7-374
SHIFT RIGHT SINGLE (32)	SRA	RS-a	C									8A	7-375
SHIFT RIGHT SINGLE (32)	SRAK	RSY-a	C	DO								EBDC	7-375
SHIFT RIGHT SINGLE (64)	SRAG	RSY-a	C	N								EB0A	7-375
SHIFT RIGHT SINGLE LOGICAL (32)	SRL	RS-a										88	7-376
SHIFT RIGHT SINGLE LOGICAL (32)	SRLK	RSY-a		DO								EBDE	7-376
SHIFT RIGHT SINGLE LOGICAL (64)	SRLG	RSY-a		N								EB0C	7-376
STORE (32)	ST	RX-a				A				ST	B_2	50	7-376
STORE (32)	STY	RXY-a		LD		A				ST	B_2	E350	7-377
STORE (64)	STG	RXY-a		N		A				ST	B_2	E324	7-377
STORE ACCESS MULTIPLE	STAM	RS-a				A	SP			ST	UB	9B	7-377
STORE ACCESS MULTIPLE	STAMY	RSY-a		LD		A	SP			ST	UB	EB9B	7-377
STORE CHARACTER	STC	RX-a				A				ST	B_2	42	7-378
STORE CHARACTER	STCY	RXY-a		LD		A				ST	B_2	E372	7-378
STORE CHARACTER HIGH (8)	STCH	RXY-a		HW		A				ST	B_2	E3C3	7-378
STORE CHARACTERS UNDER MASK (high)	STCMH	RSY-b		N	¤[9,11]	A				ST	B_2	EB2C	7-378
STORE CHARACTERS UNDER MASK (low)	STCM	RS-b				A				ST	B_2	BE	7-378
STORE CHARACTERS UNDER MASK (low)	STCMY	RSY-b		LD		A				ST	B_2	EB2D	7-378
STORE CLOCK	STCK	S	C		¤[8,9]	A		$		ST	B_2	B205	7-379
STORE CLOCK EXTENDED	STCKE	S	C		¤[8,9]	A		$		ST	B_2	B278	7-380
STORE CLOCK FAST	STCKF	S	C	SC	¤[8,9]	A				ST	B_2	B27C	7-379
STORE FACILITY LIST EXTENDED	STFLE	S	C	FL	¤[1]	A	SP	G0		ST	B_2	B2B0	7-382
STORE GUARDED STORAGE CONTROLS	STGSC	RXY-a		GF	¤[1]	A		SO		ST	B_2	E349	7-383
STORE HALFWORD (16)	STH	RX-a				A				ST	B_2	40	7-383
STORE HALFWORD (16)	STHY	RXY-a		LD		A				ST	B_2	E370	7-384
STORE HALFWORD HIGH (16)	STHH	RXY-a		HW		A				ST	B_2	E3C7	7-384
STORE HALFWORD RELATIVE LONG (16)	STHRL	RIL-b		GE		A*				ST		C47	7-384
STORE HIGH (32)	STFH	RXY-a		HW		A				ST	B_2	E3CB	7-384
STORE HIGH ON CONDITION	STOCFH	RSY-b		L2		A				ST	B_2	EBE1	7-386
STORE MULTIPLE (32)	STM	RS-a				A				ST	B_2	90	7-385
STORE MULTIPLE (32)	STMY	RSY-a		LD		A				ST	B_2	EB90	7-385
STORE MULTIPLE (64)	STMG	RSY-a		N		A				ST	B_2	EB24	7-385
STORE MULTIPLE HIGH (32)	STMH	RSY-a		N		A				ST	B_2	EB26	7-385
STORE ON CONDITION (32)	STOC	RSY-b		L1		A				ST	B_2	EBF3	7-385
STORE ON CONDITION (64)	STOCG	RSY-b		L1		A				ST	B_2	EBE3	7-385
STORE PAIR TO QUADWORD (64&64→128)	STPQ	RXY-a		N	¤[9]	A	SP			ST	B_2	E38E	7-386
STORE RELATIVE LONG (32)	STRL	RIL-b		GE		A*	SP			ST		C4F	7-377
STORE RELATIVE LONG (64)	STGRL	RIL-b		GE		A*	SP			ST		C4B	7-377
STORE REVERSED (16)	STRVH	RXY-a		N3		A				ST	B_2	E33F	7-387
STORE REVERSED (32)	STRV	RXY-a		N3		A				ST	B_2	E33E	7-387
STORE REVERSED (64)	STRVG	RXY-a		N		A				ST	B_2	E32F	7-387
SUBTRACT (32)	S	RX-a	C			A		IF			B_2	5B	7-388
SUBTRACT (32)	SR	RR	C					IF				1B	7-387
SUBTRACT (32)	SRK	RRF-a	C	DO				IF				B9F9	7-387
SUBTRACT (32)	SY	RXY-a	C	LD		A		IF			B_2	E35B	7-388
SUBTRACT (64)	SG	RXY-a	C	N		A		IF			B_2	E309	7-388
SUBTRACT (64)	SGR	RRE	C	N				IF				B909	7-387
SUBTRACT (64)	SGRK	RRF-a	C	DO				IF				B9E9	7-388
SUBTRACT (64←32)	SGF	RXY-a	C	N		A		IF			B_2	E319	7-388
SUBTRACT (64←32)	SGFR	RRE	C	N				IF				B919	7-387
SUBTRACT HALFWORD (32←16)	SH	RX-a	C			A		IF			B_2	4B	7-388
SUBTRACT HALFWORD (32←16)	SHY	RXY-a	C	LD		A		IF			B_2	E37B	7-388

Figure 7-1. Summary of General Instructions (Part 10 of 14)

Name	Mne-monic	Characteristics												Op-code	Page
SUBTRACT HALFWORD (64←16)	SGH	RXY-a	C	MI2		A			IF				B_2	E339	7-388
SUBTRACT HIGH (32)	SHHHR	RRF-a	C	HW					IF					B9C9	7-389
SUBTRACT HIGH (32)	SHHLR	RRF-a	C	HW					IF					B9D9	7-389
SUBTRACT LOGICAL (32)	SL	RX-a	C			A							B_2	5F	7-390
SUBTRACT LOGICAL (32)	SLR	RR	C											1F	7-389
SUBTRACT LOGICAL (32)	SLRK	RRF-a	C	DO										B9FB	7-389
SUBTRACT LOGICAL (32)	SLY	RXY-a	C	LD		A							B_2	E35F	7-390
SUBTRACT LOGICAL (64)	SLG	RXY-a	C	N		A							B_2	E30B	7-390
SUBTRACT LOGICAL (64)	SLGR	RRE	C	N										B90B	7-389
SUBTRACT LOGICAL (64)	SLGRK	RRF-a	C	DO										B9EB	7-389
SUBTRACT LOGICAL (64←32)	SLGF	RXY-a	C	N		A							B_2	E31B	7-390
SUBTRACT LOGICAL (64←32)	SLGFR	RRE	C	N										B91B	7-389
SUBTRACT LOGICAL HIGH (32)	SLHHHR	RRF-a	C	HW										B9CB	7-391
SUBTRACT LOGICAL HIGH (32)	SLHHLR	RRF-a	C	HW										B9DB	7-391
SUBTRACT LOGICAL IMMEDIATE (32)	SLFI	RIL-a	C	EI										C25	7-390
SUBTRACT LOGICAL IMMEDIATE (64←32)	SLGFI	RIL-a	C	EI										C24	7-390
SUBTRACT LOGICAL WITH BORROW (32)	SLB	RXY-a	C	N3		A							B_2	E399	7-391
SUBTRACT LOGICAL WITH BORROW (32)	SLBR	RRE	C	N3										B999	7-391
SUBTRACT LOGICAL WITH BORROW (64)	SLBG	RXY-a	C	N		A							B_2	E389	7-391
SUBTRACT LOGICAL WITH BORROW (64)	SLBGR	RRE	C	N										B989	7-391
SUPERVISOR CALL	SVC	I			¤[1]				¢					0A	7-392
TEST ADDRESSING MODE	TAM	E	C	N3	¤[9]									010B	7-392
TEST AND SET	TS	SI	C		¤[9]	A			$			ST	B_2	93	7-392
TEST UNDER MASK	TM	SI	C			A							B_1	91	7-393
TEST UNDER MASK	TMY	SIY	C	LD		A							B_1	EB51	7-393
TEST UNDER MASK (high high)	TMHH	RI-a	C	N										A72	7-393
TEST UNDER MASK (high low)	TMHL	RI-a	C	N										A73	7-393
TEST UNDER MASK (low high)	TMLH	RI-a	C	N										A70	7-393
TEST UNDER MASK (low low)	TMLL	RI-a	C	N										A71	7-393
TEST UNDER MASK HIGH	TMH	RI-a	C											A70	7-393
TEST UNDER MASK LOW	TML	RI-a	C											A71	7-393
TRANSACTION ABORT	TABORT	S		TX	¤[9]		SP	SO	$	EX				B2FC	7-394
TRANSACTION BEGIN (nonconstrained)	TBEGIN	SIL	C	TX	¤[9]	A	SP	SO	$	EX		ST		E560	7-395
TRANSACTION BEGIN (constrained)	TBEGINC	SIL	C	CX	¤[9]		SP	SO	$	EX				E561	7-399
TRANSACTION END	TEND	S	C	TX				SO	$	EX				B2F8	7-401
TRANSLATE	TR	SS-a			¤[9]	A						ST	B_1 B_2	DC	7-402
TRANSLATE AND TEST	TRT	SS-a	C		¤[9]	A				GM			B_1 B_2	DD	7-403
TRANSLATE AND TEST EXTENDED	TRTE	RRF-c	C	PE	¤[9]	A	SP	IC		G1			RM	B9BF	7-403
TRANSLATE AND TEST REVERSE	TRTR	SS-a	C	E3	¤[9]	A				GM			B_1 B_2	D0	7-408
TRANSLATE AND TEST REVERSE EXTENDED	TRTRE	RRF-c	C	PE	¤[9]	A	SP	IC		G1			RM	B9BD	7-404
TRANSLATE EXTENDED	TRE	RRE	C		¤[9]	A	SP	IC		G0		ST	R_1 R_2	B2A5	7-409
TRANSLATE ONE TO ONE	TROO	RRF-c	C	E2	¤[9]	A	SP	IC		GM		ST	RM R_2	B993	7-411
TRANSLATE ONE TO TWO	TROT	RRF-c	C	E2	¤[9]	A	SP	IC		GM		ST	RM R_2	B992	7-411
TRANSLATE TWO TO ONE	TRTO	RRF-c	C	E2	¤[9]	A	SP	IC		GM		ST	RM R_2	B991	7-411
TRANSLATE TWO TO TWO	TRTT	RRF-c	C	E2	¤[9]	A	SP	IC		GM		ST	RM R_2	B990	7-411
UNPACK	UNPK	SS-b			¤[9]	A						ST	B_1 B_2	F3	7-416
UNPACK ASCII	UNPKA	SS-a	C	E2	¤[9]	A	SP					ST	B_1 B_2	EA	7-417
UNPACK UNICODE	UNPKU	SS-a	C	E2	¤[9]	A	SP					ST	B_1 B_2	E2	7-418
UPDATE TREE	UPT	E	C		¤[9]	A	SP	II		GM	I4	ST		0102	7-419

Explanation:

¢ Causes serialization and checkpoint synchronization.

¢[1] Causes serialization and checkpoint synchronization when the M_1 and R_2 fields contain 1111 binary and 0000 binary, respectively. Causes only serialization when the fast-BCR-serialization facility is installed, and the M_1 and R_2 fields contain 1110 binary and 0000 binary, respectively.

$ Causes serialization.

Figure 7-1. Summary of General Instructions (Part 11 of 14)

Name	Mnemonic	Characteristics	Opcode	Page
£	Causes specific-operand serialization.			
£[1]	Causes specific-operand serialization when the interlocked-access facility 1 is installed and the storage operand is aligned on an integral boundary corresponding to its size.			
£[2]	Causes specific-operand serialization when the interlocked-access facility 2 is installed.			
¤[1]	Restricted from transactional execution.			
¤[2]	Restricted from transactional execution when R_2 nonzero and branch tracing is enabled.			
¤[3]	Restricted from transactional execution when mode tracing is enabled.			
¤[4]	Restricted from transactional execution when a monitor-event condition occurs.			
¤[5]	Model dependent whether the instruction is restricted from transactional execution.			
¤[6]	Restricted from transactional execution when the effective allow-AR-modification control is zero.			
¤[7]	Restricted from transactional execution when the effective allow-floating-point-operation control is zero.			
¤[8]	May be restricted from transactional execution depending on machine conditions.			
¤[9]	Restricted in the constrained transactional-execution mode; a transaction-constraint program exception may be recognized. For STCMH, the instruction is restricted only when the M_3 field is zero.			
¤[10]	Restricted to forward branches in the constrained transactional-execution mode.			
¤[11]	For PFD and PFDRL, it is model dependent whether the instruction is restricted from transactional execution when the code in the M_1 field is 6 or 7; for STCMH, it is model dependent whether the instruction is restricted when the M_3 field is zero, and the code in the R_1 field is 6 or 7.			
¤[12]	Restricted from transactional execution when a guarded-storage event is recognized. When in the transactional-execution mode, the transaction is aborted, and the guarded-storage event is processed.			
*	PER zero-address-detection not recognized.			
32H	32 high-order bits of a register.			
32L	32 low-order bits of a register.			
A	Access exceptions for logical addresses.			
AI	Access exceptions for instruction address.			
B	PER branch event. (For LGG and LLGFSG, the PER branch event is only recognized coincident with a guarded-storage event.)			
B_1	B_1 field designates an access register in the access-register mode.			
B_2	B_2 field designates an access register in the access-register mode.			
BP	B_2 field designates an access register when PSW bits 16 and 17 have the value 01 binary.			
C	Condition code is set.			
CS	Compare-and-swap-and-store facility.			
CX	Constrained transactional-execution facility			
Dc	Compare-and-trap data exception			
Dg	General-operand data exception.			
DO	Distinct-operands facility			
E	E instruction format.			
EH	Execution-hint facility			
EI	Extended-immediate facility.			
ET	Extract-CPU-time facility.			
EX	Execute exception.			
E2	Extended-translation facility 2.			
E3	Extended-translation facility 3.			
FC	Designation of access registers depends on the function code of the instruction.			
FL	Store-facility-list-extended facility.			
G0	Instruction execution includes the implied use of general register 0.			
G1	Instruction execution includes the implied use of general register 1.			
GE	General-instructions-extension facility.			

Figure 7-1. Summary of General Instructions (Part 12 of 14)

Name	Mnemonic	Characteristics	Opcode	Page
GM	Instruction execution includes the implied use of multiple general registers: • General registers 1, 2, and 3 for COMPARE AND FORM CODEWORD. • General registers 0 and 1 for CIPHER MESSAGE, CIPHER MESSAGE WITH AUTHENTICATION, CIPHER MESSAGE WITH CHAINING, CIPHER MESSAGE WITH CIPHER FEEDBACK, CIPHER MESSAGE WITH COUNTER, CIPHER MESSAGE WITH OUTPUT FEEDBACK, COMPARE AND SWAP AND STORE, COMPARE UNTIL SUBSTRING EQUAL, COMPRESSION CALL, COMPUTE INTERMEDIATE MESSAGE DIGEST, COMPUTE LAST MESSAGE DIGEST, COMPUTE MESSAGE AUTHENTICATION CODE, EXTRACT CPU TIME, PERFORM CRYPTOGRAPHIC COMPUTATION, PERFORM LOCKED OPERATION, PERFORM RANDOM NUMBER OPERATION, TRANSLATE ONE TO ONE, TRANSLATE ONE TO TWO, TRANSLATE TWO TO ONE, and TRANSLATE TWO TO TWO • General registers 1 and 2 for TRANSLATE AND TEST and TRANSLATE AND TEST REVERSE. • General registers 0-5 for UPDATE TREE. •			
GS	Guarded-storage facility.			
HW	High-word facility			
I	I instruction format.			
I1	Access register 1 is implicitly designated in the access-register mode.			
I2	Access register 2 is implicitly designated in the access-register mode.			
I4	Access register 4 is implicitly designated in the access-register mode.			
IA	Interlocked-access facility 1			
IC	Condition code alternative to interruptible instruction			
IE	IE instruction format.			
IF	Fixed-point-overflow exception.			
II	Interruptible instruction.			
IK	Fixed-point-divide exception.			
L	New condition code is loaded.			
L1	Load/store-on-condition facility 1.			
L2	Load/store-on-condition facility 2.			
LD	Long-displacement facility.			
LT	Load-and-trap facility			
LZ	Load-and-zero-rightmost-byte facility.			
ME	Monitor event.			
MI1	Miscellaneous-instruction-extensions facility 1			
MI2	Miscellaneous-instruction-extensions facility 2			
MII	MII instruction format.			
MS	Message-security assist.			
M4	Message-security-assist extension 4.			
M5	Message-security-assist extension 5.			
M8	Message-security-assist extension 8.			
N	Instruction is new in z/Architecture as compared to ESA/390.			
N3	Instruction is new in z/Architecture and has been added to ESA/390. Any RSY or RXY instructions still use the RSE or RXE format and 12-bit displacements in ESA/390. RSY- and RXY-format instructions having the N3 facility code may use long displacements in the ESA/390-compatibility mode.			
PA	Processor-assist facility.			
PE	Parsing-enhancement facility.			
PK	Population-count facility			
R_1	R_1 field designates an access register in the access-register mode.			
R_2	R_2 field designates an access register in the access-register mode.			
R_3	R_3 field designates an access register in the access-register mode.			
RI	RI instruction format.			
RIE	RIE instruction format.			
RIL	RIL instruction format.			

Figure 7-1. Summary of General Instructions (Part 13 of 14)

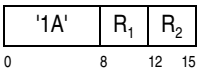
Name	Mnemonic	Characteristics	Op-code	Page
RM	R_1 field designates an access register in the access-register mode, and access-register 1 also is used in the access-register mode.			
RR	RR instruction format.			
RRE	RRE instruction format.			
RRF	RRF instruction format.			
RRS	RRS instruction format.			
RS	RS instruction format.			
RSI	RSI instruction format.			
RSY	RSY instruction format.			
RX	RX instruction format.			
RXY	RXY instruction format.			
S	S instruction format.			
SC	Store-clock-fast facility.			
SI	SI instruction format.			
SIL	SIL instruction format.			
SIY	SIY instruction format.			
SMI	SMI instruction format.			
SO	Special-operation exception.			
SP	Specification exception.			
SS	SS instruction format.			
SSF	SSF instruction format.			
ST	PER storage-alteration event. (For LGG and LLGFSG, the PER storage-alteration event is only recognized coincident with a guarded-storage event.)			
T	Trace exceptions (includes trace table, addressing, and low-address protection).			
TX	Transactional-execution facility			
U_1	R_1 field designates an access register unconditionally.			
U_2	R_2 field designates an access register unconditionally.			
UB	R_1 and R_3 fields designate access registers unconditionally, and B_2 field designates an access register in the access-register mode.			
VF	Vector facility for z/Architecture			
XX	Execute-extension facility.			

Figure 7-1. Summary of General Instructions (Part 14 of 14)

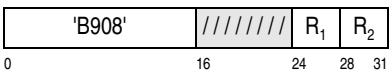
ADD

Register-and-register formats:

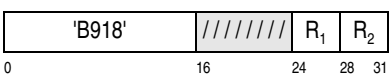
AR R_1,R_2 [RR]



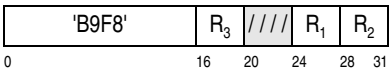
AGR R_1,R_2 [RRE]



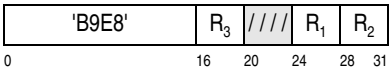
AGFR R_1,R_2 [RRE]



ARK R_1,R_2,R_3 [RRF-a]

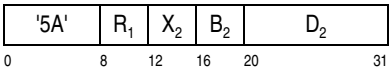


AGRK R_1,R_2,R_3 [RRF-a]

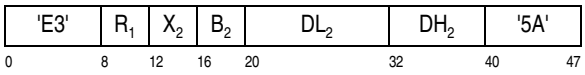


Register-and-storage formats:

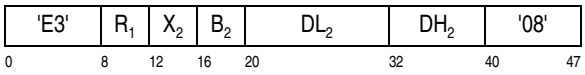
A $R_1,D_2(X_2,B_2)$ [RX-a]



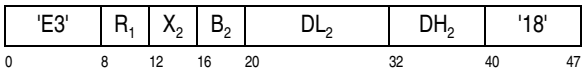
AY $R_1,D_2(X_2,B_2)$ [RXY-a]



AG $R_1,D_2(X_2,B_2)$ [RXY-a]



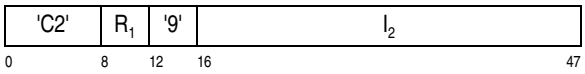
AGF $R_1,D_2(X_2,B_2)$ [RXY-a]



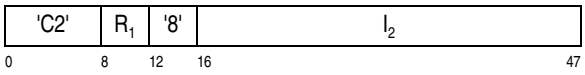
ADD IMMEDIATE

Register-and-immediate formats:

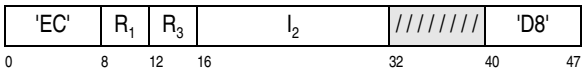
AFI R_1,I_2 [RIL-a]



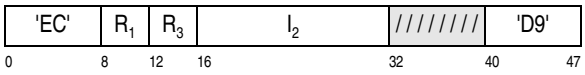
AGFI R_1,I_2 [RIL-a]



AHIK R_1,R_3,I_2 [RIE-d]

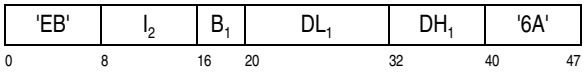


AGHIK R_1,R_3,I_2 [RIE-d]

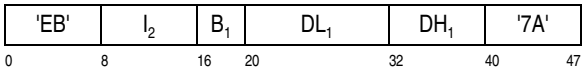


Storage-and-immediate formats:

ASI $D_1(B_1),I_2$ [SIY]



AGSI $D_1(B_1),I_2$ [SIY]



For ADD (A, AG, AGF, AGFR, AGR, AR, and AY) and for ADD IMMEDIATE (AFI, AGFI, AGSI, and ASI), the second operand is added to the first operand, and the sum is placed at the first-operand location. For ADD (AGRK and ARK) and for ADD IMMEDIATE (AGHIK and AHIK), the second operand is added to the third operand, and the sum is placed at the first-operand location.

For ADD (A, AR, ARK, and AY) and for ADD IMMEDIATE (AFI), the operands and the sum are treated as 32-bit signed binary integers. For ADD (AG, AGR, and AGRK), they are treated as 64-bit signed binary integers. For ADD (AGFR, AGF) and for ADD IMMEDIATE (AGFI), the second operand is treated as a 32-bit signed binary integer, and the first operand and the sum are treated as 64-bit signed binary integers. For ADD IMMEDIATE (ASI), the second operand is treated as an 8-bit signed binary integer, and the first operand and the sum are treated as 32-bit signed binary integers. For ADD IMMEDIATE (AGSI), the second operand is treated as an 8-bit signed binary integer, and the first operand and the sum are treated as 64-bit signed binary integers. For ADD IMMEDIATE (AHIK), the first and third operands are treated as 32-bit signed binary integers, and the second operand is treated as a 16-bit signed binary integer. For ADD IMMEDIATE (AGHIK), the first and third operands are treated as 64-bit signed binary integers, and the second operand is treated as a 16-bit signed binary integer.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

When the interlocked-access facility 1 is installed and the first operand of ADD IMMEDIATE (ASI, AGSI) is aligned on an integral boundary corresponding to its size, then the fetch and store of the first operand are performed as an interlocked update as observed by other CPUs, and a specific-operand-serialization operation is performed. When the interlocked-access facility 1 is not installed, or when the first operand of ADD IMMEDIATE (ASI, AGSI) is not aligned on an integral boundary corresponding to its size, then the fetch and store of the operand are not performed as an interlocked update.

The displacement for A is treated as a 12-bit unsigned binary integer. The displacement for AY, AG, AGF, AGSI and ASI, is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

Program Exceptions:

- Access (fetch and store, operand 1 of AGSI and ASI only; fetch, operand 2 of A, AY, AG, and AGF only)
- Fixed-point overflow
- Operation (AY, if the long-displacement facility is not installed; AFI and AGFI, if the extended-immediate facility is not installed; AGSI and ASI, if the general-instructions-extension facility is not installed; ARK, AGRK, AHIK, and AGHIK, if the distinct-operands facility is not installed)

Programming Notes:

1. Accesses to the first operand of ADD IMMEDIATE (AGSI and ASI) consist in fetching a first-operand from storage and subsequently storing the updated value.

2. When the interlocked-access facility 1 is not installed, or when the first operand is not aligned on an integral boundary corresponding to its size, the fetch and store accesses to the first operand do not necessarily occur one immediately after the other. Under such conditions, ADD IMMEDIATE (AGSI and ASI) cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown in "Multiprogramming and Multiprocessing Examples" in Appendix A, "Number Representation and Instruction-Use Examples."

   When the interlocked-access facility 1 is installed and the first operand is aligned on an integral boundary corresponding to its size, the operand is accessed using a block-concurrent interlocked update.

3. For certain programming languages which ignore overflow conditions on arithmetic operations, the setting of condition code 3 obscures the sign of the result. However, for ADD IMMEDIATE, the sign of the I_2 field (which is known at the time of code generation) may be used in setting a branch mask which will accurately determine the resulting sign, as shown below.

	Result to be Tested		
I_2 Value	**Positive**	**Negative**	**Zero**
Positive	0010	0101[a]	1000
Negative	0011[b]	0100	1000[c]
Zero	0010	0100	1000

Explanation:

a If the first operand was positive, then a mask of 0001 covers that case (overflow). If the first operand was negative, then a mask of 0100 is appropriate. Combining the two gives a mask of 0101.

b Similarly, a mask of 0010 (first operand was positive) and mask of 0001 (first operand was negative) yields a mask of 0011 which covers both cases.

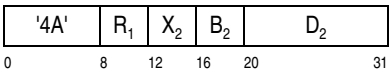
c A separate test for zero may be required when the I_2 field contains 80000000 hex.

Figure 7-2. Branch Masks to Determine Resulting Sign for ADD IMMEDIATE

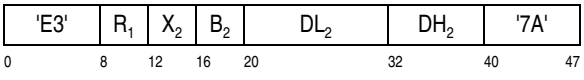
The technique described above is also applicable to ADD HALFWORD IMMEDIATE.

ADD HALFWORD

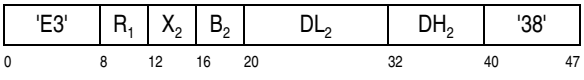
AH $R_1,D_2(X_2,B_2)$ [RX-a]



AHY $R_1,D_2(X_2,B_2)$ [RXY-a]

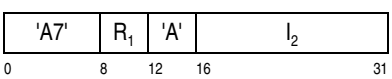


AGH $R_1,D_2(X_2,B_2)$ [RXY-a]

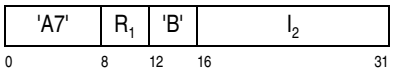


ADD HALFWORD IMMEDIATE

AHI R_1,I_2 [RI-a]



AGHI R_1,I_2 [RI-a]



The second operand is added to the first operand, and the sum is placed at the first-operand location. The second operand is two bytes in length and is treated as a 16-bit signed binary integer. For ADD HALFWORD (AH, AHY) and ADD HALFWORD IMMEDIATE (AHI), the first operand and the sum are treated as 32-bit signed binary integers. For ADD HALFWORD (AGH) and ADD HALFWORD IMMEDIATE (AGHI), they are treated as 64-bit signed binary integers.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

The displacement for AH is treated as a 12-bit unsigned binary integer. The displacement for AGH and AHY is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

Program Exceptions:

- Access (fetch, operand 2 of AH, AGH, AHY)
- Fixed-point overflow
- Operation (AHY, if the long-displacement facility is not installed; AGH, if the miscellaneous-instruction-extensions facility 2 is not installed)

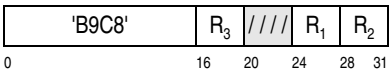
Programming Notes:

1. An example of the use of the ADD HALFWORD instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

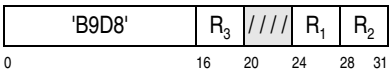
2. See programming note 3 on page 7-28 for ADD IMMEDIATE regarding the implications of ignoring overflow.

ADD HIGH

AHHHR R_1,R_2,R_3 [RRF-a]



AHHLR R_1,R_2,R_3 [RRF-a]



The second operand is added to the third operand, and the sum is placed at the first-operand location. The operands and the sum are treated as 32-bit signed binary integers. The first and second operands are in bits 0-31 of general registers R_1 and R_2, respectively; bits 32-63 of general register R_1 are unchanged, and bits 32-63 of general register R_2 are ignored. For AHHHR, the third operand is in bits 0-31 of general register R_3; bits 32-63 of the register are ignored. For AHHLR, the third operand is in bits 32-63 of general register R_3; bits 0-31 of the register are ignored.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

Resulting Condition Code:

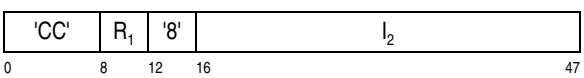
0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

Program Exceptions:

- Fixed-point overflow
- Operation (if the high-word facility is not installed)

ADD IMMEDIATE HIGH

AIH R_1,I_2 [RIL-a]



The second operand is added to the first operand, and the sum is placed at the first-operand location.

The operands and the sum are treated as 32-bit signed binary integers. The first operand is in bits 0-31 of general register R_1; bits 32-63 of the register are unchanged.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

Resulting Condition Code:

0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

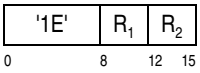
Program Exceptions:

- Fixed-point overflow
- Operation (if the high-word facility is not installed)

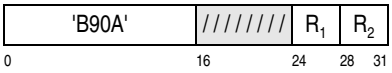
ADD LOGICAL

Register-and-register formats:

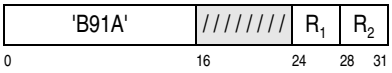
ALR R_1,R_2 [RR]



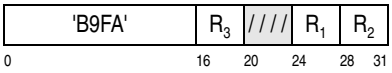
ALGR R_1,R_2 [RRE]



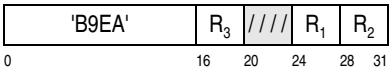
ALGFR R_1,R_2 [RRE]



ALRK R_1,R_2,R_3 [RRF-a]

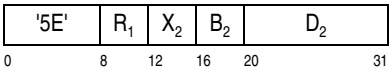


ALGRK R_1,R_2,R_3 [RRF-a]

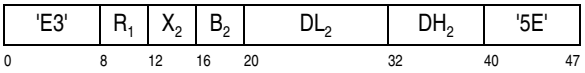


Register-and-storage formats:

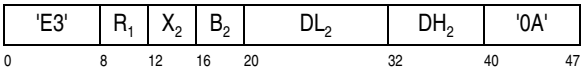
AL $R_1,D_2(X_2,B_2)$ [RX-a]



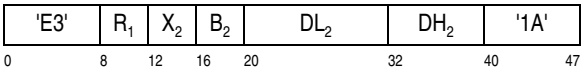
ALY $R_1,D_2(X_2,B_2)$ [RXY-a]



ALG $R_1,D_2(X_2,B_2)$ [RXY-a]

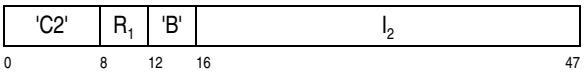


ALGF $R_1,D_2(X_2,B_2)$ [RXY-a]

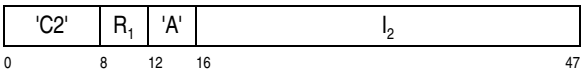


ADD LOGICAL IMMEDIATE

ALFI R_1,I_2 [RIL-a]



ALGFI R_1,I_2 [RIL-a]



For ADD LOGICAL (AL, ALG, ALGF, ALGFR, ALGR, ALR, and ALY) and for ADD LOGICAL IMMEDIATE (ALGFI and ALFI), the second operand is added to the first operand, and the sum is placed at the first-operand location. For ADD LOGICAL (ALGRK and ALRK), the second operand is added to the third operand, and the sum is placed at the first-operand location.

For ADD LOGICAL (AL, ALR, ALRK, and ALY) and for ADD LOGICAL IMMEDIATE (ALFI), the operands and the sum are treated as 32-bit unsigned binary integers. For ADD LOGICAL (ALG, ALGR, and ALGRK), they are treated as 64-bit unsigned binary integers. For ADD LOGICAL (ALGFR, ALGF) and for ADD LOGICAL IMMEDIATE (ALGFI), the second operand is treated as a 32-bit unsigned binary integer, and the first operand and the sum are treated as 64-bit unsigned binary integers.

The displacement for AL is treated as a 12-bit unsigned binary integer. The displacement for ALY, ALG, and ALGF is treated as a 20-bit signed binary integer.

Resulting Condition Code:

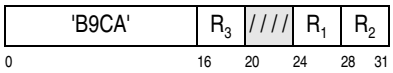
0 Result zero; no carry
1 Result not zero; no carry
2 Result zero; carry
3 Result not zero; carry

Program Exceptions:

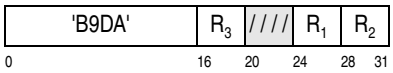
- Access (fetch, operand 2 of AL, ALY, ALG, and ALGF only)
- Operation (ALY, if the long-displacement facility is not installed; ALFI and ALGFI, if the extended-immediate facility is not installed; ALRK and ALGRK, if the distinct-operands facility is not installed)

ADD LOGICAL HIGH

ALHHHR R_1,R_2,R_3 [RRF-a]



ALHHLR R_1,R_2,R_3 [RRF-a]



The second operand is added to the third operand, and the sum is placed at the first-operand location. The operands and the sum are treated as 32-bit unsigned binary integers. The first and second operands are in bits 0-31 of general registers R_1 and R_2, respectively; bits 32-63 of general register R_1 are unchanged, and bits 32-63 of general register R_2 are ignored. For ALHHHR, the third operand is in bits 0-31 of general register R_3; bits 32-63 of the register are ignored. For ALHHLR, the third operand is in bits 32-63 of general register R_3; bits 0-31 of the register are ignored.

Resulting Condition Code:

0 Result zero; no carry
1 Result not zero; no carry
2 Result zero; carry
3 Result not zero; carry

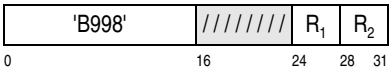
Program Exceptions:

- Operation (if the high-word facility is not installed)

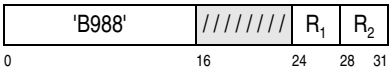
ADD LOGICAL WITH CARRY

Register-and-register formats:

ALCR R_1,R_2 [RRE]

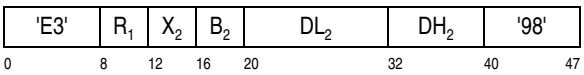


ALCGR R_1,R_2 [RRE]

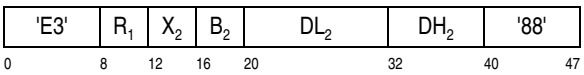


Register-and-storage formats:

ALC $R_1,D_2(X_2,B_2)$ [RXY-a]



ALCG $R_1,D_2(X_2,B_2)$ [RXY-a]



The second operand and the carry are added to the first operand, and the sum is placed at the first-operand location. For ADD LOGICAL WITH CARRY (ALCR, ALC), the operands, the carry, and the sum are treated as 32-bit unsigned binary integers. For ADD LOGICAL WITH CARRY (ALCGR, ALCG), they are treated as 64-bit unsigned binary integers.

Resulting Condition Code:

0 Result zero; no carry
1 Result not zero; no carry
2 Result zero; carry
3 Result not zero; carry

Program Exceptions:

- Access (fetch, operand 2 of ALC and ALCG only)

Programming Notes:

1. A carry is represented by a one value of bit 18 of the current PSW. Bit 18 is the leftmost bit of the two-bit condition code in the PSW. Bit 18 is set to one by an execution of an ADD LOGICAL or

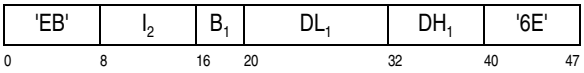
ADD LOGICAL WITH CARRY instruction that produces a carry out of bit position 0 of the result.

2. ADD and ADD LOGICAL may provide better performance than ADD LOGICAL WITH CARRY, depending on the model.

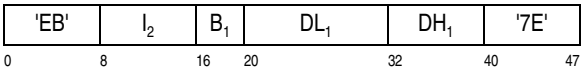
ADD LOGICAL WITH SIGNED IMMEDIATE

Storage-and-immediate formats:

ALSI $D_1(B_1),I_2$ [SIY]

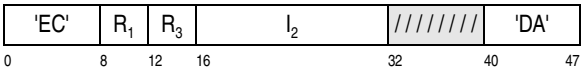


ALGSI $D_1(B_1),I_2$ [SIY]

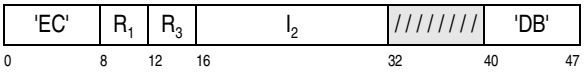


Register-and-immediate formats:

ALHSIK R_1,R_3,I_2 [RIE-d]



ALGHSIK R_1,R_3,I_2 [RIE-d]



For ALGSI and ALSI, the second operand is added to the first operand, and the sum is placed at the first-operand location. For ALGHSIK and ALHSIK, the second operand is added to the third operand, and the sum is placed at the first-operand location.

For ALSI, the first operand and the sum are treated as 32-bit unsigned binary integers. For ALGSI, the first operand and the sum are treated as 64-bit unsigned binary integers. For both ALSI and ALGSI, the second operand is treated as an 8-bit signed binary integer.

For ALHSIK, the first and third operands are treated as 32-bit unsigned binary integers. For ALGHSIK, the first and third operands are treated as 64-bit unsigned binary integers. For both ALGHSIK and ALHSIK, the second operand is treated as a 16-bit signed binary integer.

When the interlocked-access facility 1 is installed and the first operand of ALGSI or ALSI is aligned on an integral boundary corresponding to its size, then the fetch and store of the first operand is performed as an interlocked update as observed by other CPUs, and a specific-operand-serialization operation is performed. When the interlocked-access facility 1 is not installed, or when the first operand of ADD LOGICAL WITH SIGNED IMMEDIATE (ALSI, ALGSI) is not aligned on an integral boundary corresponding to its size, then the fetch and store of the operand are not performed as an interlocked update.

When the second operand contains a negative value, the condition code is set as though a SUBTRACT LOGICAL operation was performed. Condition code 0 is never set when the second operand is negative.

The displacement is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero; no carry
1 Result not zero; no carry
2 Result zero; carry
3 Result not zero; carry

Program Exceptions:

- Access (fetch and store, operand 1)
- Operation (ALGSI and ALSI, if the general-instructions-extension facility is not installed; ALGHSIK, and ALHSIK, if the distinct-operands facility is not installed)

Programming Notes:

1. When the second operand contains a negative value, the condition-code setting can also be interpreted as indicating the presence or absence of a borrow, as follows:

   1 Result not zero; borrow
   2 Result zero; no borrow
   3 Result not zero; no borrow

2. Accesses to the first operand of consist in fetching a first-operand from storage and subsequently storing the updated value.

   When the interlocked-access facility 1 is not installed, or when the first operand is not aligned on an integral boundary corresponding to its size, the fetch and store accesses to the first

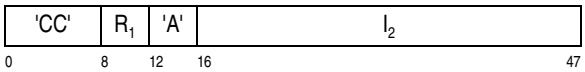
operand do not necessarily occur one immediately after the other. Under such conditions, ADD LOGICAL WITH SIGNED IMMEDIATE cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown in "Multiprogramming and Multiprocessing Examples" in Appendix A, "Number Representation and Instruction-Use Examples."

When the interlocked-access facility 1 is installed and the first operand is aligned on an integral boundary corresponding to its size, the operand is accessed using a block-concurrent interlocked update.

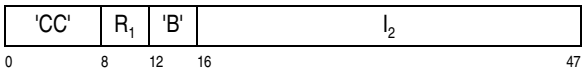
3. When the second operand is negative, the instruction effectively performs a subtraction operation.

ADD LOGICAL WITH SIGNED IMMEDIATE HIGH

ALSIH R_1,I_2 [RIL-a]



ALSIHN R_1,I_2 [RIL-a]



The second operand is added to the first operand, and the sum is placed at the first-operand location. The first operand and the sum are treated as 32-bit unsigned binary integers. The second operand is treated as a 32-bit signed binary integer. The first operand is in bits 0-31 of general register R_1; bits 32-63 of the register are unchanged.

Resulting Condition Code:

For ALSIH, the code is set as follows:

0 Result zero; no carry
1 Result not zero; no carry
2 Result zero; carry
3 Result not zero; carry

For ALSIHN, the code remains unchanged.

Program Exceptions:

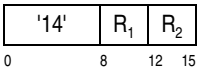
- Operation (if the high-word facility is not installed)

Programming Note: See programming notes 1 and 3 for ADD LOGICAL WITH SIGNED IMMEDIATE.

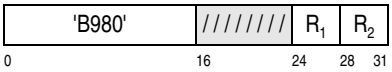
AND

Register-and-register formats:

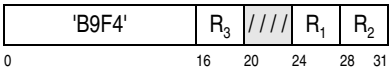
NR R_1,R_2 [RR]



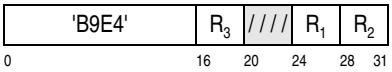
NGR R_1,R_2 [RRE]



NRK R_1,R_2,R_3 [RRF-a]

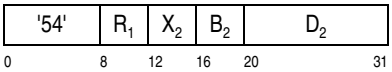


NGRK R_1,R_2,R_3 [RRF-a]

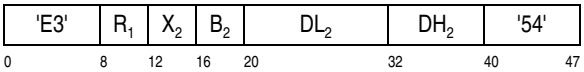


Register-and-storage formats:

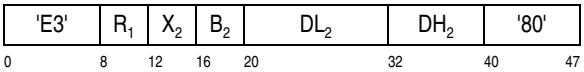
N $R_1,D_2(X_2,B_2)$ [RX-a]



NY $R_1,D_2(X_2,B_2)$ [RXY-a]

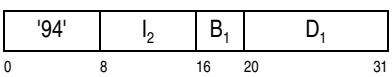


NG $R_1,D_2(X_2,B_2)$ [RXY-a]

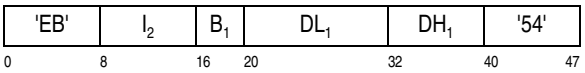


Storage-and-immediate formats:

NI $D_1(B_1),I_2$ [SI]

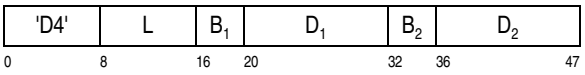


NIY $D_1(B_1),I_2$ [SIY]



Storage-and-storage format:

NC $D_1(L,B_1),D_2(B_2)$ [SS-a]



For N, NC, NG, NGR, NI, NIY, NR, and NY, the AND of the first and second operands is placed at the first-operand location. For NGRK and NRK, the AND of the second and third operands is placed at the first-operand location.

The connective AND is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit positions in both operands contain ones; otherwise, the result bit is set to zero.

For AND (NC), each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes.

For AND (NI, NIY), the first operand is one byte in length, and only one byte is stored. When the interlocked-access facility 2 is installed, the update of the first operand appears to be an interlocked-update reference as observed by other CPUs and channel programs, and a specific-operand-serialization operation is performed.

For AND (N, NR, NRK, and NY), the operands are 32 bits, and for AND (NG, NGR, and NGRK), they are 64 bits.

The displacements for N, NI, and both operands of NC are treated as 12-bit unsigned binary integers. The displacement for NY, NIY, and NG is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero
1 Result not zero
2 --
3 --

Program Exceptions:

- Access (fetch, operand 2, N, NY, NG, and NC; fetch and store, operand 1, NI, NIY, and NC)
- Operation (NY and NIY, if the long-displacement facility is not installed; NGRK and NRK, if the distinct-operands facility is not installed)
- Transaction constraint (NC)

Programming Notes:

1. An example of the use of the AND instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

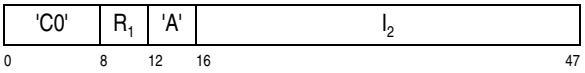
2. The AND instruction may be used to set a bit to zero.

3. Accesses to the first operand of AND (NC) – and, when the interlocked-access facility 2 is not installed, accesses to the first operand of AND (NI, NIY) – consist in fetching a first-operand byte from storage and subsequently storing the updated value. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, these instructions cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for OR (OI) in “Multiprogramming and Multiprocessing Examples” on page A-45.

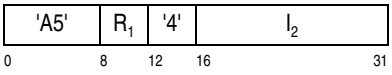
   When the interlocked-access facility 2 is installed, AND (NI, NIY) can be safely used to update a location in storage, even if the possibility exists that another CPU or a channel program may also be updating the location.

AND IMMEDIATE

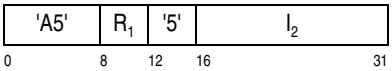
NIHF R_1,I_2 [RIL-a]



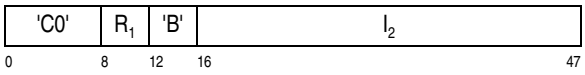
NIHH R_1,I_2 [RI-a]



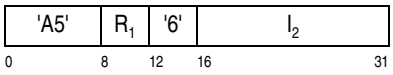
NIHL R_1,I_2 [RI-a]



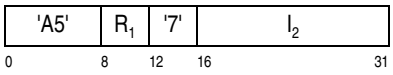
NILF R_1,I_2 [RIL-a]



NILH R_1,I_2 [RI-a]



NILL R_1,I_2 [RI-a]



The second operand is ANDed with bits of the first operand, and the result replaces those bits of the first operand. The remainder of the first operand remains unchanged.

For each instruction, the bits of the first operand that are ANDed with the second operand and then replaced are as follows:

Instruction	Bits ANDed and Replaced
NIHF	0-31
NIHH	0-15
NIHL	16-31
NILF	32-63
NILH	32-47
NILL	48-63

The connective AND is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit positions in both operands contain ones; otherwise, the result bit is set to zero.

Resulting Condition Code:

0 Result is zero
1 Result is not zero
2 --
3 --

Program Exceptions:

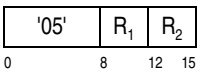
- Operation (NIHF, NILF, if the extended-immediate facility is not installed)

Programming Note: The setting of the condition code is based only on the bits that are ANDed and replaced.

BRANCH AND LINK

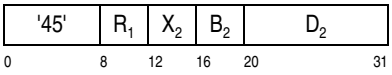
Register-and-register format:

BALR R_1,R_2 [RR]



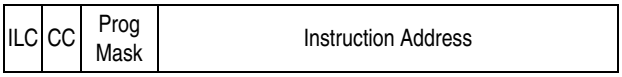
Register-and-storage format:

BAL $R_1,D_2(X_2,B_2)$ [RX-a]



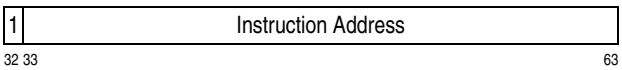
Information from the current PSW, including the updated instruction address, is saved as link information at the first-operand location. Subsequently, the instruction address in the PSW is replaced by the branch address.

The link information in the 24-bit addressing mode consists of the instruction-length code (ILC), the condition code (CC), the program-mask bits, and the rightmost 24 bits of the updated instruction address, arranged in bit positions 32-63 of the first-operand location in the following format:



When BRANCH AND LINK is not the target of an execute-type instruction, the ILC is set to 1 for BALR or 2 for BAL. When BRANCH AND LINK is the target of an EXECUTE instruction, the ILC is set to 2. When BRANCH AND LINK is the target of an EXECUTE RELATIVE LONG instruction, the ILC is set to 3.

The link information in the 31-bit addressing mode consists of bit 32 of the PSW, the basic-addressing-mode bit (always a one) and the rightmost 31 bits of the updated instruction address, arranged in bit positions 32-63 of the first-operand location in the following format:



In the 24-bit or 31-bit addressing mode, bits 0-31 of the first-operand location remain unchanged.

The link information in the 64-bit addressing mode consists of the updated instruction address, placed in bit positions 0-63 of the first-operand location.

In the RX format, the second-operand address is used as the branch address. In the RR format, the contents of general register R_2 are used to generate the branch address; however, when the R_2 field is zero, the operation is performed without branching. The branch address is computed before general register R_1 is changed.

Condition Code: The code remains unchanged.

Program Exceptions:

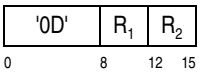
- Trace (R_2 field nonzero, BALR only)
- Transaction constraint

Programming Notes:

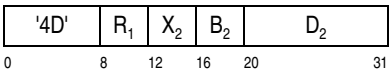
1. An example of the use of the BRANCH AND LINK instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. When the R_2 field in the RR format is zero, the link information is loaded without branching.
3. The BRANCH AND LINK instruction (BAL and BALR) is provided for compatibility purposes. It is recommended that, where possible, the BRANCH AND SAVE instruction (BAS and BASR), BRANCH RELATIVE AND SAVE, or BRANCH RELATIVE AND SAVE LONG be used and BRANCH AND LINK avoided, since the latter places nonzero information in bit positions 32-39 of the link register in the 24-bit addressing mode, which may lead to problems. Additionally, in the 24-bit addressing mode, BRANCH AND LINK may be slower than the other instructions because BRANCH AND LINK must construct the ILC, condition code, and program mask to be placed in bit positions 32-39 of the link register.
4. The condition-code and program-mask information, which is provided in the leftmost byte of the link information only in the 24-bit addressing mode, can be obtained in any addressing mode by means of the INSERT PROGRAM MASK instruction.

BRANCH AND SAVE

BASR R_1,R_2 [RR]



BAS $R_1,D_2(X_2,B_2)$ [RX-a]



Information from the current PSW, including the updated instruction address, is saved as link information at the first-operand location. Subsequently, the instruction address in the PSW is replaced by the branch address.

In the 24-bit or 31-bit addressing mode, the link information is bits 32 and 97-127 of the PSW, consisting of the basic-addressing-mode bit and the rightmost 31 bits of the updated instruction address. The link information is placed in bit positions 32 and 33-63, respectively, of the first-operand location, and bits 0-31 of the location remain unchanged.

In the 64-bit addressing mode, the link information consists of the updated instruction address, placed in bit positions 0-63 of the first-operand location.

In the RX format, the second-operand address is used as the branch address. In the RR format, the contents of general register R_2 are used to generate the branch address; however, when the R_2 field is zero, the operation is performed without branching. The branch address is computed before general register R_1 is changed.

Condition Code: The code remains unchanged.

Program Exceptions:

- Trace (R_2 field nonzero, BASR only)
- Transaction constraint

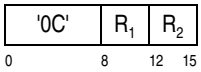
Programming Notes:

1. An example of the use of the BRANCH AND SAVE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. The BRANCH AND SAVE instruction (BAS and BASR) is intended to be used for linkage to pro-

grams known to be in the same addressing mode as the caller. This instruction should be used in place of the BRANCH AND LINK instruction (BAL and BALR). See the programming notes on pages 5-15 and 5-18 in the section “Subroutine Linkage without the Linkage Stack” for a detailed discussion of this and other linkage instructions. See also the programming note under BRANCH AND LINK for a discussion of the advantages of the BRANCH AND SAVE instruction.

BRANCH AND SAVE AND SET MODE

BASSM R_1,R_2 [RR]



Information from the current PSW, including the updated instruction address, is saved as link information at the first-operand location. Subsequently, if the R_2 field is nonzero, the addressing-mode bits and instruction address in the PSW are replaced as specified by the second operand.

In the 24-bit or 31-bit addressing mode, the link information is bits 32 and 97-127 of the PSW, consisting of the basic-addressing-mode bit and the rightmost 31 bits of the updated instruction address. The link information is placed in bit positions 32 and 33-63, respectively, of the first-operand location, and bits 0-31 of the location remain unchanged. In the 64-bit addressing mode, the link information is the updated instruction address in bits 64-126 of the PSW with a one appended on the right, placed in bit positions 0-63 of the first-operand location.

The contents of general register R_2 specify the new addressing mode and designate the branch address; however, when the R_2 field is zero, the operation is performed without branching and without setting either addressing-mode bit.

When the contents of general register R_2 are used and bit 63 of the register is zero, bit 31 of the current PSW, the extended-addressing-mode bit, is set to zero, bit 32 of the register specifies the new basic addressing mode and replaces bit 32 of the PSW, and the branch address is generated from the contents of the register under the control of the new addressing mode. The branch address replaces the instruction address in the PSW.

In the z/Architecture architectural mode, when the contents of general register R_2 are used and bit 63 of the register is one, the following occurs. Bits 31 and 32 of the current PSW are set to one, the branch address is generated from the contents of the register, except with bit 63 of the register treated as a zero, under the control of the new extended addressing mode, and the branch address replaces the instruction address in the PSW. Bit 63 of the register remains one. However, if R_2 is the same as R_1, the results in the designated general register are as specified for the R_1 register.

The new value for the PSW is computed before general register R_1 is changed.

In the ESA/390-compatibility mode, when the R_2 field is nonzero and bit 63 of general register R_2 is one, it is unpredictable which of the following occurs:

- The instruction completes by setting PSW bit 31 to one, and either (a) an early specification exception is recognized immediately after the PSW becomes active due to bit 31 of the PSW being one, or (b) the CPU enters the 64-bit addressing mode as described above for the z/Architecture architectural mode.
- The instruction completes by placing the branch address, including bit 63, into the instruction-address field of the PSW, and then a late specification exception is recognized due to an odd instruction address.

Condition Code: The code remains unchanged.

Program Exceptions:

- Trace (R_2 field nonzero)
- Transaction constraint

Programming Notes:

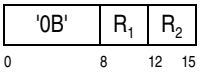
1. An example of the use of the BRANCH AND SAVE AND SET MODE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. BRANCH AND SAVE AND SET MODE is intended to be the principal calling instruction to subroutines which may operate in a different addressing mode from that of the caller. See the programming notes on pages 5-15 and 5-18 in the section “Subroutine Linkage without the Link-

age Stack" for a detailed discussion of this and other linkage instructions.

3. An old 24-bit or 31-bit program can use BRANCH AND SAVE AND SET MODE to call a new 64-bit program without any change, provided that bits 0-31 of general register R_2 are all zeros. The old program can load into bit positions 32-63 of general register R_2 a four-byte address constant, which is provided from outside the program, in which bit 63 in the register (bit 31 of the constant in storage) either is or is not one. If the addressing mode is not changed to the 64-bit mode by the execution of the BRANCH AND SAVE AND SET MODE instruction, or even if it is, the called program can set the 64-bit mode by issuing a SET ADDRESSING MODE (SAM64) instruction.
4. See the programming notes on page 5-15 (under "Simple Branch Instructions").

BRANCH AND SET MODE

BSM R_1,R_2 [RR]



In the 24-bit or 31-bit addressing mode, bit 32 of the current PSW, the basic-addressing-mode bit, is inserted into bit position 32 of the first operand, and bits 0-31 and 33-63 of the operand remain unchanged. In the 64-bit addressing mode, a one is inserted into bit position 63 of the first operand, and bits 0-62 of the operand remain unchanged. Subsequently, the addressing-mode bits and instruction address in the PSW are replaced as specified by the second operand. The action associated with an operand is not performed if the associated R field is zero.

The contents of general register R_2 specify the new addressing mode and designate the branch address; however, when the R_2 field is zero, the operation is performed without branching and without setting either addressing-mode bit.

When the contents of general register R_2 are used and bit 63 of the register is zero, bit 31 of the current PSW, the extended-addressing-mode bit, is set to zero, bit 32 of the register specifies the new basic addressing mode and replaces bit 32 of the PSW, and the branch address is generated from the contents of the register under the control of the new addressing mode. The branch address replaces the instruction address in the PSW.

In the z/Architecture architectural mode, when the contents of general register R_2 are used and bit 63 of the register is one, the following occurs. Bits 31 and 32 of the current PSW are set to one, the branch address is generated from the contents of the register, except with bit 63 of the register treated as a zero, under the control of the new extended addressing mode, and the branch address replaces the instruction address in the PSW. Bit 63 of the register remains one. However, if R_2 is the same as R_1, the results in the designated general register are as specified for the R_1 register.

The new value for the PSW is computed before general register R_1 is changed.

In the ESA/390-compatibility mode, when the R_2 field is nonzero and bit 63 of general register R_2 is one, it is unpredictable which of the following occurs:

- The instruction completes by setting PSW bit 31 to one, and either (a) an early specification exception is recognized immediately after the PSW becomes active due to bit 31 of the PSW being one, or (b) the CPU enters the 64-bit addressing mode as described above for the z/Architecture architectural mode.
- The instruction completes by placing the branch address, including bit 63, into the instruction-address field of the PSW, and then a late specification exception is recognized due to an odd instruction address.

Condition Code: The code remains unchanged.

Program Exceptions:

- Trace
- Transaction constraint

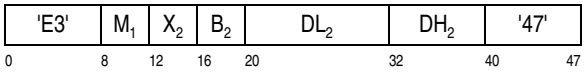
Programming Notes:

1. An example of the use of the BRANCH AND SET MODE instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. BRANCH AND SET MODE with an R_1 field of zero is intended to be the standard return

instruction in a program entered by means of BRANCH AND SAVE AND SET MODE. It can also be the return instruction in a program entered in the 24-bit or 31-bit addressing mode by means of BRANCH AND SAVE, BRANCH RELATIVE AND SAVE, or BRANCH RELATIVE AND SAVE LONG. BRANCH AND SET MODE with a nonzero R_1 field is intended to be used in a "glue module" to connect either old 24-bit programs and newer programs that are executed in the 31-bit addressing mode or old 24-bit or 31-bit programs and new programs that are executed in the 64-bit addressing mode. See the programming notes on pages 5-15 and 5-18 in the section "Subroutine Linkage without the Linkage Stack" for a detailed discussion of this and other linkage instructions.

BRANCH INDIRECT ON CONDITION

BIC $M_1,D_2(X_2,B_2)$ [RXY-b]



The instruction-address in the current PSW is replaced by the branch address if the condition code in the current PSW designates a bit position in the M_1 field containing a one; otherwise, normal instruction sequencing proceeds with the updated instruction address.

The eight-byte second operand in storage is used as the branch address. The branch address is subject to the current addressing mode. All eight bytes of the second operand are accessed, regardless of the addressing mode.

The M_1 field is used as a four-bit mask. The four condition codes (0, 1, 2, and 3) correspond, left to right, with the four bits of the mask, as follows:

Condition Code	0	1	2	3
Instruction Bit Number of Mask	8	9	10	11
Mask Position Value	8	4	2	1

The current condition code is used to select the corresponding mask bit. If the mask bit selected by the condition code is one, the branch is successful. If the mask bit selected is zero, normal instruction sequencing proceeds with the next sequential instruction.

The displacement is treated as a 20-bit signed binary integer.

It is model dependent whether an access exception or PER zero-address-detection event is recognized for the second operand when the condition code in the current PSW designates a bit position in the M_1 field containing a zero.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2)
- Operation (if the miscellaneous-instruction-extensions facility 2 is not installed)
- Transaction constraint

Programming Notes:

1. Unlike other branch-type instructions such as BRANCH ON CONDITION, where PER zero-address detection is not performed on the branch address, a PER zero-address-detection event may be recognized for the second operand of BRANCH INDIRECT ON CONDITION.
2. BRANCH INDIRECT ON CONDITION may be useful in returning to a calling program based on a return address in memory (such as in a program stack or save area).
3. The second operand is fetched using the current DAT and address-space controls in the PSW. However, the branch address that is fetched from the second-operand location is treated as an instruction address and is treated as a real address in the real mode, as a primary virtual address in the primary-space mode, secondary-space mode, or access-register mode, and as a home virtual address in the home-space mode.

4. The High-Level Assembler provides extended-mnemonic suffixes for BRANCH INDIRECT ON CONDITION as follows:

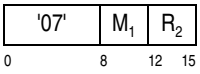
M_1 Field			Extended Mnemonics	
Decimal	Hex	Binary		
1	1	0001	BIO	—
2	2	0010	BIP	BIH
4	4	0100	BIM	BIL
7	7	0111	BINZ	BINE
8	8	1000	BIZ	BIE
11	B	1011	BINM	BINL
13	D	1101	BINP	BINH
14	E	1110	BINO	—
15	F	1111	BI	—

   When an extended mnemonic is coded, the M_1 field must be omitted.

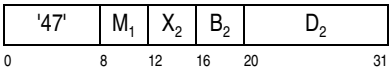
5. BRANCH INDIRECT ON CONDITION may be useful in a table-based branch mechanism.

BRANCH ON CONDITION

BCR M_1,R_2 [RR]



BC $M_1,D_2(X_2,B_2)$ [RX-b]



The instruction address in the current PSW is replaced by the branch address if the condition code has one of the values specified by M_1; otherwise, normal instruction sequencing proceeds with the updated instruction address.

In the RX format, the second-operand address is used as the branch address. In the RR format, the contents of general register R_2 are used to generate the branch address; however, when the R_2 field is zero, the operation is performed without branching.

The M_1 field is used as a four-bit mask. The four condition codes (0, 1, 2, and 3) correspond, left to right, with the four bits of the mask, as follows:

Condition Code	0	1	2	3
Instruction Bit Number of Mask	8	9	10	11
Mask Position Value	8	4	2	1

The current condition code is used to select the corresponding mask bit. If the mask bit selected by the condition code is one, the branch is successful. If the mask bit selected is zero, normal instruction sequencing proceeds with the next sequential instruction.

When the M_1 and R_2 fields of BRANCH ON CONDITION (BCR) are 1111 binary and 0000 binary, respectively, a serialization and checkpoint-synchronization function is performed. When the fast-BCR-serialization facility is installed and the M_1 and R_2 fields of BRANCH ON CONDITION (BCR) are 1110 binary and 0000 binary, respectively, a serialization function is performed.

Condition Code: The code remains unchanged.

Program Exceptions:

- Transaction constraint

Programming Notes:

1. An example of the use of the BRANCH ON CONDITION instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. When a branch is to depend on more than one condition, the pertinent condition codes are specified in the mask as the sum of their mask position values. A mask of 12, for example, specifies that a branch is to be made when the condition code is 0 or 1.

3. When all four mask bits are zeros or when the R_2 field in the RR format contains zero, the branch instruction is equivalent to a no-operation. When all four mask bits are ones, that is, the mask value is 15, the branch is unconditional unless the R_2 field in the RR format is zero.

4. Execution of BCR 15,0 (that is, an instruction with a value of 07F0 hex) may result in significant performance degradation. To ensure optimum

performance, the program should avoid use of BCR 15,0 except in cases when the serialization or checkpoint-synchronization function is actually required.

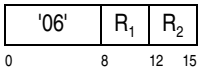
When the fast-BCR-serialization facility is installed, and the program only needs the serialization function (without the checkpoint synchronization function), then BCR 14,0 should be used. Depending on the model, this may be faster than BCR 15,0.

5. Note that the relation between the RR and RX formats in branch-address specification is not the same as in operand-address specification. For branch instructions in the RX format, the branch address is the address specified by X_2, B_2, and D_2; in the RR format, the branch address is contained in the register designated by R_2. For operands, the address specified by X_2, B_2, and D_2 is the operand address, but the register designated by R_2 contains the operand, not the operand address.

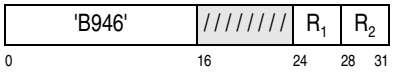
BRANCH ON COUNT

Register-and-register formats:

BCTR R_1,R_2 [RR]

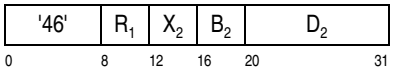


BCTGR R_1,R_2 [RRE]

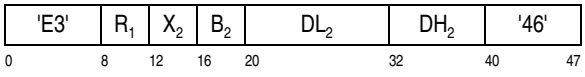


Register-and-storage formats:

BCT $R_1,D_2(X_2,B_2)$ [RX-a]



BCTG $R_1,D_2(X_2,B_2)$ [RXY-a]



A one is subtracted from the first operand, and the result is placed at the first-operand location. For BRANCH ON COUNT (BCT, BCTR), the first operand and result are treated as 32-bit binary integers, with overflow ignored. For BRANCH ON COUNT (BCTG, BCTGR), the first operand and result are treated as 64-bit binary integers, with overflow ignored. When the result is zero, normal instruction sequencing proceeds with the updated instruction address. When the result is not zero, the instruction address in the current PSW is replaced by the branch address.

In the RX or RXY format, the second-operand address is used as the branch address. In the RR or RRE format, the contents of general register R_2 are used to generate the branch address; however, when the R_2 field is zero, the operation is performed without branching. The branch address is generated before general register R_1 is changed.

Condition Code: The code remains unchanged.

Program Exceptions:

- Transaction constraint

Programming Notes:

1. An example of the use of the BRANCH ON COUNT instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

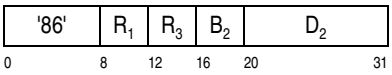
2. The first operand and result can be considered as either signed or unsigned binary integers since the result of a binary subtraction is the same in both cases.

3. An initial count of one results in zero, and no branching takes place; an initial count of zero results in -1 and causes branching to be performed; an initial count of -1 results in -2 and causes branching to be performed; and so on. In a loop, branching takes place each time the instruction is executed until the result is again zero. Note that for BCT or BCTR, because of the number range, an initial count of -2^{31} results in a positive value of $2^{31} - 1$, or, for BCTG or BCTGR, an initial count of -2^{63} results in a positive value of $2^{63} - 1$.

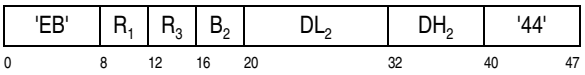
4. Counting is performed without branching when the R_2 field in the RR or RRE format contains zero.

BRANCH ON INDEX HIGH

BXH $R_1,R_3,D_2(B_2)$ [RS-a]

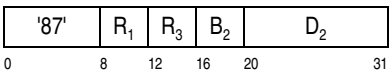


BXHG $R_1,R_3,D_2(B_2)$ [RSY-a]

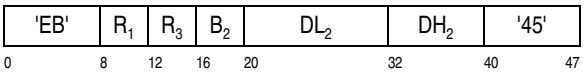


BRANCH ON INDEX LOW OR EQUAL

BXLE $R_1,R_3,D_2(B_2)$ [RS-a]



BXLEG $R_1,R_3,D_2(B_2)$ [RSY-a]



An increment is added to the first operand, and the sum is compared with a compare value. The result of the comparison determines whether branching occurs. Subsequently, the sum is placed at the first-operand location. The second-operand address is used as a branch address. The R_3 field designates registers containing the increment and the compare value.

For BRANCH ON INDEX HIGH, when the sum is high, the instruction address in the current PSW is replaced by the branch address. When the sum is low or equal, normal instruction sequencing proceeds with the updated instruction address.

For BRANCH ON INDEX LOW OR EQUAL, when the sum is low or equal, the instruction address in the current PSW is replaced by the branch address. When the sum is high, normal instruction sequencing proceeds with the updated instruction address.

When the R_3 field is even, it designates a pair of registers; the contents of the even and odd registers of the pair are used as the increment and the compare value, respectively. When the R_3 field is odd, it designates a single register, the contents of which are used as both the increment and the compare value.

For purposes of the addition and comparison, all operands and results are treated as 32-bit signed binary integers for BXH and BXLE or as 64-bit signed binary integers for BXHG and BXLEG. Overflow caused by the addition is ignored.

The original contents of the compare-value register are used as the compare value even when that register is also specified to be the first-operand location. The branch address is generated before general register R_1 is changed.

The sum is placed at the first-operand location, regardless of whether the branch is taken.

Condition Code: The code remains unchanged.

Program Exceptions:

- Transaction constraint

Programming Notes:

1. Several examples of the use of the BRANCH ON INDEX HIGH and BRANCH ON INDEX LOW OR EQUAL instructions are given in Appendix A, “Number Representation and Instruction-Use Examples.”

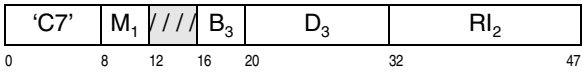
2. The word “index” in the names of these instructions indicates that one of the major purposes is the incrementing and testing of an index value. The increment, being a signed binary integer, may be used to increase or decrease the value in general register R_1 by an arbitrary amount, subject to the limit of the integer size.

3. Care must be taken in the 31-bit addressing mode when a data area in storage is at the rightmost end of a 31-bit address space and a BRANCH ON INDEX HIGH (BXH) or BRANCH ON INDEX LOW OR EQUAL (BXLE) instruction is used to step upward through the data. Since the addition and comparison operations performed during the execution of these instructions treat the operands as 32-bit signed binary integers, the value following $2^{31} - 1$ is not 2^{31}, which cannot be represented in that format, but -2^{31}. The instruction does not provide an indication of such overflow. Consequently, some common looping techniques based on the use of these instructions do not work when a data area ends at address $2^{31} - 1$. This problem is illustrated in a BRANCH ON INDEX LOW OR EQUAL example

in Appendix A, “Number Representation and Instruction-Use Examples.” A similar caution applies in the 64-bit addressing mode when data is at the end of a 64-bit address space and BRANCH ON INDEX HIGH (BXHG) or BRANCH ON INDEX LOW OR EQUAL (BXLEG) is used.

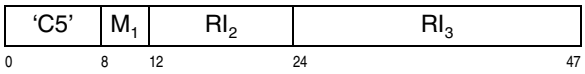
BRANCH PREDICTION PRELOAD

BPP $M_1,RI_2,D_3(B_3)$ [SMI]



BRANCH PREDICTION RELATIVE PRELOAD

BPRP M_1,RI_2,RI_3 [MII]



Subject to the controls in the M_1 field, the CPU is provided with information about a branch or execute-type instruction designated by the second operand. The predicted target address of the designated instruction is specified by the third operand.

The contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the current instruction to generate the address of a branch, or execute-type instruction. For BRANCH PREDICTION PRELOAD, the RI_2 field contains a signed 16-bit integer. For BRANCH PREDICTION RELATIVE PRELOAD, the RI_2 field contains a signed 12-bit integer.

When adding the number of halfwords specified in the RI_2 field to the address of the current instruction, the result is subject to the current addressing mode. That is, the result is treated as a 24-bit address in the 24-bit addressing mode, a 31-bit address in the 31-bit addressing mode, or a 64-bit address in the 64-bit addressing mode.

The M_1 field contains a 4-bit unsigned binary integer that is used as a code to signal the CPU attributes of the instruction designated by the second operand. The codes are as follows:

Code	Meaning
0	The instruction is a branch instruction that is 4 bytes in length. The branch instruction is not used for calling linkage or a returning linkage and there may be multiple potential targets of this branch instruction.
1-4	Reserved.
5	The instruction is a branch instruction that is 2 bytes in length. The branch instruction is used for calling linkage and there is only one target of this branch instruction.
6	The instruction is a branch instruction that is 2 bytes in length. The branch instruction is used for returning linkage.
7	The instruction is a branch instruction that is 2 bytes in length. The branch instruction is used for calling linkage and there may be multiple potential targets of this branch instruction.
8	The instruction is a branch instruction that is 4 bytes in length. The branch instruction is not used for calling linkage or returning linkage and there is only one target of this branch instruction.
9	The instruction is a branch instruction that is 4 bytes in length. The branch instruction is used for calling linkage and there is only one target of the branch instruction.
10	The instruction is a branch instruction that is 4 bytes in length. The branch instruction is used for returning linkage.
11	The instruction is a branch instruction that is 4 bytes in length. The branch instruction is used for calling linkage and there may be multiple potential targets of this branch instruction.
12	The instruction is a branch instruction that is 6 bytes in length. The branch instruction is not used for calling linkage or returning linkage and there is only one target of this branch instruction.
13	The instruction is a branch instruction that is 6 bytes in length. The branch instruction is used for calling linkage and there is only one target of this branch instruction.
14	The instruction is an EXECUTE instruction.
15	The instruction is an EXECUTE RELATIVE LONG instruction.

For BRANCH PREDICTION PRELOAD when the M_1 field specifies a branch instruction, the third operand address is the predicted branch-target address of the

instruction designated by the second operand. For BRANCH PREDICTION PRELOAD when the M_1 field specifies an execute-type instruction, the third operand address is the execute-target address of the instruction designated by the second operand.

For BRANCH PREDICTION RELATIVE PRELOAD, when the M_1 field specifies a branch instruction, the contents of the RI_3 field are a 24-bit signed binary integer specifying the number of halfwords that is added to the address of the BRANCH PREDICTION RELATIVE PRELOAD instruction to generate the branch-target address of the instruction designated by the second operand. For BRANCH PREDICTION RELATIVE PRELOAD, when the M_1 field specifies an execute-type instruction, the contents of the RI_3 field are a 24-bit signed binary integer specifying the number of halfwords that is added to the address of the BRANCH PREDICTION RELATIVE PRELOAD instruction to generate the execute-target address of the instruction designated by the second operand. When adding the number of halfwords specified by the RI_3 field to the address of the BRANCH PREDICTION RELATIVE PRELOAD instruction, the result is subject to the current addressing mode. That is, the result is treated as a 24-bit address in the 24-bit addressing mode, a 31-bit address in the 31-bit addressing mode, or a 64-bit address in the 64-bit addressing mode.

When the boundary of the third operand of BRANCH PREDICTION PRELOAD is not on a halfword boundary, it is model dependent whether the instruction acts as a no-operation.

Depending on the model, the CPU may not implement all of the branch-attribute codes. For codes that are not recognized by the CPU, and for reserved codes, the instruction acts as a no-operation.

The second and third operand addresses are instruction or effective addresses, not logical addresses. No access exceptions are recognized for the second or third operands. For BRANCH PREDICTION PRELOAD, there is no specification exception when the third operand specifies an odd address.

Condition Code: The code remains unchanged.

Program Exceptions:

- Operation (if the execution-hint facility is not installed)

Programming Notes:

1. In the absence of any branch-history information observed during program execution or provided by BPP or BPRP, current processor models that implement branch-prediction logic assume that the following operations represent conditional branches that are taken (that is, branches that update the PSW instruction address):

   - BAL, BAS, BCT, BCTG, BRAS, BRASL, BRCT, BRCTG, BRCTH, BRXLE, BRXLG, BXLE, and BXLEG
   - BALR, BASR, BCTGR, and BCTR, when the R_2 field is nonzero
   - BC, BIC, BRC, and BRCL, when the mask field is 1111 binary
   - BCR, when both the mask field is 1111 binary and the R_2 field is nonzero

   Similarly, for the purposes of branch prediction, the following operations are assumed to represent conditional branches that are not taken:

   - BC, BIC, BRC, and BRCL, when the mask field is between 0001 and 1110 binary, inclusive
   - BCR, when both the mask field is between 0001 and 1110 binary, inclusive, and the R_2 field is nonzero
   - BRXH, BRXHG, BXH, and BXHG
   - CGIB, CGIJ, CGRB, CGRJ, CIB, CIJ, CLGIB, CLGIJ, CLGRB, CLGRJ, CLIB, CLIJ, CLRB, CLRJ, CRB, and CRJ, when bits 0-2 of the mask field are between 001 and 110 binary

   Additionally, for the purposes of branch-prediction, the following operations are not considered to be branches.

   - BALR, BASR, BCTGR, and BCTR, when the R_2 field is zero
   - BC, BIC, BRC, and BRCL, when the mask field is 0000 binary
   - BCR, when either the mask field is 0000 binary or the R_2 field is zero

   The predictive behavior of conditional-branching operations not listed above is model dependent.

   The BPP and BPRP instructions provide a means by which the program can influence the branch-history logic, informing the CPU that the branching instruction designated by the second operand is expected to be a taken branch,

regardless of the assumed behavior described above.

2. BRANCH PREDICTION PRELOAD and BRANCH PREDICTION RELATIVE PRELOAD inform the CPU of a branch instruction and its predicted target, but this does not guarantee that the CPU will necessarily retain or use this branch-prediction information.

3. If a CPU retains the specified branch-prediction information, it is model dependent how long the information is retained.

4. Figure 7-3 shows the branch-prediction codes and their most-common usage. Refer to the definition of the codes on page 7-43 for a complete explanation of the codes and their respective meanings.

M_1 Code Value	IL	Corresponding Branch Instruction(s) by Mnemonic	Usage Type
0	4	BC[2], BIC[2]	Branch table
1		(none - code is reserved)	
2		(none - code is reserved)	
3		(none - code is reserved)	
4		(none - code is reserved)	
5	2	BALR[1], BASR[1], BCR[1,2]	Static calling linkage
6	2	BCR[1,2]	Returning linkage
7	2	BALR[1], BASR[1], BCR[1,2]	Dynamic calling linkage
8	4	BC[2], BIC[2], BCT, BRXH, BRXLE, BXH, BXLE, BRC[2], BRCT, BRCTG, BCTGR[1]	Conditional or unconditional branches
9	4	BAL, BAS, BRAS	Static calling linkage
10	4	BC[3] , BIC[2]	unconditional returning linkage
11	4	BAL, BAS,	Dynamic calling linkage
12	6	BRCTH, BRCL[2], BCTG, BXHG, BXLEG, BRXHG, BRXLG, CGRJ[2], CLGRJ[2], CRJ[2], CLRJ[2], CGIJ[2], CLGIJ[2], CIJ[2], CLIJ[2], CGRB[2], CLGRB[2], CRB[2], CLRB[2], CGIB[2], CLGIB[2], CIB[2], CLIB[2]	Conditional or unconditional branches
13	6	BRASL	Static calling linkage
14	4	EX	
15	6	EXRL	

Figure 7-3. Branch-Prediction Codes and Most-Common Usage (Part 1 of 2)

M_1 Code Value	IL	Corresponding Branch Instruction(s) by Mnemonic	Usage Type

Explanation:

IL Instruction length
1 When the R_2 field is not zero.
2 When the mask field is not zero.
3 When the mask field is 1111 binary.

Notes:

1. A usage type of *branch table* indicates an instruction that is used to branch to a table of other instructions. An example is the "B BR_TBL(15)" instruction shown in programming note 3 for "TRANSLATE AND TEST EXTENDED" on page 7-403.
2. A usage type of *linkage* designates an instruction that calls or returns from a subroutine, as described in "Subroutine Linkage without the Linkage Stack" on page 5-14. Static calling linkage indicates a branch whose target location is a single location that is not changed by the program during execution. Dynamic calling linkage indicates a branch whose target location may be one of many locations that is determined by the program during execution.

Figure 7-3. Branch-Prediction Codes and Most-Common Usage (Part 2 of 2)

5. Performance degradation may occur for any of the following:

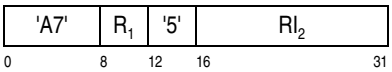
   - The RI_2 field is zero or the second operand designates an instruction that is not a branch instruction or execute-type instruction.
   - The RI_2 and M_1 fields designate a branch instruction and the branch is not taken.
   - The M_1 field designates a code specifying an incorrect length, instruction or usage type for the instruction designated by the second operand as shown by programming note 4.
   - The M_1 field designates a code specifying that the instruction designated by the second operand is a calling-linkage instruction, however there is no direct returning-linkage instruction. That is, the returning-linkage instruction is executed from a nested use of calling and returning subroutine linkage.
   - For BRANCH PREDICTION PRELOAD, the third operand specifies an odd address or designates an incorrect branch-target for the instruction designated by the second operand. For BRANCH PREDICTION RELATIVE PRELOAD, the RI_3 field designates an incorrect branch-target for the instruction designated by the RI_2 field.

- BRANCH PREDICTION PRELOAD or BRANCH PREDICTION RELATIVE PRELOAD are the target of an execute-type instruction.

6. Other than the performance implications described in the preceding note, the use of BPP or BPRP in a program does not change the execution sequence from that of a program that does not use BPP or BPRP.

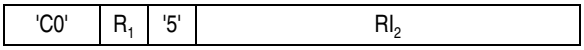
BRANCH RELATIVE AND SAVE

BRAS R_1,RI_2 [RI-b]



BRANCH RELATIVE AND SAVE LONG

BRASL R_1,RI_2 [RIL-b]



Information from the current PSW, including the updated instruction address, is saved as link information at the first-operand location. Subsequently, the instruction address in the PSW is replaced by the branch address.

In the 24-bit or 31-bit addressing mode, the link information is bits 32 and 97-127 of the PSW, consisting of the basic-addressing-mode bit and the rightmost 31 bits of the updated instruction address. The link information is placed in bit positions 32 and 33-63, respectively, of the first-operand location, and bits 0-31 of the location remain unchanged.

In the 64-bit addressing mode, the link information consists of the updated instruction address, placed in bit positions 0-63 of the first-operand location.

The contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the branch address.

Condition Code: The code remains unchanged.

Program Exceptions:

- Transaction constraint

Programming Notes:

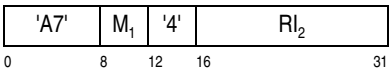
1. The operation is the same as that of the BRANCH AND SAVE (BAS) instruction except for the means of specifying the branch address. An example of the use of BRANCH AND SAVE is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. The BRANCH RELATIVE AND SAVE and BRANCH RELATIVE AND SAVE LONG instructions, like the BRANCH AND SAVE instruction, are intended to be used for linkage to programs known to be in the same addressing mode as the caller. These instructions should be used in place of the BRANCH AND LINK instruction (BAL and BALR). See the programming notes on pages 5-15 and 5-18 in the section “Subroutine Linkage without the Linkage Stack” for a detailed discussion of these and other linkage instructions. See also the programming note under BRANCH AND LINK for a discussion of the advantages of the BRANCH RELATIVE AND SAVE, BRANCH RELATIVE AND SAVE LONG, and BRANCH AND SAVE instructions.

3. When the instruction is the target of an execute-type instruction, the branch is relative to the target address; see “Branch-Address Generation” on page 5-12.

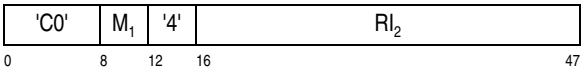
BRANCH RELATIVE ON CONDITION

BRC M_1,RI_2 [RI-c]



BRANCH RELATIVE ON CONDITION LONG

BRCL M_1,RI_2 [RIL-c]



The instruction address in the current PSW is replaced by the branch address if the condition code has one of the values specified by M_1; otherwise, nor-

mal instruction sequencing proceeds with the updated instruction address.

The contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the branch address.

The M_1 field is used as a four-bit mask. The four condition codes (0, 1, 2, and 3) correspond, left to right, with the four bits of the mask, as follows:

Condition Code	0	1	2	3
Instruction Bit Number of Mask	8	9	10	11
Mask Position Value	8	4	2	1

The current condition code is used to select the corresponding mask bit. If the mask bit selected by the condition code is one, the branch is successful. If the mask bit selected is zero, normal instruction sequencing proceeds with the next sequential instruction.

Condition Code: The code remains unchanged.

Program Exceptions:

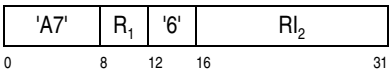
- Transaction constraint

Programming Notes:

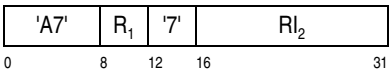
1. The operation is the same as that of the BRANCH ON CONDITION instruction except for the means of specifying the branch address. An example of the use of BRANCH ON CONDITION is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. When a branch is to depend on more than one condition, the pertinent condition codes are specified in the mask as the sum of their mask position values. A mask of 12, for example, specifies that a branch is to be made when the condition code is 0 or 1.
3. When all four mask bits are zeros, the branch instruction is equivalent to a no-operation. When all four mask bits are ones, that is, the mask value is 15, the branch is unconditional.
4. When the instruction is the target of an execute-type instruction, the branch is relative to the target address; see “Branch-Address Generation” on page 5-12.

BRANCH RELATIVE ON COUNT

BRCT R_1,RI_2 [RI-b]

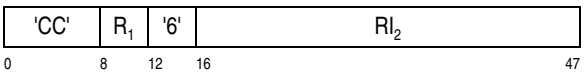


BRCTG R_1,RI_2 [RI-b]



BRANCH RELATIVE ON COUNT HIGH

BRCTH R_1,RI_2 [RIL-b]



A one is subtracted from the first operand, and the result is placed at the first-operand location. For BRANCH RELATIVE ON COUNT (BRCT), the first operand and result are treated as 32-bit binary integers in bits 32-63 of general register R_1, with overflow ignored; bits 0-31 of the register are unchanged. For BRANCH RELATIVE ON COUNT HIGH (BRCTH), the first operand and result are treated as 32-bit binary integers in bits 0-31 of general register R_1, with overflow ignored; bits 32-63 of the register are unchanged. For BRANCH RELATIVE ON COUNT (BRCTG), the first operand and result are treated as 64-bit binary integers, with overflow ignored. When the result is zero, normal instruction sequencing proceeds with the updated instruction address. When the result is not zero, the instruction address in the current PSW is replaced by the branch address.

The contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the branch address.

Condition Code: The code remains unchanged.

Program Exceptions:

- Operation (BRCTH, if the high-word facility is not installed)
- Transaction constraint

Programming Notes:

1. The operation is the same as that of the BRANCH ON COUNT instruction except for the means of specifying the branch address. An example of the use of BRANCH ON COUNT is given in Appendix A, “Number Representation and Instruction-Use Examples.”

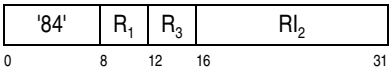
2. The first operand and result can be considered as either signed or unsigned binary integers since the result of a binary subtraction is the same in both cases.

3. An initial count of one results in zero, and no branching takes place; an initial count of zero results in -1 and causes branching to be executed; an initial count of -1 results in -2 and causes branching to be executed; and so on. In a loop, branching takes place each time the instruction is executed until the result is again zero. Note that for BRCT, because of the number range, an initial count of -2^{31} results in a positive value of $2^{31} - 1$, or, for BRCTG, an initial count of -2^{63} results in a positive value of $2^{63} - 1$.

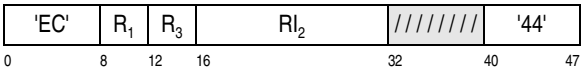
4. When the instruction is the target of an execute-type instruction, the branch is relative to the target address; see “Branch-Address Generation” on page 5-12.

BRANCH RELATIVE ON INDEX HIGH

BRXH R_1,R_3,RI_2 [RSI]

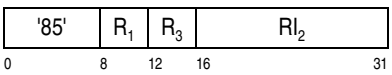


BRXHG R_1,R_3,RI_2 [RIE-e]

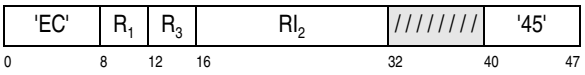


BRANCH RELATIVE ON INDEX LOW OR EQUAL

BRXLE R_1,R_3,RI_2 [RSI]



BRXLG R_1,R_3,RI_2 [RIE-e]



An increment is added to the first operand, and the sum is compared with a compare value. The result of the comparison determines whether branching occurs. Subsequently, the sum is placed at the first-operand location. The R_3 field designates registers containing the increment and the compare value.

The contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the branch address.

For BRANCH RELATIVE ON INDEX HIGH, when the sum is high, the instruction address in the current PSW is replaced by the branch address. When the sum is low or equal, normal instruction sequencing proceeds with the updated instruction address.

For BRANCH RELATIVE ON INDEX LOW OR EQUAL, when the sum is low or equal, the instruction address in the current PSW is replaced by the branch address. When the sum is high, normal instruction sequencing proceeds with the updated instruction address.

When the R_3 field is even, it designates a pair of registers; the contents of the even and odd registers of the pair are used as the increment and the compare value, respectively. When the R_3 field is odd, it designates a single register, the contents of which are used as both the increment and the compare value.

For purposes of the addition and comparison, all operands and results are treated as 32-bit signed binary integers for BRXH and BRXLE or as 64-bit signed binary integers for BRXHG and BRXLG. Overflow caused by the addition is ignored.

The original contents of the compare-value register are used as the compare value even when that register is also specified to be the first-operand location.

The sum is placed at the first-operand location, regardless of whether the branch is taken.

Condition Code: The code remains unchanged.

Program Exceptions:

- Transaction constraint

Programming Notes:

1. The operations are the same as those of the BRANCH ON INDEX HIGH and BRANCH ON INDEX LOW OR EQUAL instructions except for the means of specifying the branch address. Several examples of the use of BRANCH ON INDEX HIGH and BRANCH ON INDEX LOW OR EQUAL are given in Appendix A, “Number Representation and Instruction-Use Examples.”

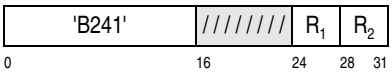
2. The word “index” in the names of these instructions indicates that one of the major purposes is the incrementing and testing of an index value. The increment, being a signed binary integer, may be used to increase or decrease the value in general register R_1 by an arbitrary amount.

3. Care must be taken in the 31-bit addressing mode when a data area in storage is at the rightmost end of an address space and a BRANCH RELATIVE ON INDEX HIGH (BRXH) or BRANCH RELATIVE ON INDEX LOW OR EQUAL (BRXLE) instruction is used to step upward through the data. Since the addition and comparison operations performed during the execution of these instructions treat the operands as 32-bit signed binary integers, the value following $2^{31} - 1$ is not 2^{31}, which cannot be represented in that format, but -2^{31}. The instruction does not provide an indication of such overflow. Consequently, some common looping techniques based on the use of these instructions do not work when a data area ends at address $2^{31} - 1$. This problem is illustrated in a BRANCH ON INDEX LOW OR EQUAL example in Appendix A, “Number Representation and Instruction-Use Examples.” A similar caution applies in the 64-bit addressing mode when data is at the end of a 64-bit address space and BRANCH RELATIVE ON INDEX HIGH (BRXHG) or BRANCH RELATIVE ON INDEX LOW OR EQUAL (BRXLG) is used.

4. When the instruction is the target of an execute-type instruction, the branch is relative to the target address; see “Branch-Address Generation” on page 5-12.

CHECKSUM

CKSM R_1,R_2 [RRE]



Successive four-byte elements of the second operand are added to the first operand in bit positions 32-63 of general register R_1 to form a 32-bit checksum in those bit positions. The first operand and the four-byte elements are treated as 32-bit unsigned binary integers. After each addition of an element, a carry out of bit position 32 of the first operand is added to bit position 63 of the first operand. Bits 0-31 of general register R_1 always remain unchanged. If the second operand is not a multiple of four bytes, its last one, two, or three bytes are treated as appended on the right with the number of all-zeros bytes needed to form a four-byte element. The four-byte elements are added to the first operand until either the entire second operand or a CPU-determined amount of the second operand has been processed. The result is indicated in the condition code.

The R_2 field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the second operand is specified by the contents of the R_2 general register. The number of bytes in the second-operand location is specified by the 32-bit or 64-bit unsigned binary integer in the $R_2 + 1$ general register.

The handling of the address in general register R_2 and the length in general register $R_2 + 1$ is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the register constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of the register constitute the address. In the 24-bit or 31-bit addressing mode, the length is a 32-bit unsigned binary integer in bit positions 32-63 of general register $R_2 + 1$, and the contents of bit positions 0-31 are ignored. In the 64-bit addressing mode, the length is a 64-bit unsigned binary integer in the register.

The addition of second-operand four-byte elements to the first operand proceeds left to right, four-byte element by four-byte element, and ends as soon as (1) the entire second operand has been processed or (2) a lesser CPU-determined amount of the second operand has been processed. In either case, the result in bit positions 32-63 of general register R_1 is a 32-bit checksum for the part of the second operand that has been processed. When the second operand is not a multiple of four bytes, the final second-operand bytes in excess of a multiple of four are conceptually appended on the right with an appropriate number of all-zeros bytes to form the final four-byte element.

If the operation ends because the entire second operand has been processed, the condition code is set to 0. If the operation ends because a lesser CPU-determined amount of the second operand has been processed, the condition code is set to 3. When the operation is to end with a setting of condition code 3, any carry out of bit position 32 of the first operand is added to bit position 63 of the first operand before the operation ends.

At the completion of the operation, the 32-bit or 64-bit operand-length field in the $R_2 + 1$ register is decremented by the number of actual second-operand bytes added to the first operand (not including any conceptually appended all-zeros bytes), and the address in the R_2 register is incremented by the same number. Thus, the 32-bit or 64-bit operand-length field contains a zero value if the condition code is set to 0, or it contains a nonzero value if the condition code is set to 3. In the 24-bit or 31-bit addressing mode, bits 0-31 of the $R_2 + 1$ register always remain unchanged.

When condition code 3 is set, the general registers used by the instruction have been set so that the remainder of the second operand can be processed by simply branching back to reexecute the instruction.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed. The minimum amount is four bytes or the number of bytes specified in the $R_2 + 1$ general register, whichever is smaller.

At the completion of the operation in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general register R_2 may be set to zeros or may remain unchanged, even when the initial length in register $R_2 + 1$ is zero. Bits 0-31 of general register R_2 remain unchanged.

When the R_1 register is the same register as the R_2 or $R_2 + 1$ register, the results are unpredictable.

Access exceptions for the portion of the second operand to the right of the last byte processed may or may not be recognized. For a second operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

Access exceptions are not recognized if the R_2 field is odd. When the length of the second operand is zero, no access exceptions are recognized.

Resulting Condition Code:

0 Entire second operand processed
1 --
2 --
3 CPU-determined amount of the second operand processed

Program Exceptions:

- Access (fetch, operand 2)
- Specification
- Transaction constraint

Programming Notes:

1. The initial contents of bit positions 32-63 of the R_1 general register contribute to the 32-bit checksum. The program normally should set those contents to all zeros before issuing the CHECKSUM instruction.

2. A 16-bit checksum is used in, for example, the TCP/IP application. The following program can be executed after the CHECKSUM instruction to produce in bit positions 32-63 of general register R_2 a 16-bit checksum from the 32-bit checksum in bit positions 32-63 of general register R_1. The program is annotated to show the contents of bit positions 32-63 of the R_2 and $R_2 + 1$ registers after the execution of each instruction. The contents of bit positions 32-63 of the R_1 register are represented as A,B, meaning the value A in bit positions 32-47 and the value B in bit positions

48-63. The value C is a carry from A + B. Note that bit positions 32-63 of register $R_2 + 1$ are known to contain all zeros when CHECKSUM has set condition code 0.

Program		R_2 Bits 32-63	$R_2 + 1$ Bits 32-63
LR	R_2,R_1	A,B	0,0
SRDL	R_2,16	0,A	B,0
ALR	R_2,R_2+1	B,A	B,0
ALR	R_2,R_1	A+B+C,A+B	B,0
SRL	R_2,16	0,A+B+C	B,0

3. The CHECKSUM instruction may be used in computing hash values as illustrated in the following programming example. The variable KEY contains a string to be mapped into a slot in a hash table. The variable SIZE is a prime number designating the size of the hash table. The value of SIZE is determined by (a) the number of strings to be hashed into the table divided by the acceptable number of hash collisions, and (b) a value that is not too close to a power of two. Following the DIVIDE (D) instruction, the remainder in register 0 represents the resulting hash value.

```
        SR      1,1         Zero accumulator
        LA      2,KEY       Point to string
        LA      3,L'KEY     Load string length
LOOP    CKSM    1,2         Compute checksum
        BNZ     LOOP        Repeat if not done
        SR      0,0         Zero for divide
        D       0,SIZE      Compute hash value
        …
KEY     DS      CL64        String to be hashed
SIZE    DS      F           Size of hash table
```

4. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.
5. The storage-operand references of CHECKSUM may be multiple-access references. (See “Storage-Operand Consistency” on page 5-125.)
6. Figure 7-4 on page 7-52 contains a summary of the operation.

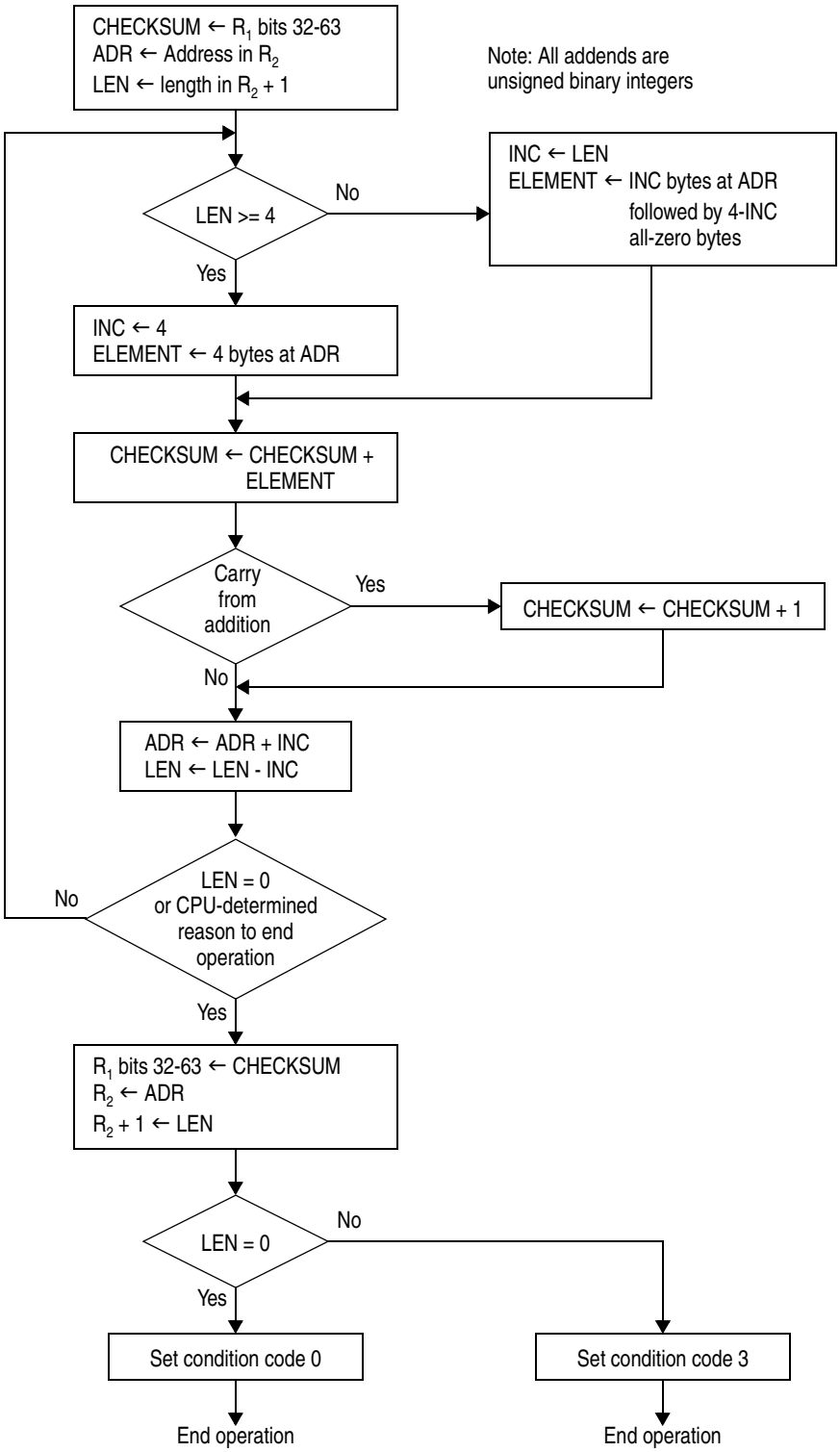
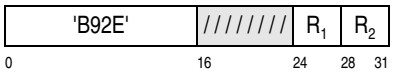


Figure 7-4. Execution of CHECKSUM

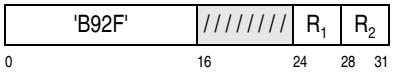
CIPHER MESSAGE

KM R_1,R_2 [RRE]



CIPHER MESSAGE WITH CHAINING

KMC R_1,R_2 [RRE]



A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction are ignored.

Bit positions 57-63 of general register 0 contain the function code. Figure 7-5 and 7-6 show the assigned function codes for CIPHER MESSAGE and CIPHER MESSAGE WITH CHAINING, respectively. All other function codes are unassigned. For cipher functions, bit 56 is the modifier bit which specifies whether an encryption or a decryption operation is to be performed. The modifier bit is ignored for all other functions. All other bits of general register 0 are ignored.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register 1 constitute the address.

The function codes for CIPHER MESSAGE are as follows.

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
0	KM-Query *	16	—
1	KM-DEA *	8	8
2	KM-TDEA-128 *	16	8
3	KM-TDEA-192 *	24	8
9	KM-Encrypted-DEA	32	8
10	KM-Encrypted-TDEA-128	40	8
11	KM-Encrypted-TDEA-192	48	8
18	KM-AES-128 *	16	16
19	KM-AES-192 *	24	16
20	KM-AES-256 *	32	16
26	KM-Encrypted-AES-128	48	16
27	KM-Encrypted-AES-192	56	16
28	KM-Encrypted-AES-256	64	16
50	KM-XTS-AES-128	32	16
52	KM-XTS-AES-256	48	16
58	KM-XTS-Encrypted-AES-128	64	16
60	KM-XTS-Encrypted-AES-256	80	16

Explanation:

— Not applicable

* Function is also defined in the ESA/390 architectural mode and the ESA/390-compatibility mode. It is unpredictable whether other function codes are available in the ESA/390-compatibility mode.

Figure 7-5. Function Codes for CIPHER MESSAGE

The function codes for CIPHER MESSAGE WITH CHAINING are as follows.

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
0	KMC-Query *	16	—
1	KMC-DEA *	16	8
2	KMC-TDEA-128 *	24	8
3	KMC-TDEA-192 *	32	8
9	KMC-Encrypted-DEA	40	8

Figure 7-6. Function Codes for CIPHER MESSAGE WITH CHAINING

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
10	KMC-Encrypted-TDEA-128	48	8
11	KMC-Encrypted-TDEA-192	56	8
18	KMC-AES-128 *	32	16
19	KMC-AES-192 *	40	16
20	KMC-AES-256 *	48	16
26	KMC-Encrypted-AES-128	64	16
27	KMC-Encrypted-AES-192	72	16
28	KMC-Encrypted-AES-256	80	16
67	KMC-PRNG	32	8

Explanation:

— Not applicable

* Function is also defined in the ESA/390 architectural mode and the ESA/390-compatibility mode. It is unpredictable whether other function codes are available in the ESA/390-compatibility mode.

Figure 7-6. Function Codes for CIPHER MESSAGE WITH CHAINING (Continued)

All other function codes are unassigned.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R_1, R_2, and $R_2 + 1$ are ignored for the query function.

For all other functions, the second operand is ciphered as specified by the function code using a cryptographic key in the parameter block, and the result is placed in the first-operand location. For CIPHER MESSAGE WITH CHAINING, ciphering also uses an initial chaining value in the parameter block, and the chaining value is updated as part of the operation. For XTS functions, ciphering also uses an XTS parameter in the parameter block, and the XTS parameter is updated as part of the operation.

The R_1 field designates a general register and must designate an even-numbered register other than general register 0; otherwise, a specification exception is recognized. The R_2 field designates an even-odd pair of general registers and must designate an even-numbered register other than general register 0; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first and second operands is specified by the contents of the R_1 and R_2 general registers, respectively. The number of bytes in the second-operand location is specified in general register $R_2 + 1$. The first operand is the same length as the second operand.

As part of the operation, the addresses in general registers R_1 and R_2 are incremented by the number of bytes processed, and the length in general register $R_2 + 1$ is decremented by the same number. The formation and updating of the addresses and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the addresses of the first and second operands, respectively, and the contents of bit positions 0-39 are ignored; bits 40-63 of the updated addresses replace the corresponding bits in general registers R_1 and R_2, carries out of bit position 40 of the updated address are ignored, and the contents of bit positions 32-39 of general registers R_1 and R_2 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general registers R_1 and R_2 constitute the addresses of the first and second operands, respectively, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated addresses replace the corresponding bits in general registers R_1 and R_2, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general registers R_1 and R_2 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general registers R_1 and R_2 constitute the addresses of the first and second operands, respectively; bits 0-63 of the updated addresses replace the contents of general registers R_1 and R_2, and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_2 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the first and second operands, and the contents of bit positions 0-31 are ignored; bits 32-63 of the updated value replace the corresponding bits in general register $R_2 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the first and second operands; and the updated value replaces the contents of general register $R_2 + 1$.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, R_2, and $R_2 + 1$, always remain unchanged.

Figure 7-7 on page 7-55 shows the contents of the general registers just described.

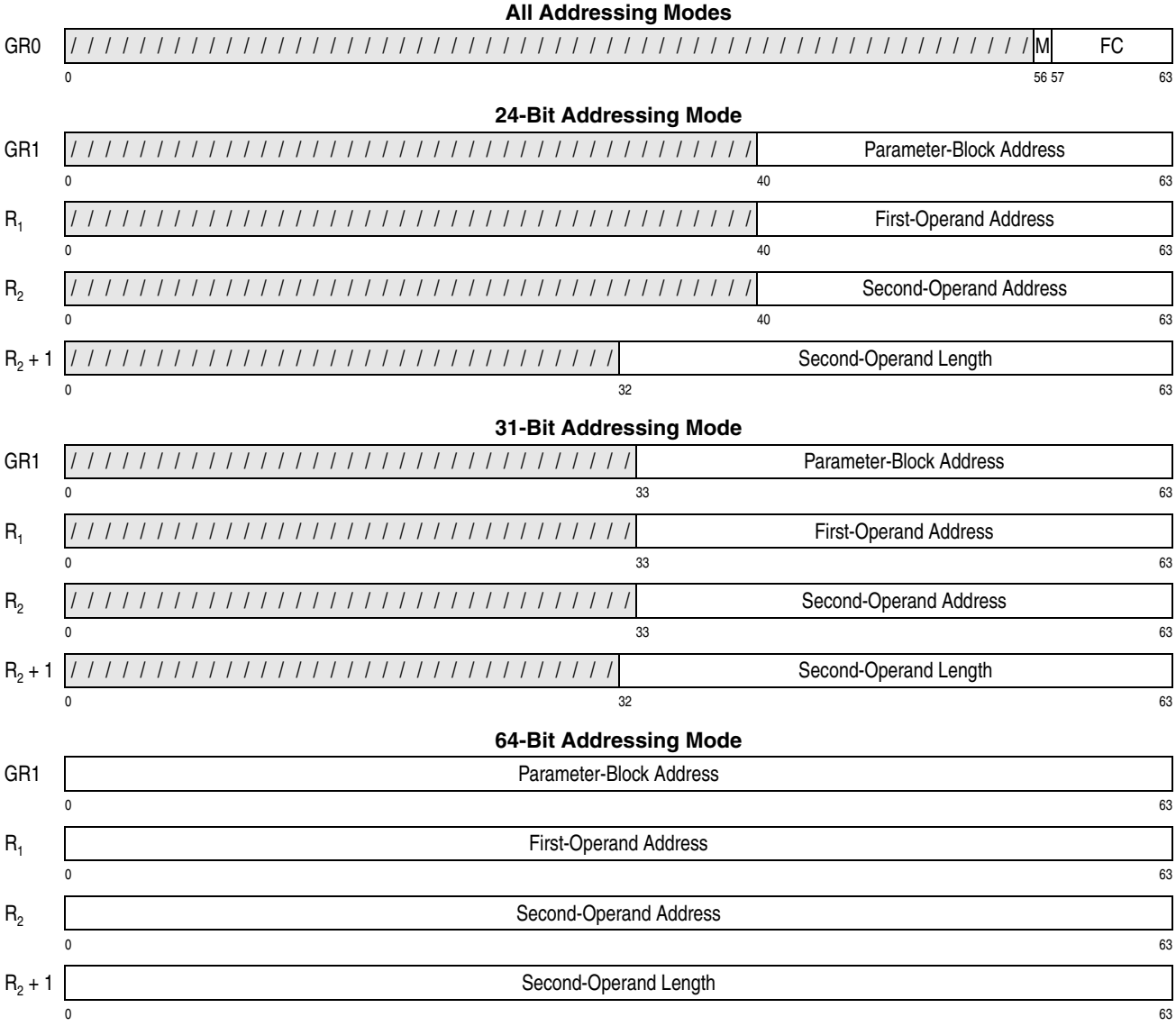


Figure 7-7. General Register Assignment for KM and KMC

In the access-register mode, access registers 1, R_1, and R_2 specify the address spaces containing the parameter block, first, and second operands, respectively.

The result is obtained as if processing starts at the left end of both the first and second operands and proceeds to the right, block by block. The operation is ended when the number of bytes in the second operand as specified in general register $R_2 + 1$ have been processed and placed at the first-operand location (called normal completion) or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

The results in the first-operand location, the chaining-value field, or the XTS parameter field are unpredictable if any of the following situations occur:

1. The cryptographic-key field or the encrypted cryptographic-key field overlaps any portion of the first operand.
2. The XTS parameter field overlaps any portion of the first operand.
3. The chaining-value field overlaps any portion of the first operand or the second operand.
4. The first and second operands overlap destructively. Operands are said to overlap destructively when the first-operand location would be used as a source after data would have been moved into it, assuming processing to be performed from left to right and one byte at a time.

When the operation ends due to normal completion, condition code 0 is set and the resulting value in $R_2 + 1$ is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in $R_2 + 1$ is nonzero.

A PER storage-alteration event may be recognized both for the first-operand location and for the portion of the parameter block that is stored. A PER zero-address-detection event may be recognized for the first- and second-operand locations and for the parameter block. When PER events are detected for one or more of these locations, it is unpredictable which location is identified in the PER access identification (PAID) and PER ASCE ID (AI).

When a storage-alteration PER event is recognized, fewer than 4K additional bytes are stored into the first-operand locations before the event is reported.

When the second-operand length is initially zero, the parameter block, first, and second operands are not accessed, general registers R_1, R_2, and $R_2 + 1$ are not changed, and condition code 0 is set.

When the contents of the R_1 and R_2 fields are the same, the contents of the designated registers are incremented only by the number of bytes processed, not by twice the number of bytes processed.

As observed by this CPU, other CPUs, and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

In certain unusual situations, instruction execution may complete by setting condition code 3 without updating the registers, the chaining value, and the XTS parameter to reflect the last unit of the first and second operands processed. The size of the unit processed in this case depends on the situation and the model, but is limited such that the portion of the first and second operands which have been processed and not reported do not overlap in storage. In all cases, change bits are set and PER storage-alteration events are reported, when applicable, for all first-operand locations processed.

For functions that perform a comparison of the encrypted cryptographic key with the wrapping-key verification-pattern register, it is unpredictable whether access exceptions and PER zero-address-detection events are recognized for the first and second operands when the comparison results in a mismatch.

Access exceptions may be reported for a larger portion of an operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of an operand nor for locations more than 4K bytes beyond the current location being processed.

Symbols Used in Function Descriptions

The following symbols are used in the subsequent description of the CIPHER MESSAGE and CIPHER MESSAGE WITH CHAINING functions. For data-encryption-algorithm (DEA) functions, the DEA-key-parity bit in each byte of the DEA key is ignored, and the operation proceeds normally, regardless of the DEA-key parity of the key.

Further description of the data-encryption algorithm may be found in Reference [13.] on page xxx. Further description of the AES standard may be found in Reference [14.] on page xxx. The XTS multiplication operation is the same as the GCM (Galois/counter mode) multiplication operation. Further description of

the GCM multiplication over GF(2^{128}) may be found in Reference [17.] on page xxx.

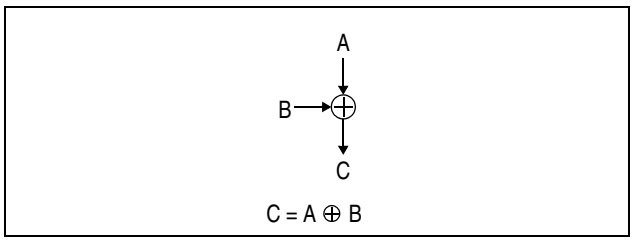


Figure 7-8. Symbol For Bit-Wise Exclusive OR

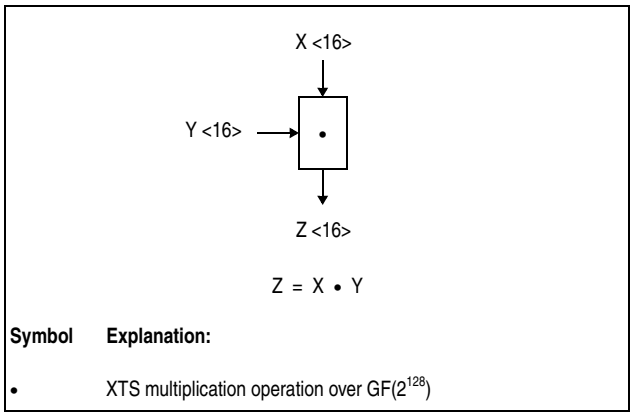


Figure 7-9. Symbol For XTS Multiplication Operation Over GF(2^{128})

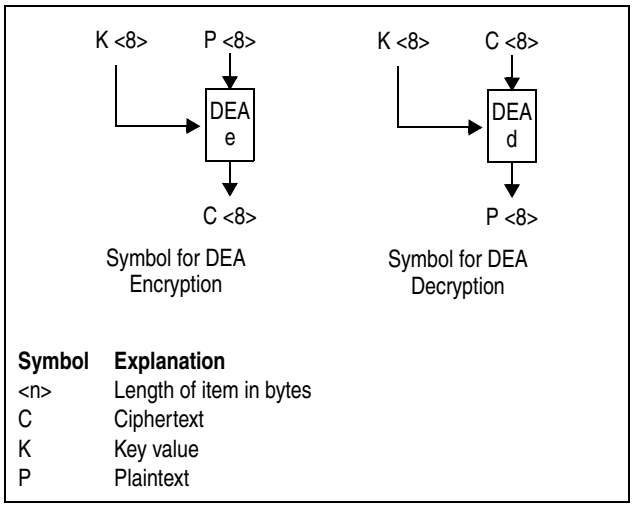


Figure 7-10. Symbols for DEA Encryption and Decryption

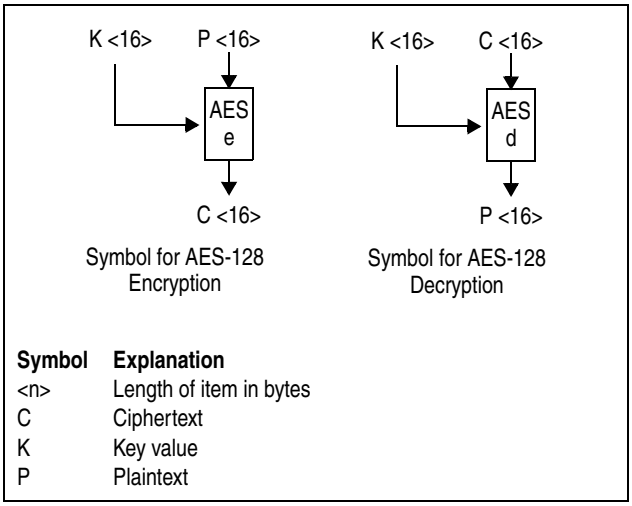


Figure 7-11. Symbols for AES-128 Encryption and Decryption

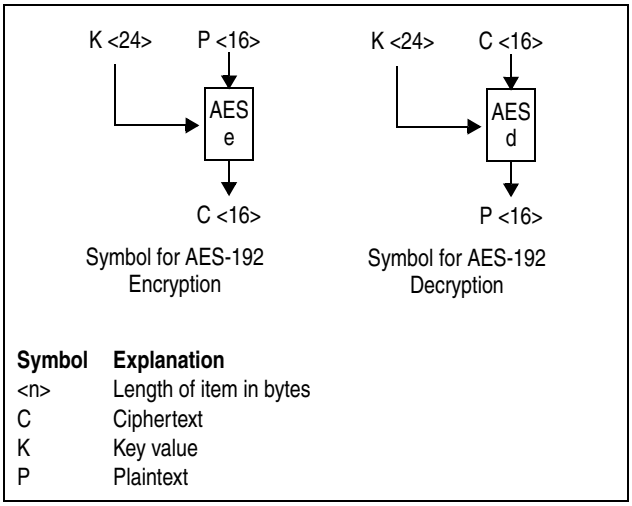


Figure 7-12. Symbols for AES-192 Encryption and Decryption

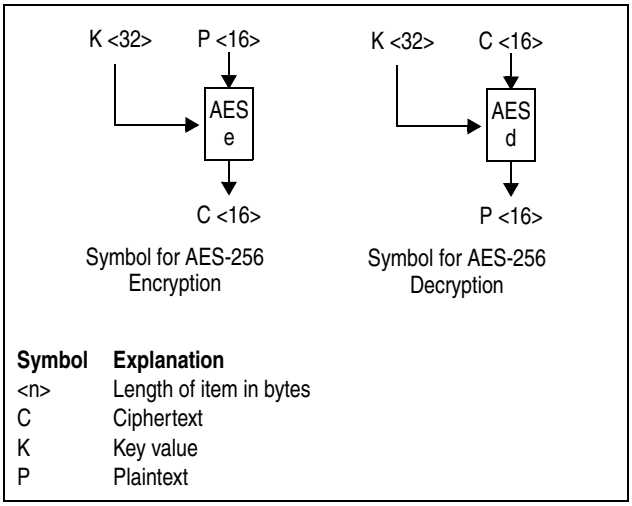


Figure 7-13. Symbols for AES-256 Encryption and Decryption

KM-Query (KM Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the function has the following format:

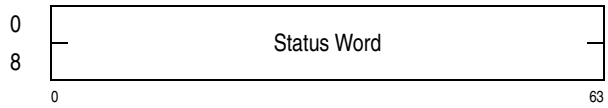


Figure 7-14. Parameter Block for KM-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the CIPHER MESSAGE instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KM-Query function completes; condition codes 1 and 3 are not applicable to this function.

KM-DEA (KM Function Code 1)

KM-Encrypted-DEA (KM Function Code 9)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KM-DEA function has the following format:

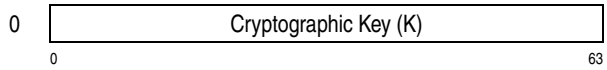


Figure 7-15. Parameter Block for KM-DEA

For the KM-DEA function, the cryptographic key is in byte offsets 0-7 of the parameter block.

The parameter block used for the KM-Encrypted-DEA function has the following format:

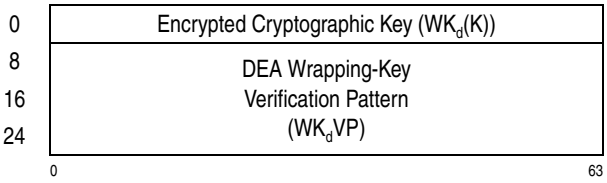


Figure 7-16. Parameter Block for KM-Encrypted-DEA

For the KM-Encrypted-DEA function, the contents of byte offsets 8-31 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 0-7 of the parameter block are deciphered using the DEA wrapping key to obtain the 64-bit cryptographic key. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the DEA algorithm with the 64-bit cryptographic key. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, …,

Cn) are stored in operand 1. The operation is shown in Figure 7-17.

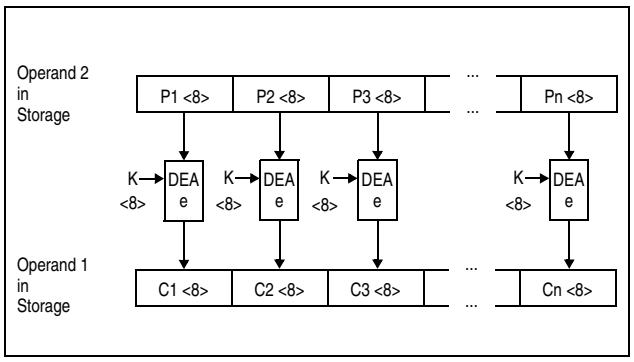


Figure 7-17. KM-DEA Encipher Operation Using 64-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the DEA algorithm with the 64-bit cryptographic key. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The operation is shown in Figure 7-18.

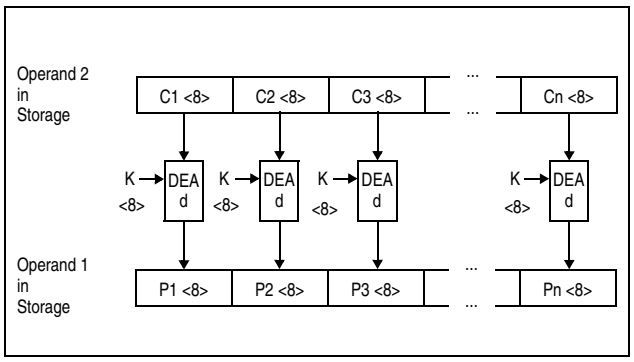


Figure 7-18. KM-DEA Decipher Operation Using 64-Bit Key

KM-TDEA-128 (KM Function Code 2)

KM-Encrypted-TDEA-128 (KM Function Code 10)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KM-TDEA-128 function has the following format:

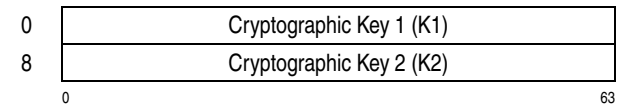


Figure 7-19. Parameter Block for KM-TDEA-128

For the KM-TDEA-128 function, the cryptographic key is in byte offsets 0-15 of the parameter block.

The parameter block used for the KM-Encrypted-TDEA-128 function has the following format:

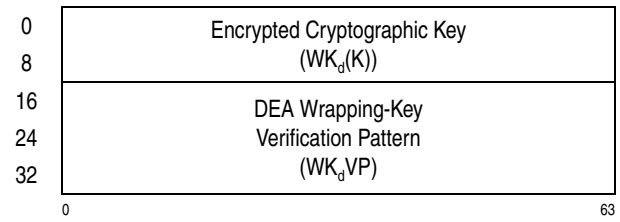


Figure 7-20. Parameter Block for KM-Encrypted-TDEA-128

For the KM-Encrypted-TDEA-128 function, the contents of byte offsets 16-39 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 0-15 of the parameter block are deciphered using the DEA wrapping key to obtain the 128-bit cryptographic key, K = K1 || K2, where K1 is the leftmost 64 bits of K and K2 is the rightmost 64 bits of K. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the TDEA (triple DEA) algorithm with the two 64-bit cryptographic keys. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext

blocks (C1, C2, …, Cn) are stored in operand 1. The operation is shown in Figure 7-21.

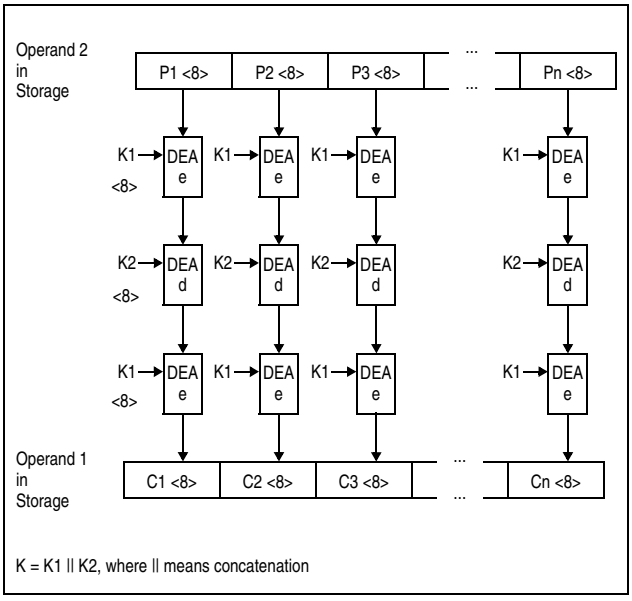


Figure 7-21. KM-TDEA Encipher Operation Using 128-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the TDEA algorithm with the two 64-bit cryptographic keys. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The operation is shown in Figure 7-22.

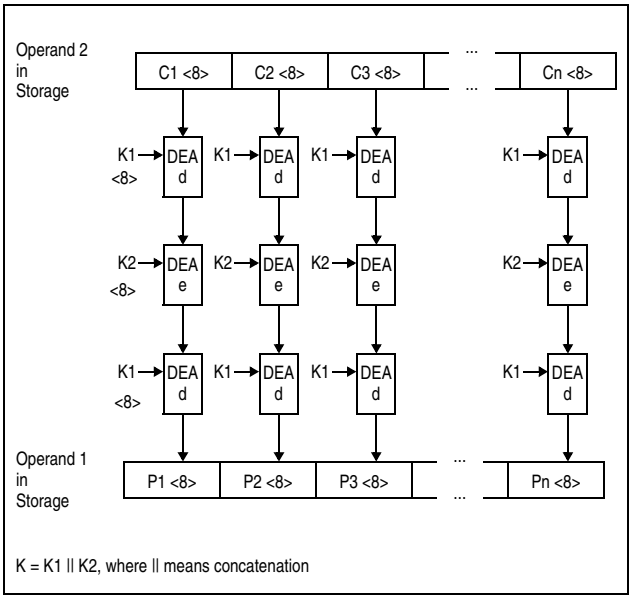


Figure 7-22. KM TDEA Decipher Operation Using 128-Bit Key

KM-TDEA-192 (KM Function Code 3)

KM-Encrypted-TDEA-192 (KM Function Code 11)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KM-TDEA-192 function has the following format:

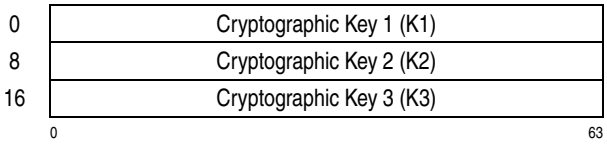


Figure 7-23. Parameter Block for KM-TDEA-192

For the KM-TDEA-192 function, the cryptographic key is in byte offsets 0-23 of the parameter block.

The parameter block used for the KM-Encrypted-TDEA-192 function has the following format:

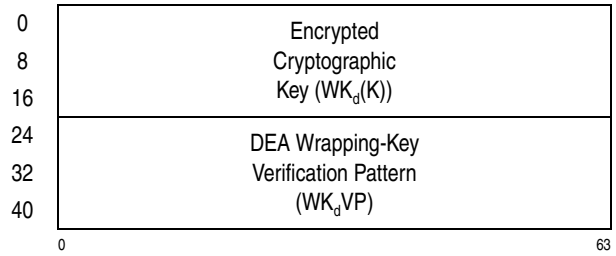


Figure 7-24. Parameter Block for KM-Encrypted-TDEA-192

For the KM-Encrypted-TDEA-192 function, the contents of byte offsets 24-47 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 0-23 of the parameter block are deciphered using the DEA wrapping key to obtain the 192-bit cryptographic key, K = K1 || K2 || K3, where K1 is the leftmost 64 bits of K, K2 is the middle 64 bits of K, and K3 is the rightmost 64 bits of K. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the TDEA algorithm with the three 64-bit cryptographic keys. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The operation is shown in Figure 7-25.

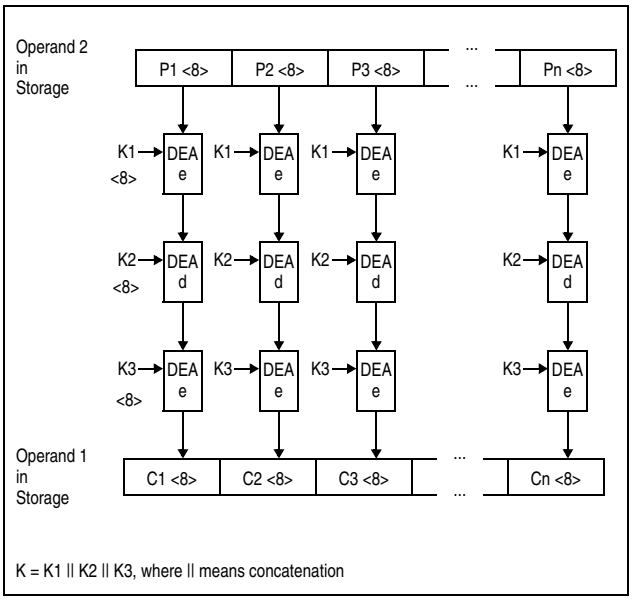


Figure 7-25. KM TDEA Encipher Operation Using 192-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the TDEA algorithm with the three 64-bit cryptographic keys. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext

blocks (P1, P2, …, Pn) are stored in operand 1. The operation is shown in Figure 7-26.

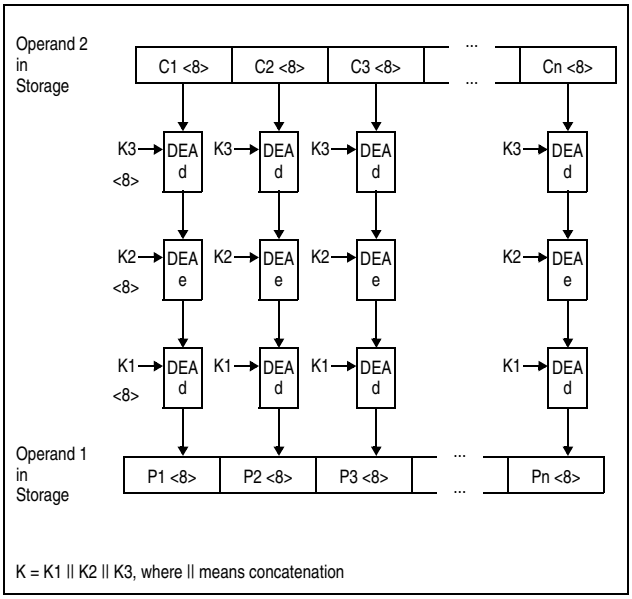


Figure 7-26. KM TDEA Decipher Operation Using 192-Bit Key

KM-AES-128 (KM Function Code 18)

KM-Encrypted-AES-128 (KM Function Code 26)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KM-AES-128 function has the following format:

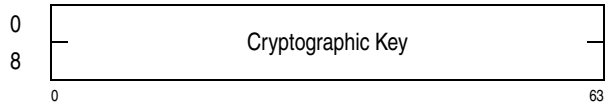


Figure 7-27. Parameter Block for KM-AES-128

For the KM-AES-128 function, the cryptographic key is in byte offsets 0-15 of the parameter block.

The parameter block used for the KM-encrypted-AES-128 function has the following format:

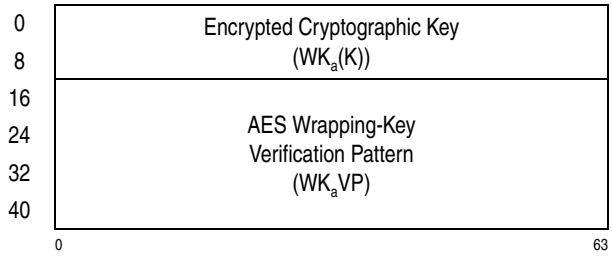


Figure 7-28. Parameter Block for KM-Encrypted-AES-128

For the KM-Encrypted-AES-128 function, the contents of byte offsets 16-47 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 0-15 of the parameter block are deciphered using the AES wrapping key to obtain the 128-bit cryptographic key, K. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES algorithm with the 128-bit cryptographic key. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The operation is shown in Figure 7-29.

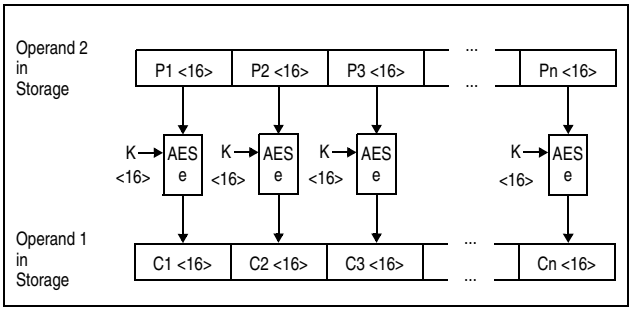


Figure 7-29. KM-AES Encipher Operation Using 128-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deci-

phered using the AES algorithm with the 128-bit cryptographic key. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The operation is shown in Figure 7-30.

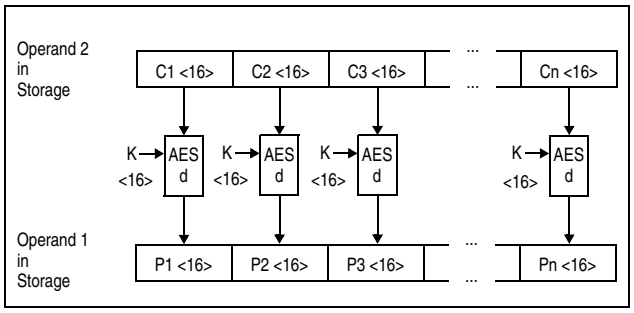


Figure 7-30. KM-AES Decipher Operation Using 128-Bit Key

KM-AES-192 (KM Function Code 19)

KM-Encrypted-AES-192 (KM Function Code 27)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KM-AES-192 function has the following format:

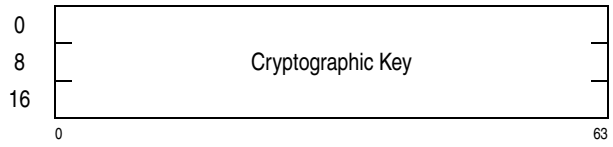


Figure 7-31. Parameter Block for KM-AES-192

For the KM-AES-192 function, the cryptographic key is in byte offsets 0-23 of the parameter block.

The parameter block used for the KM-Encrypted-AES-192 function has the following format:

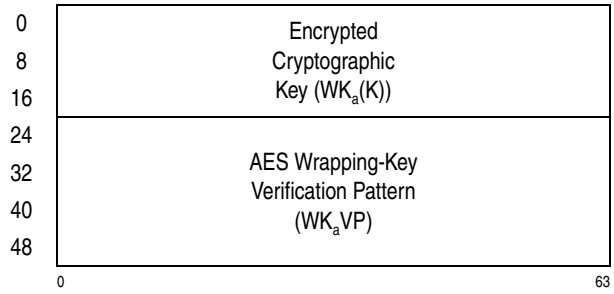


Figure 7-32. Parameter Block for KM-Encrypted-AES-192

For the KM-Encrypted-AES-192 function, the contents of byte offsets 24-55 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 0-23 of the parameter block are deciphered using the AES wrapping key to obtain the 192-bit cryptographic key, K. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES algorithm with the 192-bit cryptographic key. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The operation is shown in Figure 7-33.

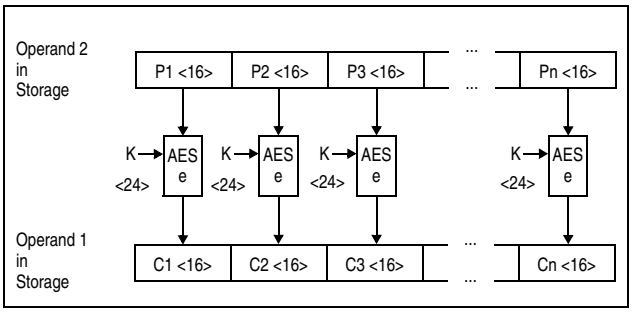


Figure 7-33. KM AES Encipher Operation Using 192-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES algorithm with the 192-bit cryptographic key. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The operation is shown Figure 7-34.

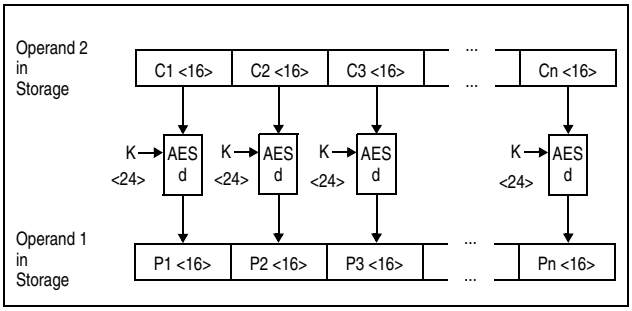


Figure 7-34. KM AES Decipher Operation Using 192-Bit Key

KM-AES-256 (KM Function Code 20)

KM-Encrypted-AES-256 (KM Function Code 28)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KM-AES-256 function has the following format:

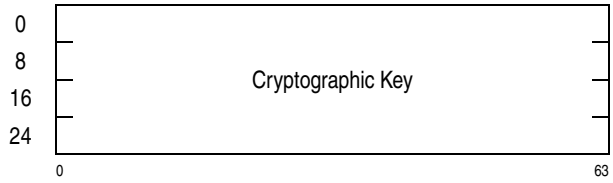


Figure 7-35. Parameter Block for KM-AES-256

For the KM-AES-256 function, the cryptographic key is in byte offsets 0-31 of the parameter block.

The parameter block used for the KM-Encrypted-AES-256 function has the following format:

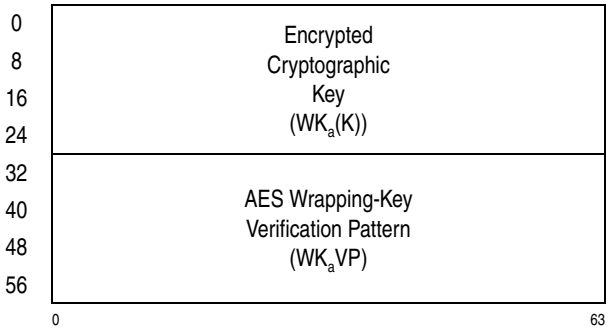


Figure 7-36. Parameter Block for KM-Encrypted-AES-256

For the KM-Encrypted-AES-256 function, the contents of byte offsets 32-63 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 0-31 of the parameter block are deciphered using the AES wrapping key to obtain the 256-bit cryptographic key, K. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES algorithm with the 256-bit cryptographic key. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The operation is shown in Figure 7-37.

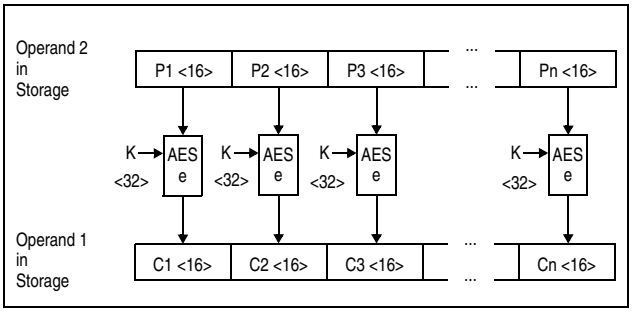


Figure 7-37. KM AES Encipher Operation Using 256-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES algorithm with the 256-bit cryptographic key. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The operation is shown in Figure 7-38.

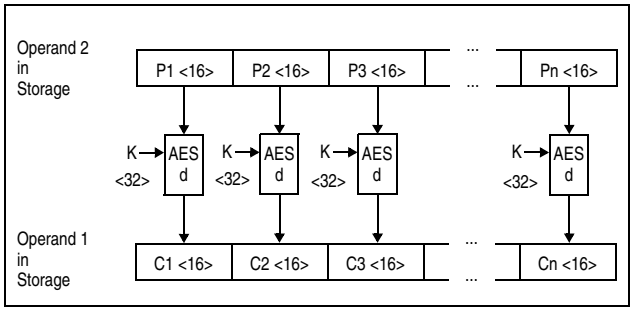


Figure 7-38. KM AES Decipher Operation Using 256-Bit Key

KM-XTS-AES-128 (KM Function Code 50)

KM-XTS-Encrypted-AES-128 (KM Function Code 58)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KM-XTS-AES-128 function has the following format:

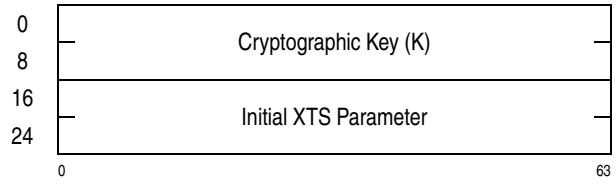


Figure 7-39. Parameter Block for KM-XTS-AES-128

For the KM-XTS-AES-128 function, the cryptographic key is in byte offsets 0-15 of the parameter block, and the initial XTS parameter is in bytes offsets 16-31 of the parameter block.

The parameter block used for the KM-XTS-encrypted-AES-128 function has the following format:

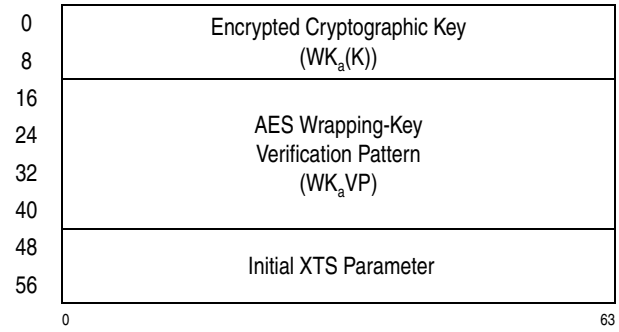


Figure 7-40. Parameter Block for KM-XTS-Encrypted-AES-128

For the KM-XTS-Encrypted-AES-128 function, the contents of byte offsets 16-47 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 48-63 of the parameter block contain the initial XTS parameter, and the contents of byte offsets 0-15 of the parameter block are deciphered using the AES wrapping key to obtain the 128-bit cryptographic key. (See the section “Protection of Cryptographic Keys” on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES-encryption algorithm with the 128-bit cryptographic key and the 128-bit initial XTS parameter.

Except for the first block, the XTS parameter used is the XTS parameter for the previous block multiplied by the value of 2 in GF(2^{128}). To encrypt the first block of plaintext, the initial XTS parameter is used.

The XTS parameter for each block is exclusive-ORed with the corresponding plaintext block. The result of the exclusive-OR operation is then encrypted using the AES-encryption algorithm with the 128-bit cryptographic key. The result of the encryption is then exclusive-ORed with the XTS parameter to produce the ciphertext block. The XTS parameter for this block is multiplied by the value of 2 in GF(2^{128}) to

obtain the XTS parameter for the next block. The operation is shown in Figure 7-41.

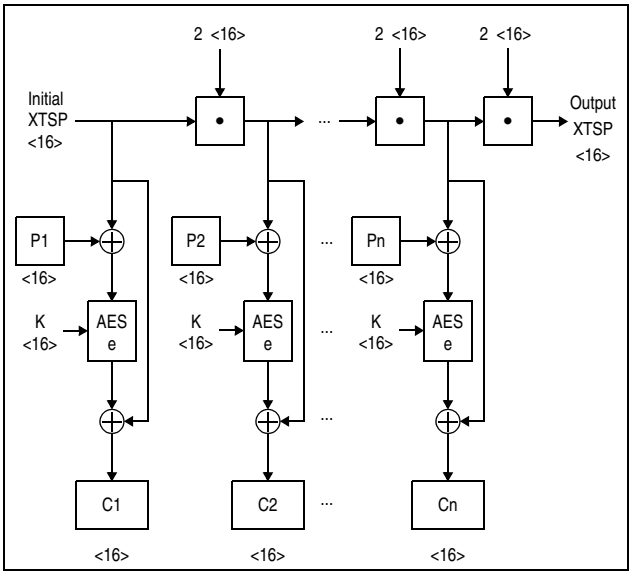


Figure 7-41. KM-XTS-AES-128 Encipher Operation

The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The result of the final GCM multiplication, called the output XTS parameter (output XTSP), is stored into the initial XTS parameter field of the parameter block.

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES-decryption algorithm with the 128-bit cryptographic key and the 128-bit initial XTS parameter.

Except for the first block, the XTS parameter used is the XTS parameter for the previous block multiplied by the value of 2 in GF(2^{128}). To decrypt the first block of ciphertext, the initial XTS parameter is used.

The XTS parameter for each block is exclusive-ORed with the corresponding ciphertext block. The result of the exclusive-OR operation is then decrypted using the AES-decryption algorithm with the 128-bit cryptographic key. The result of the encryption is then exclusive-ORed with the XTS parameter to produce the plaintext block. The XTS parameter for this block is multiplied by the value of 2 in GF(2^{128}) to obtain the XTS parameter for the next block. The operation is shown in Figure 7-42.

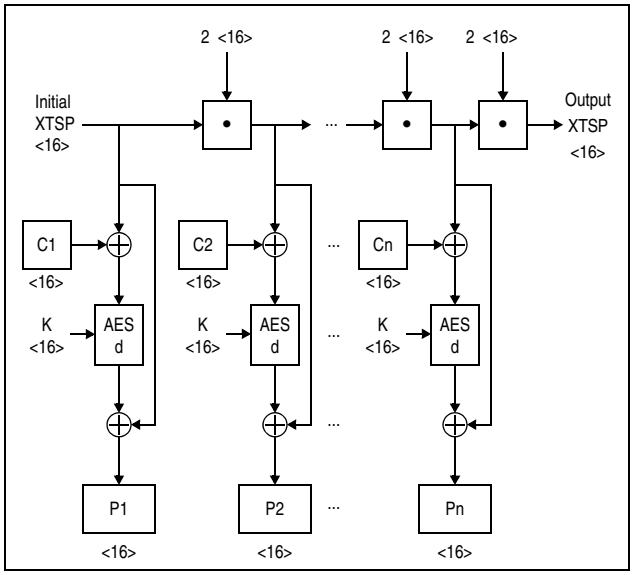


Figure 7-42. KM-XTS-AES-128 Decipher Operation

The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The result of the final GCM multiplication, called the output XTS parameter (output XTSP), is stored into the initial XTS parameter field of the parameter block.

KM-XTS-AES-256 (KM Function Code 52)

KM-XTS-Encrypted-AES-256 (KM Function Code 60)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KM-XTS-AES-256 function has the following format:

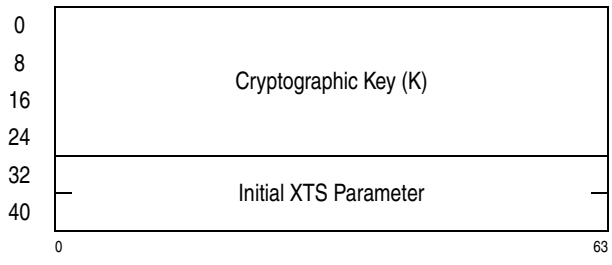


Figure 7-43. Parameter Block for KM-XTS-AES-256

For the KM-XTS-AES-256 function, the cryptographic key is in byte offsets 0-31 of the parameter block, and

the initial XTS parameter is in bytes offsets 32-47 of the parameter block.

The parameter block used for the KM-XTS-encrypted-AES-256 function has the following format:

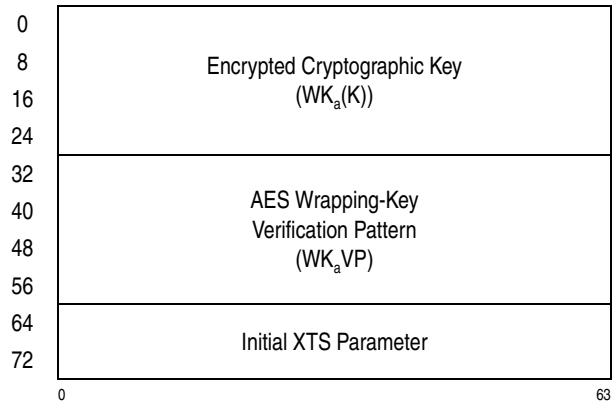


Figure 7-44. Parameter Block for KM-XTS-Encrypted-AES-256

For the KM-XTS-Encrypted-AES-256 function, the contents of byte offsets 32-63 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 64-79 of the parameter block contain the initial XTS parameter, and the contents of byte offsets 0-31 of the parameter block are deciphered using the AES wrapping key to obtain the 256-bit cryptographic key. (See the section "Protection of Cryptographic Keys" on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES-encryption algorithm with the 256-bit cryptographic key and the 128-bit initial XTS parameter.

Except for the first block, the XTS parameter used is the XTS parameter for the previous block multiplied by the value of 2 in GF(2^{128}). To encrypt the first block of plaintext, the initial XTS parameter is used.

The XTS parameter for each block is exclusive-ORed with the corresponding plaintext block. The result of the exclusive-OR operation is then encrypted using the AES-encryption algorithm with the 256-bit cryptographic key. The result of the encryption is then exclusive-ORed with the XTS parameter to produce the ciphertext block. The XTS parameter for this block is multiplied by the value of 2 in GF(2^{128}) to obtain the XTS parameter for the next block. The operation is shown in Figure 7-45.

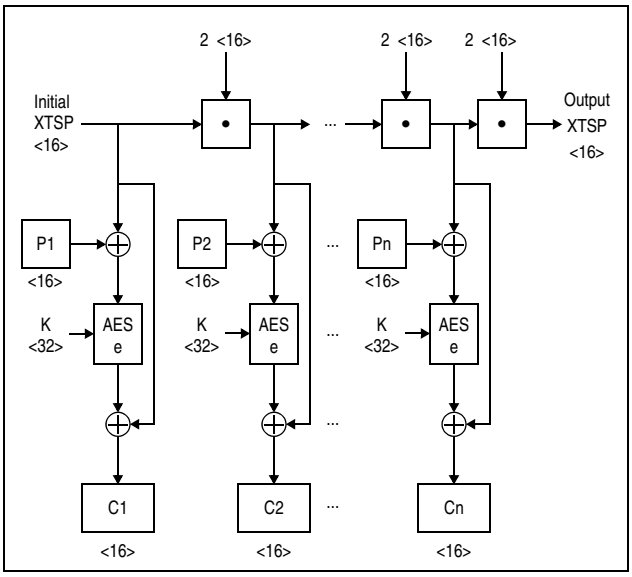


Figure 7-45. KM-XTS-AES-256 Encipher Operation

The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The result of the final GCM multiplication, called the output XTS parameter (output XTSP), is stored into the initial XTS parameter field of the parameter block.

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES-decryption algorithm with the 256-bit cryptographic key and the 128-bit initial XTS parameter.

Except for the first block, the XTS parameter used is the XTS parameter for the previous block multiplied by the value of 2 in GF(2^{128}). To decrypt the first block of ciphertext, the initial XTS parameter is used.

The XTS parameter for each block is exclusive-ORed with the corresponding ciphertext block. The result of the exclusive-OR operation is then decrypted using the AES-decryption algorithm with the 256-bit cryptographic key. The result of the encryption is then exclusive-ORed with the XTS parameter to produce the plaintext block. The XTS parameter for this block is multiplied by the value of 2 in GF(2^{128}) to obtain the

XTS parameter for the next block. The operation is shown in Figure 7-46.

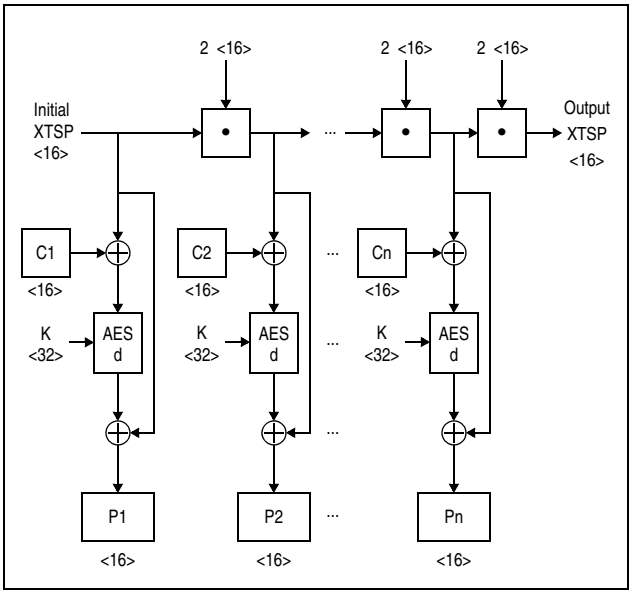


Figure 7-46. XTS-AES-256 Decipher Operation

The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The result of the final GCM multiplication, called the output XTS parameter (output XTSP), is stored into the initial XTS parameter field of the parameter block.

KMC-Query (KMC Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the function has the following format:

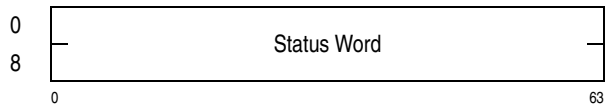


Figure 7-47. Parameter Block for KMC-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the CIPHER MESSAGE WITH CHAINING instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KMC-Query function completes; condition codes 1 and 3 are not applicable to this function.

KMC-DEA (KMC Function Code 1)

KMC-Encrypted-DEA (KMC Function Code 9)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KMC-DEA function has the following format:

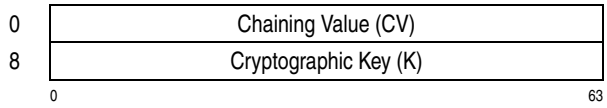


Figure 7-48. Parameter Block for KMC-DEA

For the KMC-DEA function, the chaining value is in byte offsets 0-7 of the parameter block and the cryptographic key is in byte offsets 8-15 of the parameter block.

The parameter block used for the KMC-Encrypted-DEA function has the following format:

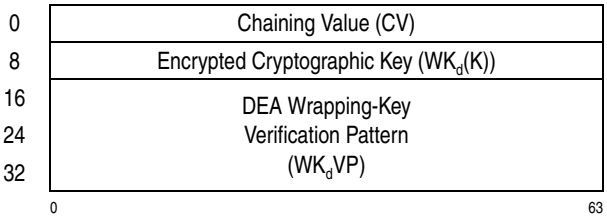


Figure 7-49. Parameter Block for KMC-Encrypted-DEA

For the KMC-Encrypted-DEA function, the contents of byte offsets 16-39 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 8-15 of the parameter block are deciphered using the DEA wrapping key to obtain the 64-bit cryptographic key. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.) The chaining value is in byte offsets 0-7 of the parameter block.

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered

using the DEA algorithm with the 64-bit cryptographic key and the 64-bit chaining value.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-50.

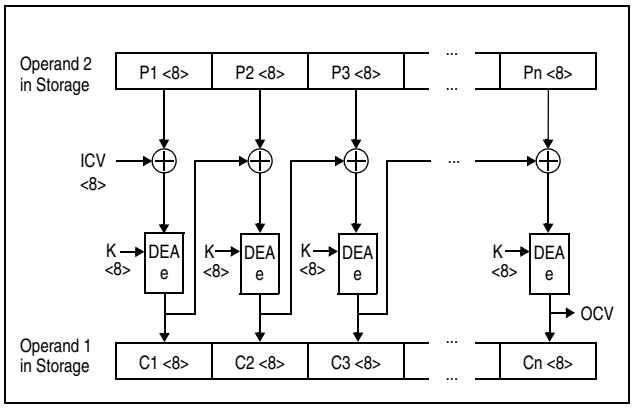


Figure 7-50. KMC DEA Encipher Operation Using 64-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the DEA algorithm with the 64-bit cryptographic key and the 64-bit chaining value.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The operation is shown in Figure 7-51.

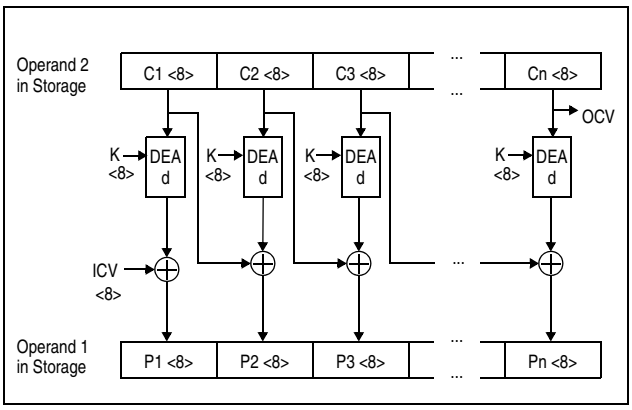


Figure 7-51. KMC DEA Decipher Operation Using 64-Bit Key

KMC-TDEA-128 (KMC Function Code 2)

KMC-Encrypted-TDEA-128 (KMC Function Code 10)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KMC-TDEA-128 function has the following format:

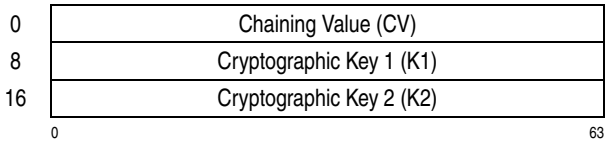


Figure 7-52. Parameter Block for KMC-TDEA-128

For the KMC-TDEA-128 function, the chaining value is in byte offsets 0-7 of the parameter block and the cryptographic key is in byte offsets 8-23 of the parameter block.

The parameter block used for the KMC-Encrypted-TDEA-128 function has the following format:

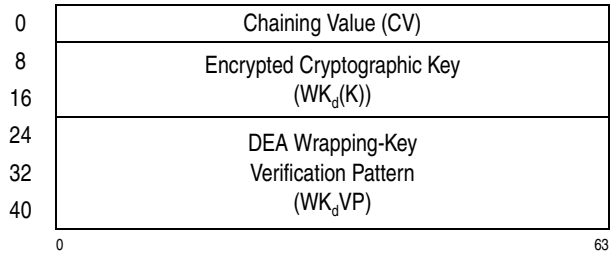


Figure 7-53. Parameter Block for KMC-Encrypted-TDEA-128

For the KMC-Encrypted-TDEA-128 function, the contents of byte offsets 24-47 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 8-23 of the parameter block are deciphered using the DEA wrapping key to obtain the 128-bit cryptographic key. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.) The chaining value is in byte offsets 0-7 of the parameter block.

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the TDEA algorithm with the two 64-bit cryptographic keys and the 64-bit chaining value.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-54.

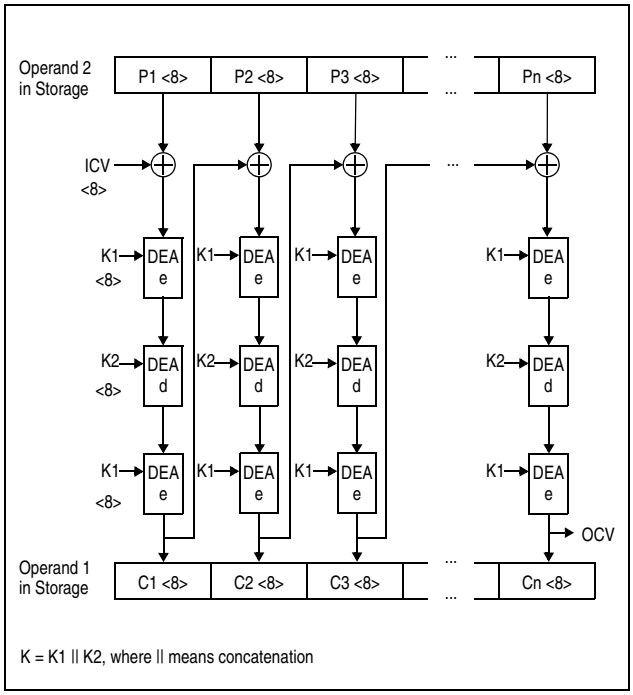


Figure 7-54. KMC TDEA Encipher Operation Using 128-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the TDEA algorithm with the two 64-bit cryptographic keys and the 64-bit chaining value.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is

stored into the chaining-value field in the parameter block. The operation is shown in Figure 7-55.

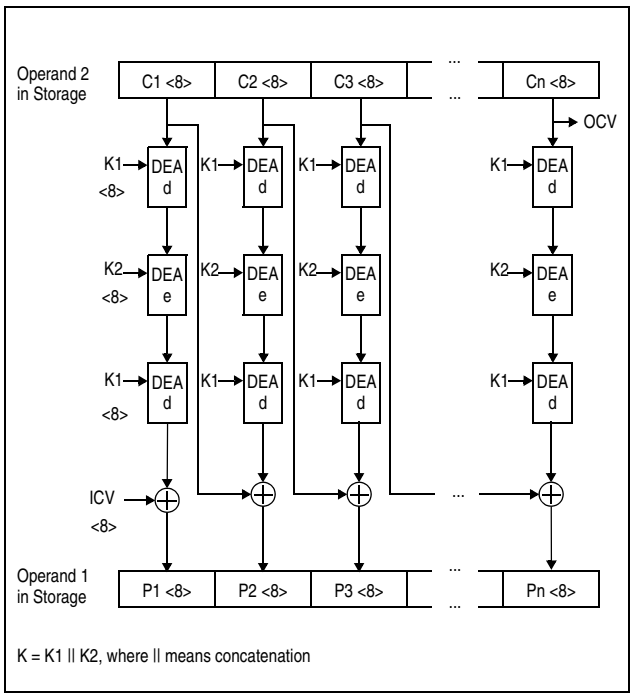


Figure 7-55. KMC TDEA Decipher Operation Using 128-Bit Key

KMC-TDEA-192 (KMC Function Code 3)

KMC-Encrypted-TDEA-192 (KMC Function Code 11)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KMC-TDEA-192 function has the following format:

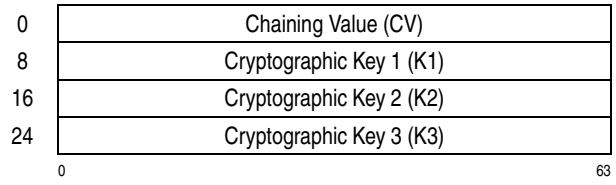


Figure 7-56. Parameter Block for KMC-TDEA-192

For the KMC-TDEA-192 function, the chaining value is in byte offsets 0-7 of the parameter block and the cryptographic key is in byte offsets 8-31of the parameter block.

The parameter block used for the KMC-Encrypted-TDEA-192 function has the following format:

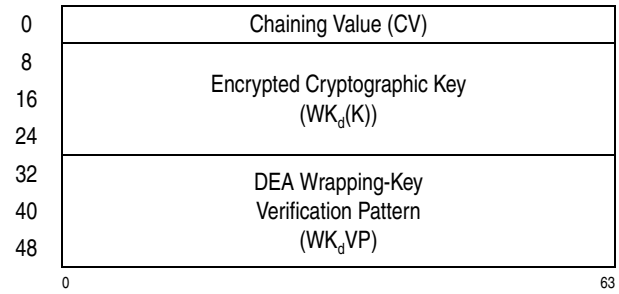


Figure 7-57. Parameter Block for KMC-Encrypted-TDEA-192

For the KMC-Encrypted-TDEA-192 function, the contents of byte offsets 32-55 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 8-31 of the parameter block are deciphered using the DEA wrapping key to obtain the 192-bit cryptographic key. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.) The chaining value is in byte offsets 0-7 of the parameter block.

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the TDEA algorithm with the three 64-bit cryptographic keys and the 64-bit chaining value.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value

field of the parameter block. The operation is shown in Figure 7-58.

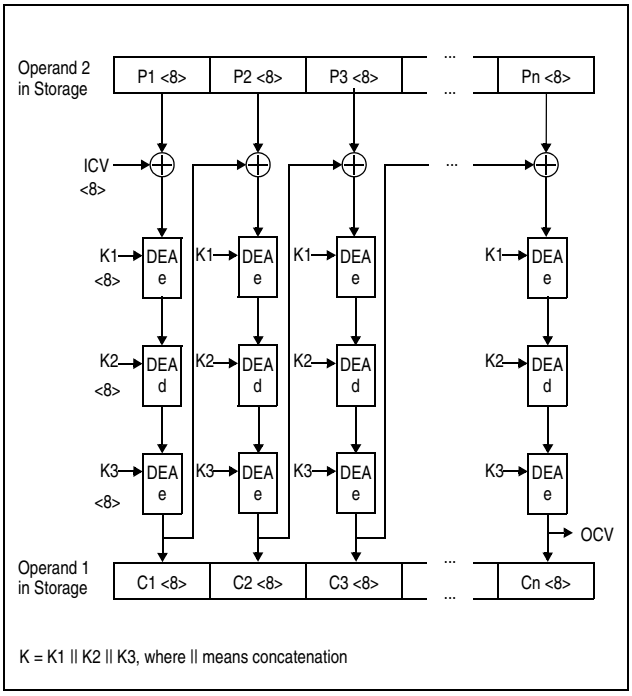


Figure 7-58. KMC TDEA Encipher Operation Using 192-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the TDEA algorithm with the three 64-bit cryptographic keys and the 64-bit chaining value.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The operation is shown in Figure 7-59.

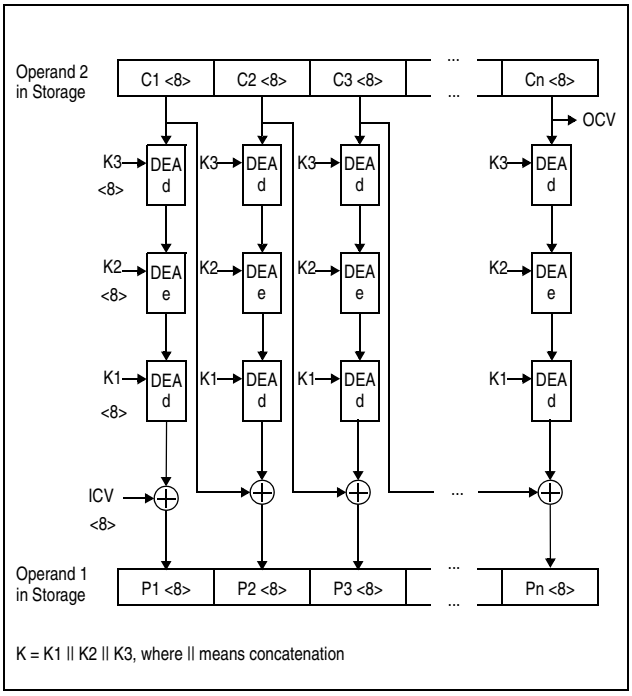


Figure 7-59. KMC TDEA Decipher Operation Using 192-Bit Key

KMC-AES-128 (KMC Function Code 18)

KMC-Encrypted-AES-128 (KMC Function Code 26)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KMC-AES-128 function has the following format:

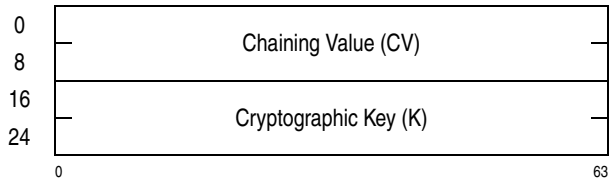


Figure 7-60. Parameter Block for KMC-AES-128

For the KMC-AES-128 function, the chaining value is in byte offsets 0-15 of the parameter block and the cryptographic key is in byte offsets 16-31 of the parameter block.

The parameter block used for the KMC-Encrypted-AES-128 function has the following format:

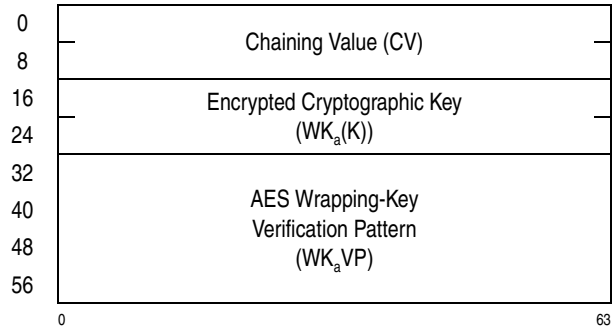


Figure 7-61. Parameter Block for KMC-Encrypted-AES-128

For the KMC-Encrypted-AES-128 function, the contents of byte offsets 32-63 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 16-31 of the parameter block are deciphered using the AES wrapping key to obtain the 128-bit cryptographic key, K. (See the section, "Protection of Cryptographic Keys" on page 7-424, for details.) The chaining value is in byte offsets 0-15 of the parameter block.

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES algorithm with the 128-bit cryptographic key and the 128-bit chaining value.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-62.

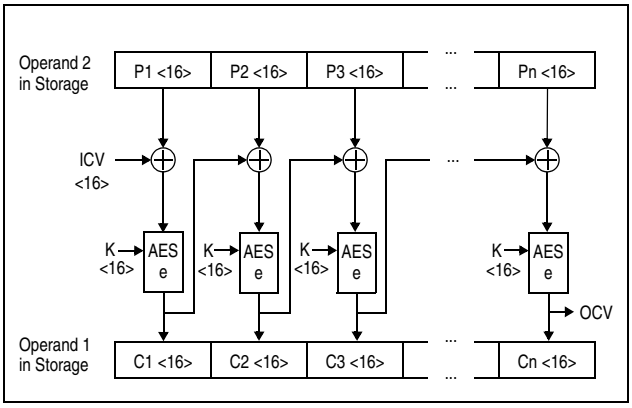


Figure 7-62. KMC AES Encipher Operation Using 128-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES algorithm with the 128-bit cryptographic key and the 128-bit chaining value.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The operation is shown in Figure 7-63.

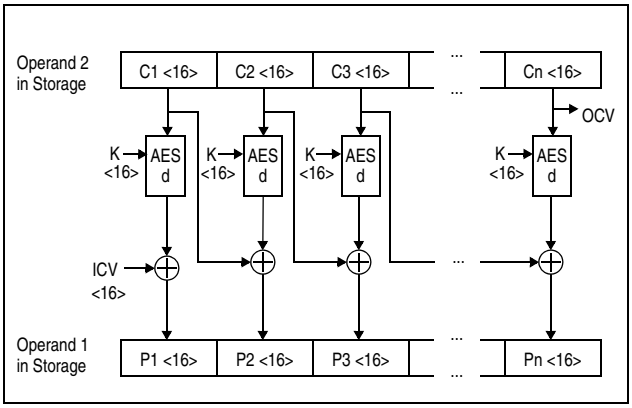


Figure 7-63. KMC AES Decipher Operation Using 128-Bit Key

KMC-AES-192 (KMC Function Code 19)

KMC-Encrypted-AES-192 (KMC Function Code 27)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KMC-AES-192 function has the following format:

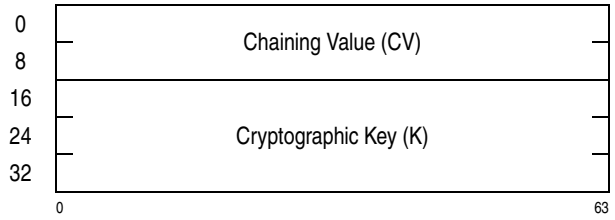


Figure 7-64. Parameter Block for KMC-AES-192

For the KMC-AES-192 function, the chaining value is in byte offsets 0-15 of the parameter block and the cryptographic key is in byte offsets 16-39 of the parameter block.

The parameter block used for the KMC-Encrypted-AES-192 function has the following format:

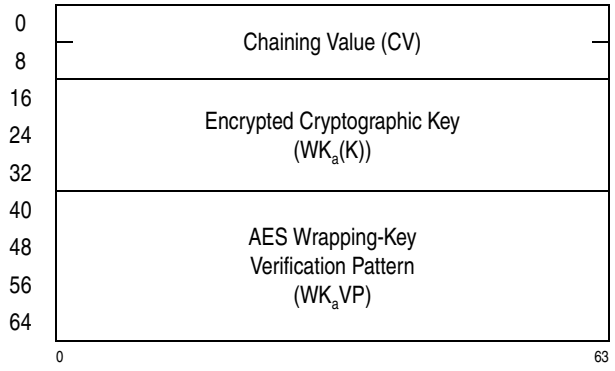


Figure 7-65. Parameter Block for KMC-Encrypted-AES-192

For the KMC-Encrypted-AES-192 function, the contents of byte offsets 40-71 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 16-39 of the parameter block are deciphered using the AES wrapping key to obtain the 192-bit cryptographic key, K. (See the section, "Protection of Cryptographic Keys" on page 7-424, for details.) The chaining value is in byte offsets 0-15 of the parameter block.

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES algorithm with the 192-bit cryptographic key and the 128-bit chaining value.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-66.

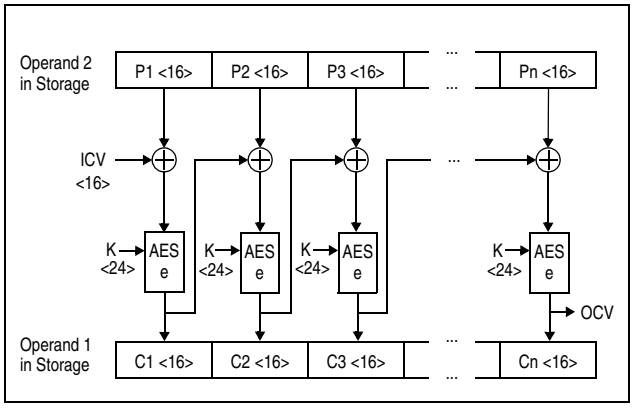


Figure 7-66. KMC AES Encipher Operation Using 192-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES algorithm with the 192-bit cryptographic key and the 128-bit chaining value.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is

stored into the chaining-value field in the parameter block. The operation is shown in Figure 7-67.

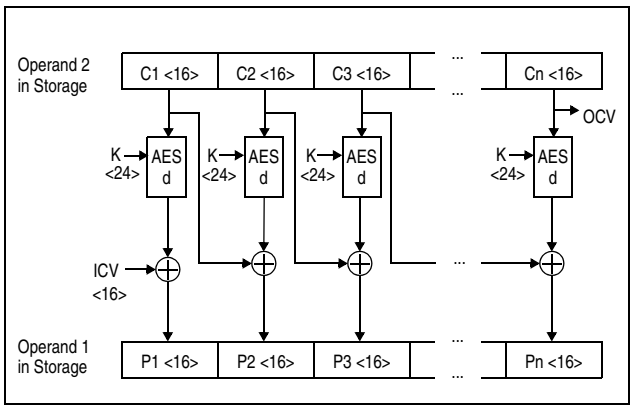


Figure 7-67. KMC AES Decipher Operation Using 192-Bit Key

KMC-AES-256 (KMC Function Code 20)

KMC-Encrypted-AES-256 (KMC Function Code 28)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KMC-AES-256 function has the following format:

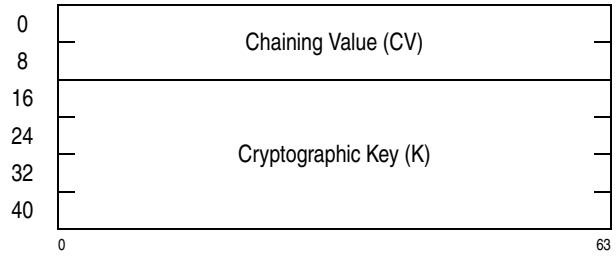


Figure 7-68. Parameter Block for KMC-AES-256

For the KMC-AES-256 function, the chaining value is in byte offsets 0-15 of the parameter block and the cryptographic key is in byte offsets 16-47 of the parameter block.

The parameter block used for the KMC-Encrypted-AES-256 function has the following format:

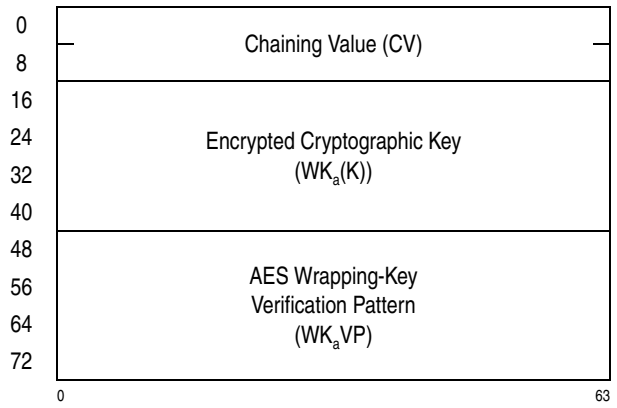


Figure 7-69. Parameter Block for KMC-Encrypted-AES-256

For the KMC-Encrypted-AES-256 function, the contents of byte offsets 48-79 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 16-47 of the parameter block are deciphered using the AES wrapping key to obtain the 256-bit cryptographic key, K. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.) The chaining value is in byte offsets 0-15 of the parameter block.

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES algorithm with the 256-bit cryptographic key and the 128-bit chaining value.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value

field of the parameter block. The operation is shown in Figure 7-70.

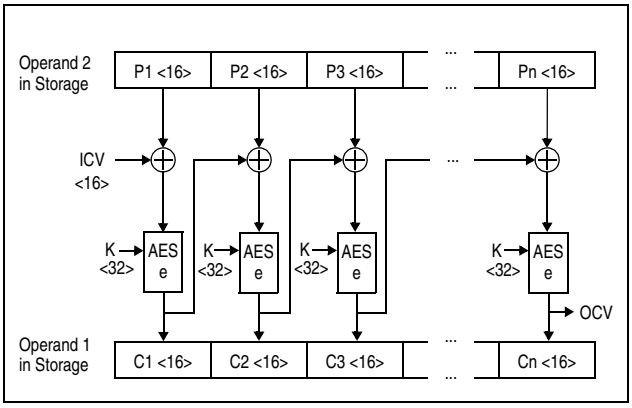


Figure 7-70. KMC AES Encipher Operation Using 256-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES algorithm with the 256-bit cryptographic key and the 128-bit chaining value.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The operation is shown in Figure 7-71.

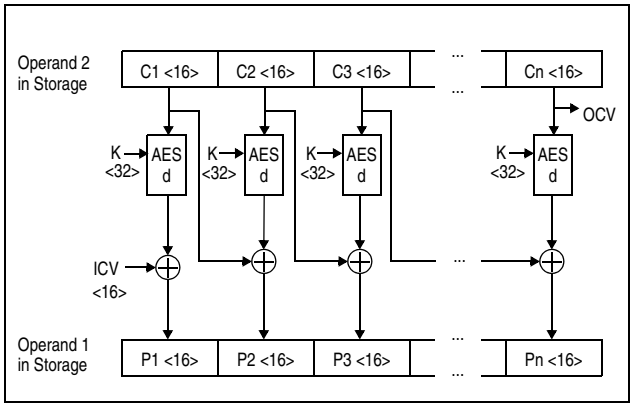


Figure 7-71. KMC AES Decipher Operation Using 256-Bit Key

KMC-PRNG (KMC Function Code 67)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the function has the following format:

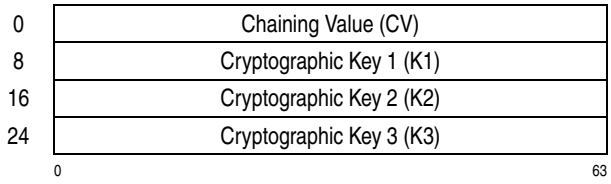


Figure 7-72. Parameter Block for KMC-PRNG

Regardless of the value of the modifier bit in general register 0, the 8-byte plaintext blocks in operand 2 are used to generate the 8-byte ciphertext blocks in operand 1 based on a pseudo-random-number-generation algorithm. The algorithm is based on TDEA using three 64-bit cryptographic keys and a 64-bit chaining value. The output chaining value (OCV) is

stored in the chaining-value field of the parameter block. The operation is shown in Figure 7-73.

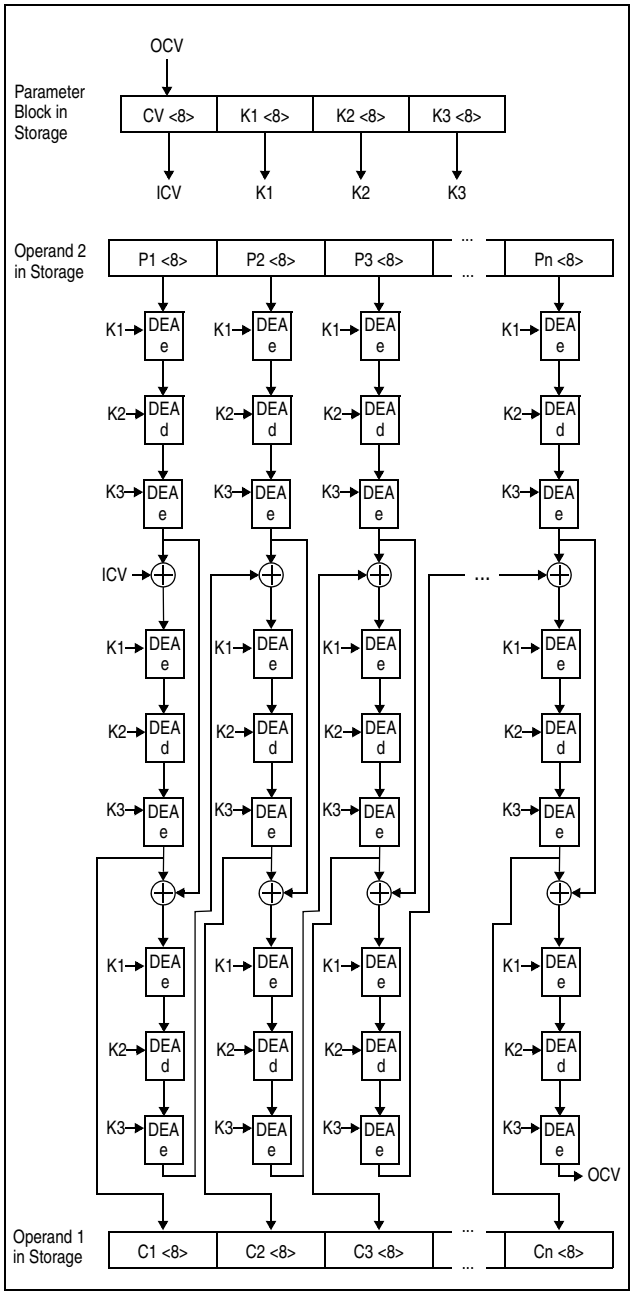


Figure 7-73. KMC-PRNG Operation

Special Conditions for KM and KMC

A specification exception is recognized and no other action is taken if any of the following occurs:

1. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
2. The R_1 or R_2 field designates an odd-numbered register or general register 0.
3. The second operand length is not a multiple of the data block size of the designated function (see Figure 7-5 on page 7-53 to determine the data block sizes for CIPHER MESSAGE functions; see Figure 7-6 on page 7-53 to determine the data block sizes for CIPHER MESSAGE WITH CHAINING functions). This specification-exception condition does not apply to the query functions.

Resulting Condition Code:

0 Normal completion
1 Verification-pattern mismatch
2 --
3 Partial completion

Program Exceptions:

- Access (fetch, operand 2, cryptographic key, and wrapping-key verification pattern; store, operand 1; fetch and store, chaining value, XTS parameter)
- Operation (if the message-security assist is not installed)
- Specification
- Transaction constraint

1.-6.	Exceptions with the same priority as the priority of program-interruption conditions for the general case.
7.A	Access exceptions for second instruction halfword.
7.B	Operation exception.
7.C	Transaction constraint
8.	Specification exception due to invalid function code or invalid register number.
9.	Specification exception due to invalid operand length.
10.	Condition code 0 due to second-operand length originally zero.
11.A.1	Access exceptions for an access to the parameter block.
11.A.2.	Condition code 1 due to verification-pattern mismatch.

Figure 7-74. Priority of Execution: KM and KMC

11.B	Access exceptions for an access to the first, or second operand.
12.	Condition code 0 due to normal completion (second-operand length originally nonzero, but stepped to zero).
13.	Condition code 3 due to partial completion (second-operand length still nonzero).

Figure 7-74. Priority of Execution: KM and KMC

Programming Notes:

1. The performance of the query function (code 0) may be significantly slower than that of simply examining a location in storage. If the program needs to frequently test for the availability of a function, it should perform the query function once during initialization; subsequently it should examine the stored results of the query function in memory with an instruction such as TEST UNDER MASK.

2. When condition code 3 is set, the general registers containing the operand addresses and length, and the chaining value or the XTS parameter in the parameter block, are usually updated such that the program can simply branch back to the instruction to continue the operation.

   For unusual situations, the CPU protects against endless reoccurrence of the no-progress case and also protects against setting condition code 3 when the portion of the first and second operands to be reprocessed overlap in storage. Thus, the program can safely branch back to the instruction whenever condition code 3 is set with no exposure to an endless loop and no exposure to incorrectly retrying the instruction.

3. If the length of the second operand is nonzero initially and condition code 0 is set, the registers are updated in the same manner as for condition code 3. For CIPHER MESSAGE WITH CHAINING, the chaining value in this case is such that additional operands can be processed as if they were part of the same chain.

4. To save storage, the first and second operands may overlap exactly or the starting point of the first operand may be to the left of the starting point of the second operand. In either case, the overlap is not destructive.

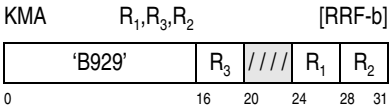
5. For XTS functions, the program must compute the initial XTS parameter by using the appropriate Compute-XTS-Parameter PCC functions before processing the first part of a message.

6. The initial XTS parameter used in the KM-XTS-AES-128 (or the KM-XTS-Encrypted-AES-128) function is computed by using the AES encryption algorithm with a 128-bit cryptographic key, which shall be different from the 128-bit cryptographic key used in the KM-XTS-AES-128 (or the KM-XTS-Encrypted-AES-128) function.

   Similarly, the initial XTS parameter used in the KM-XTS-AES-256 (or the KM-XTS-Encrypted-AES-256) function is computed by using the AES-encryption algorithm with a 256-bit cryptographic key, which shall be different from the 256- bit cryptographic key used in the KM-XTS-AES-256 (or the KM-XTS-Encrypted-AES-256) function.

7. As observed by this CPU, other CPUs, and the I/O subsystem, inconsistent results may be briefly stored in the first-operand location. See the section “Effects of CPU Retry” on page 11-3 for further details.

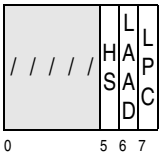
CIPHER MESSAGE WITH AUTHENTICATION



A function specified by the function code in general register 0 is performed.

General register 0 contains various controls affecting the operation of the instruction, as follows:

Flags (F): Bit positions 48-55 of general register 0 contain an 8-bit flags field controlling the operation of the function. The flags field is meaningful only when the function code in bits 57-63 of general register 0 designates a ciphering function (that is, when the function code is nonzero). The format of the flags field is as follows:



The individual flag fields are as follows:

Reserved: Bits 0-4 of the flags field are reserved and should contain zeros; otherwise, the program may not operate compatibly in the future.

Hash-subkey-Supplied (HS) Flag: Bit 5 of the flags field indicates whether the hash-subkey (H) field in the parameter block contains a valid hash subkey. The HS flag is meaningful only when the function code is nonzero, and the function does not use an encrypted cryptographic key; the HS flag is ignored for functions that use an encrypted cryptographic key.

Last-AAD (LAAD) Flag: Bit 6 of the flags field qualifies the contents of the third operand. When the LAAD flag is one, it indicates that the third operand designates the last series of additional-authenticated-data (AAD) blocks. When the LAAD flag is zero, it indicates that the third operand does not designate the last series of AAD blocks.

Last-Plaintext / Ciphertext (LPC) Flag: Bit 7 of the flags field qualifies the contents of the second operand. When the LPC flag is one, it indicates that the second operand designates the last series of plaintext or ciphertext blocks. When the LPC flag is zero, it indicates that the second operand does not designate the last series of plaintext or ciphertext blocks.

A specification exception is recognized, and the operation is suppressed when the LPC flag is one, and the LAAD flag is zero.

Modifier (M): When the function code in bits 57-63 of general register 0 is nonzero, bit position 56 of general register 0 contains a modifier bit indicating encryption or decryption is to be performed by the function. When the M bit is zero, the function performs encryption of the second operand; when the M bit is one, the function performs decryption of the second operand. The M bit is ignored when the function code is zero.

Function Code (FC): Bit positions 57-63 of general register 0 contain the function code. Figure 7-75 shows the assigned function codes for CIPHER MESSAGE WITH AUTHENTICATION. All other function codes are unassigned.

Code	Function	Param. Block Size (bytes)	Data Block Size (bytes)
0	KMA-Query	16	—
18	KMA-GCM-AES-128	96	16
19	KMA-GCM-AES-192	104	16
20	KMA-GCM-AES-256	112	16
26	KMA-GCM-Encrypted-AES-128	128	16
27	KMA-GCM-Encrypted-AES-192	136	16
28	KMA-GCM-Encrypted-AES-256	144	16

Explanation:

— Not applicable

Figure 7-75. Function Codes for CIPHER MESSAGE WITH AUTHENTICATION

Bits 0-31 of general register 0 are ignored. Bits 32-47 of general register 0 are reserved and should contain zeros; otherwise, the program may not operate compatibly in the future.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R_1, R_2, $R_2 + 1$, R_3, and $R_3 + 1$ are ignored for the query function.

For functions other then the query function (that is, for functions having a nonzero function code), a message-authentication tag is formed from the contents of the third operand and from the contents of either the resulting first operand or the second operand, depending on whether the M bit is 0 or 1, respectively. Based on the M bit, the second operand is either encrypted or decrypted using a cryptographic key and counter values from the parameter block, and the result is placed in the first-operand location.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register 1 constitute the address.

The R_1 field designates a general register and must designate an even-numbered register other than general register 0; otherwise, a specification exception is recognized. The R_2 and R_3 fields each designate an even-odd pair of general registers and must designate even-numbered registers other than general register 0; otherwise, a specification exception is recognized. A specification exception is also recognized if the R_3 field designates the same register as either the R_1 or R_2 fields.

The location of the leftmost byte of the first, second, and third operands is specified by the contents of general registers R_1, R_2, and R_3, respectively. The number of bytes in the second-operand location is specified in general register $R_2 + 1$. The first operand is the same length as the second operand. The number of bytes in the third-operand location is specified in general register $R_3 + 1$.

As part of the operation, the address in general register R_3 is incremented by the number of third-operand bytes processed, and the length in general register $R_3 + 1$ is decremented by the same number; additionally, the addresses in general registers R_1 and R_2 are each incremented by the number of second-operand bytes processed, and the length in general register $R_2 + 1$ is decremented by the same number. The formation and updating of the addresses and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1, R_2, and R_3 constitute the addresses of the first, second, and third operands, respectively, and the contents of bit positions 0-39 are ignored; bits 40-63 of the updated addresses replace the corresponding bits in general registers R_1, R_2, and R_3, carries out of bit position 40 of the updated address are ignored, and the contents of bit positions 32-39 of general registers R_1, R_2, and R_3 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general registers R_1, R_2, and R_3 constitute the addresses of the first, second, and third operands, respectively, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated addresses replace the corresponding bits in general registers R_1, R_2, and R_3, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general registers R_1, R_2, and R_3 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general registers R_1, R_2, and R_3 constitute the addresses of the first, second, and third operands, respectively; bits 0-63 of the updated addresses replace the contents of general registers R_1, R_2, and R_3, and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_2 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the first and second operands, and the contents of bit positions 0-31 are ignored; bits 32-63 of the updated value replace the corresponding bits in general register $R_2 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the first and second operands; the updated value replaces the contents of general register $R_2 + 1$.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_3 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the third operand, and the contents of bit positions 0-31 are ignored; bits 32-63 of the updated value replace the corresponding bits in general register $R_3 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_3 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the third operand; the updated value replaces the contents of general register $R_3 + 1$.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, R_2, $R_2 + 1$, R_3, and $R_3 + 1$ always remain unchanged.

Figure 7-126 on page 7-109 shows the contents of the general registers just described.

All Addressing Modes

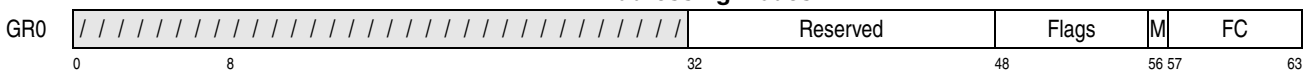


Figure 7-76. General Register Assignment for CIPHER MESSAGE WITH AUTHENTICATION (Part 1 of 2)

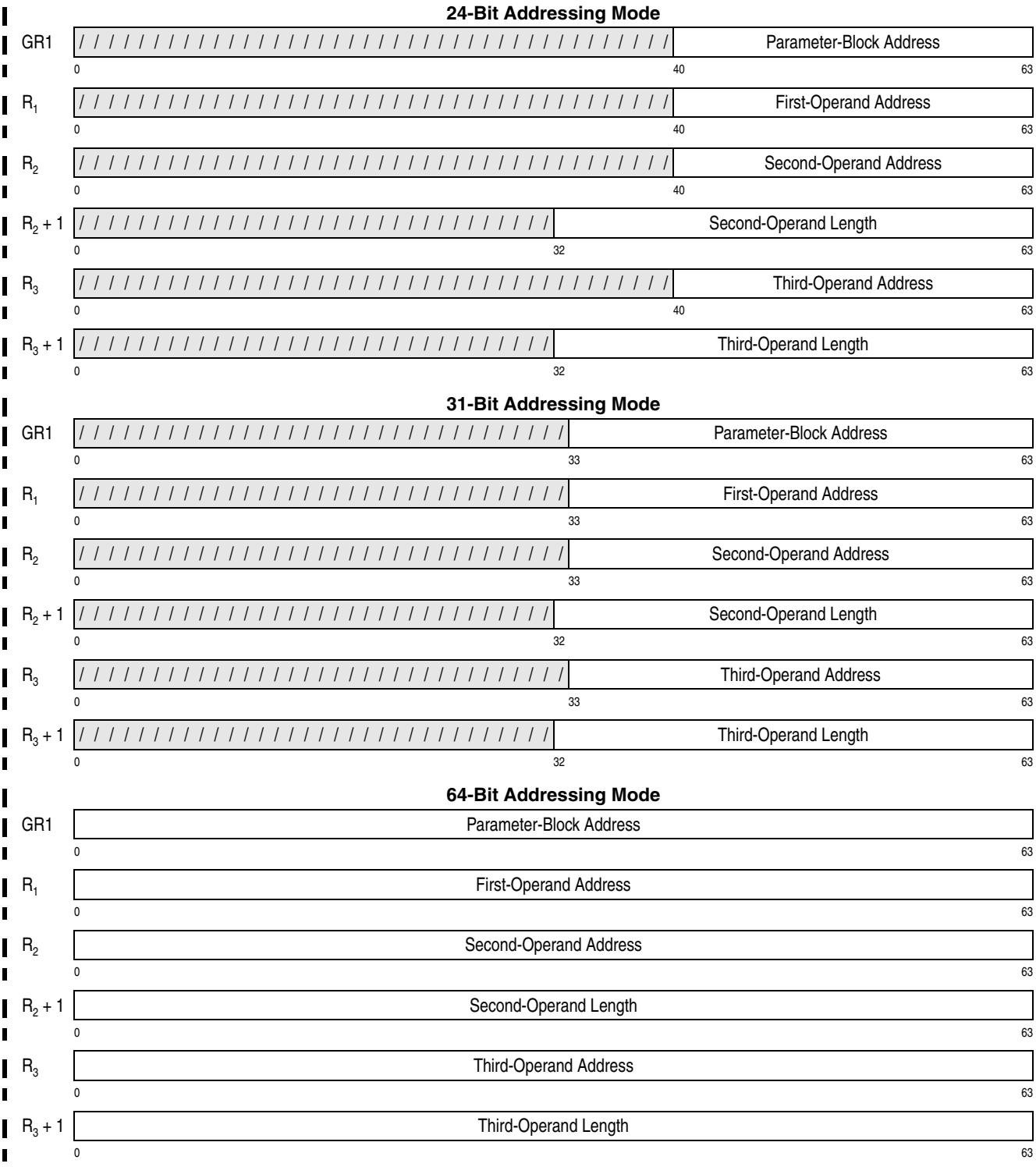


Figure 7-76. General Register Assignment for CIPHER MESSAGE WITH AUTHENTICATION (Part 2 of 2)

In the access-register mode, access registers 1, R_1, R_2, and R_3 specify the address spaces containing the parameter block, first, second, and third operands, respectively.

KMA-Query (Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-126 on page 7-109.

The parameter block used for the function has the following format:

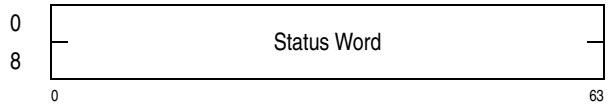


Figure 7-77. Parameter Block for KMA-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the CIPHER MESSAGE WITH AUTHENTICATION instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KMA-Query function completes; condition codes 1, 2, and 3 are not applicable to this function.

KMA-GCM-AES Functions

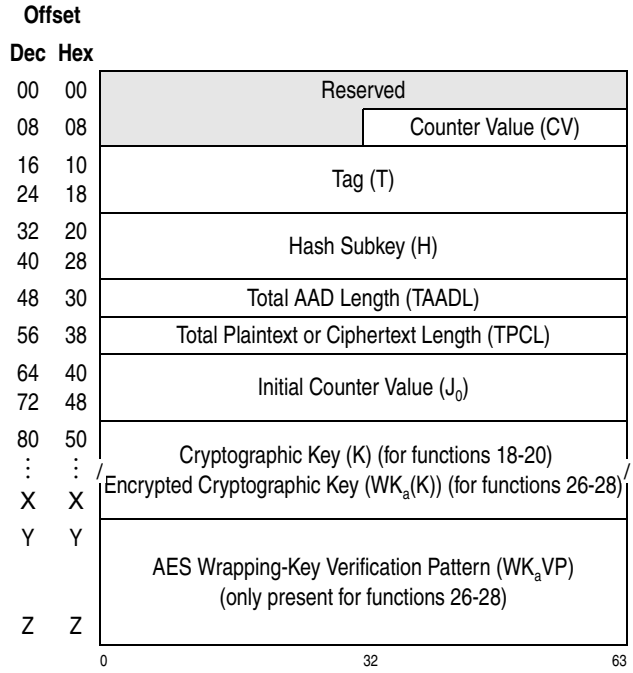
Note: The description of the KMA-GCM-AES instruction assumes that the reader is familiar with the Galois counter mode (GCM) described in Reference [17.] on page xxx. In particular, see [17] for a description of the GCM functions GHASH and GCTR.

This section illustrates the operation for six KMA-GCM-AES functions:

- KMA-GCM-AES-128 (function code 18)
- KMA-GCM-AES-192 (function code 19)
- KMA-GCM-AES-256 (function code 20)
- KMA-GCM-Encrypted AES-128 (function code 26)
- KMA-GCM-Encrypted AES-192 (function code 27)
- KMA-GCM-Encrypted AES-256 (function code 28)

The locations of the operands and addresses used by each of these functions are as shown in Figure 7-126 on page 7-109.

The parameter block used for all KMA-GCM-AES functions has a common format as shown in Figure 7-78, below.



Explanation:

X 72 (48 hex) plus the length of K (in bytes)
Y 80 (50 hex) plus the length of K (in bytes)
Z Y plus 24 (18 hex; the first byte of the last 8 bytes).

Figure 7-78. Parameter Block for KMA-GCM-AES Functions

The fields of the parameter block for all KMA-GCM-AES functions are as follows:

Reserved: Bytes 0-11 of the parameter block are reserved. The reserved field may contain model dependent values at the completion of the instruction.

Counter Value (CV): Bytes 12-15 of the parameter block contain a 32-bit binary integer. The leftmost 12 bytes of the initial-counter value (J_0, in bytes 64-79 of the parameter block) are concatenated with the contents of the CV field on the right to form the initial-counter block (ICB) that is used by the GCTR function (described in Reference [17.] on page xxx).

For each execution of the instruction, the CPU increments the CV field in the parameter block by the number of second-operand blocks processed. A carry out of bit position 0 of the CV field is ignored.

Tag (T): Bytes 16-31 of the parameter block contain the message-authentication tag field.

For each block of the third operand, and for each block of the resulting first operand (when M is 0) or each block of the second operand (when M is 1), the CPU uses the tag field as both input and output to the GHASH function. When all AAD and ciphertext has been hashed, the concatenation of the TAADL and TPCL fields are hashed using GHASH, and the results of the hash are encrypted using the GCTR function to produce a last tag (T) field in the parameter block.

Hash Subkey (H): For the KMA-GCM-AES functions, bytes 32-47 of the parameter block contain a 128-bit hash subkey that is used by the GHASH functions of the instruction. When the hash-subkey-supplied flag (HS, bit 53 of general register 0) is zero, the CPU computes the hash subkey by encrypting 128 bits of binary zeros using the cryptographic key (K), stores the hash subkey in the H field, and sets the HS flag to one. When the HS flag is one, the CPU uses the program-supplied hash subkey in the H field; the H field and HS flag are not altered in this case.

For the KMA-GCM-encrypted-AES functions, bytes 32-47 of the parameter block are reserved and should contain zeros; otherwise, the program may not operate compatibly in the future. In this case, the CPU always computes the hash subkey by encrypting 128 bits of binary zeros using the deciphered cryptographic key (K); the H field and HS flag are not altered in this case.

Total AAD Length (TAADL): Bytes 48-55 of the parameter block contain a 64-bit unsigned binary integer designating the total length in bits of the entire additional-authenticated-data (AAD) for the message being processed.

Total Plaintext or Ciphertext Length (TPCL): Bytes 56-63 of the parameter block contain a 64-bit unsigned binary integer designating the total length in bits of the entire plaintext or ciphertext for the message being processed.

Initial Counter Value (J_0): Bytes 64-79 of the parameter block contain a 128-bit initial counter value that is used (a) to provide the leftmost 96 bits of the initial-counter block used by the GCTR function, and (b) to encrypt the last authentication tag (T) field.

Cryptographic Key (K): The cryptographic key used in the encipher and decipher operations begins at byte 80 of the parameter block. The size of the key field and its offset in the parameter block are dependent on the function code, as shown in Figure 7-79.

Function Code	Function	Key Length (bytes)	Key Offset (bytes)
18	KMA-GCM-AES-128	16	80-95
19	KMA-GCM-AES-192	24	80-103
20	KMA-GCM-AES-256	32	80-111
26	KMA-GCM-Encrypted-AES-128	16	80-95
27	KMA-GCM-Encrypted-AES-192	24	80-103
28	KMA-GCM-Encrypted-AES-256	32	80-111

Figure 7-79. Key Lengths and Offsets for KMA-GCM Functions

AES Wrapping-Key Verification Pattern (WK_aVP): For the KMA-GCM-encrypted-AES functions (codes 26-28), the 32 bytes immediately following the key in the parameter block contain the AES wrapping-key verification pattern (WK_aVP).

For the KM-AES functions (codes 18-20), the WK_aVP field is not present in the parameter block.

Symbols Used in KMA-GCM-AES Function Descriptions

The following symbols are used in the subsequent description of the KMA-GCM-AES functions.

Further description of the AES standard may be found in Reference [14.] on page xxx. Further description of the Galois-counter mode of encryption and decryption may be found in Reference [17.] on page xxx.

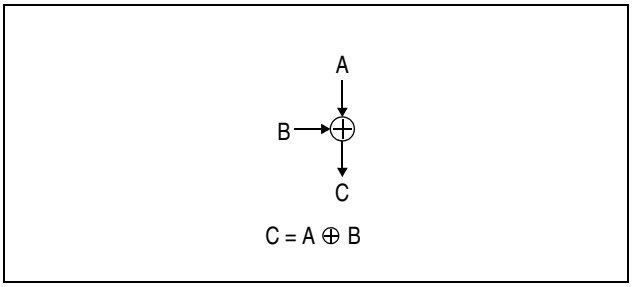
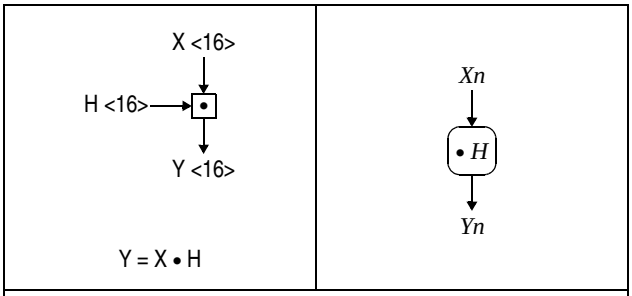


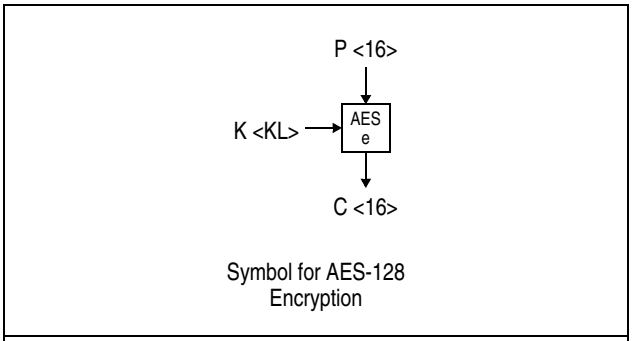
Figure 7-80. Symbol For Bit-Wise Exclusive OR



Symbol	Explanation
•	GCM multiplication operation over GF(2^{128})
<n>	Length of item in bytes.
H	Hash subkey

Note: The figure on the right illustrates the GCM multiplication of the hash subkey as illustrated in Reference [17.] on page xxx.

Figure 7-81. Symbols For GCM Multiplication Operation Over GF(2^{128})



Symbol	Explanation
<n>	Length of item in bytes.
C	Ciphertext.
K	Key value.
KL	Key length (see Figure 7-79).
P	Plaintext

Figure 7-82. Symbols for AES Encryption

The following description applies to all KMA-GCM-AES functions. A sequence of operations is performed, depending on the flags and function code in general register 0, as follows:

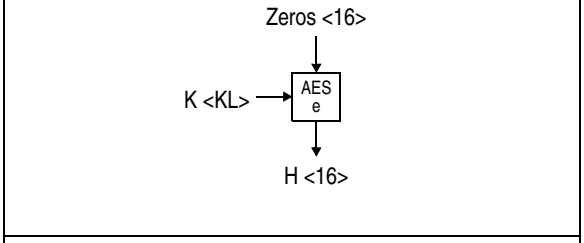
Wrapping-Key Verification: For the KMA-GCM-encrypted-AES functions (function codes 26-28), the contents of the 32-byte WK_aVP field are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of the key field of the parameter block are deciphered using the AES wrapping key to obtain the 128-bit cryptographic key, K. (See the section "Protection of Cryptographic Keys" on page 7-424 for details.)

For the KMA-GCM-AES functions that do not use encrypted keys (codes 18-20), wrapping-key verification is not performed.

Computing the Hash Subkey: A hash subkey is used in the GHASH processing to produce the resulting message-authentication tag. GHASH processing is described in Reference [17.] on page xxx.

For KMA-GCM-AES functions (function codes 18-20), the following applies:

- When the hash-subkey-supplied flag (HS, bit 53 of general register 0) is zero, a block of 128 binary zeros is encrypted using the AES algorithm described in Reference [14.] on page xxx. The AES algorithm uses the key (K) field from the parameter block, as shown in Figure 7-83, below.



Symbol	Explanation
<n>	Length of item in bytes
H	Hash subkey.
K	Key value.
KL	Key length (see Figure 7-79).

Figure 7-83. Encrypting the Hash Subkey

  The resulting 128-bit hash subkey is placed into the H field of the parameter block, and the HS flag is set to one in general register 0.

- When the HS flag is one, the H field in the parameter block is used as the hash subkey. In this case, the H field and HS flag are not altered.

For KMA-GCM-encrypted-AES functions (function codes 26-28), the HS flag is ignored. A block of 128 binary zeros is always encrypted using the AES algorithm as shown in Figure 7-83, using the decrypted key (K). In this case, the H field of the parameter block and the HS flag are not altered.

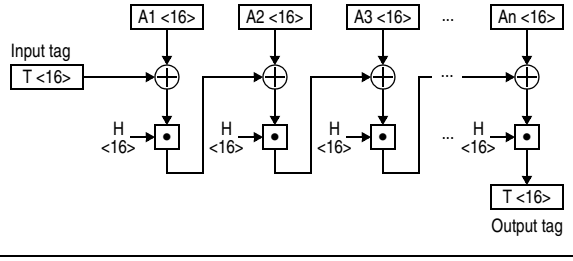
Additional-Authenticated-Data (AAD) Hashing: AAD hashing is not performed when either of the following conditions apply. In either of these conditions,

general registers R_3 and R_3+1 are not modified, and it is model dependent whether the T field in the parameter block is fetched and stored back unmodified.

- The third-operand length in general register $R_3 + 1$ is initially less than 16, and the last-AAD flag (LAAD, bit 54 of general register 0) is zero. In this case, the instruction completes by setting condition code 2.
- The third-operand length is initially zero, and the LAAD flag is one. In this case, processing continues as described in "Ciphering and Hashing" on page 7-86.

When the third-operand length in general register $R_3 + 1$ is nonzero, additional-authenticated-data hashing is performed. In this case, general register R_3 contains the address of a storage location containing data from which a message-authentication tag is computed using the GHASH algorithm described in Reference [17.] on page xxx. In addition to the blocks of the third operand, the GHASH function uses the hash subkey (described above) and the tag field in the parameter block as input values.

The result is obtained as if processing starts at the left end of the third operand and proceeds to the right, block by block. When one or more full 16-byte blocks of AAD remain, the processing is illustrated in Figure 7-84.



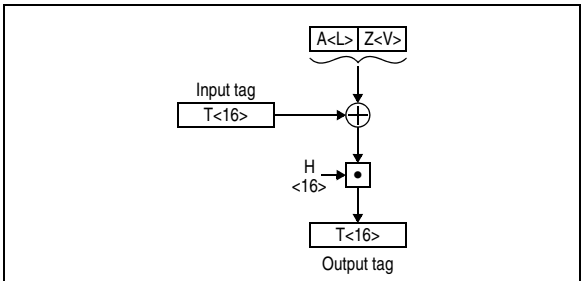
Symbol	Explanation
<n>	Length of item in bytes
An	Block n of additional authenticated data (third operand).
H	Hash subkey
T	Message-authentication tag (in the parameter block).

Figure 7-84. GHASH Processing of Full Blocks.

The AAD-hashing process ends when any of the following is true:

- A CPU-determined number of blocks that is less than the length of the third operand has been processed. In this case, the message-authentication tag computed thus far is placed into the T field of the parameter block, general register R_3 is incremented by the number of third-operand bytes processed, general register $R_3 + 1$ is decremented by the same amount, and the instruction completes by setting condition code 3.
- The last-AAD flag (LAAD, bit 54 of general register 0) is zero, and number of bytes remaining in the third operand is less than 16. In this case, the message-authentication tag computed thus far is placed into the T field of the parameter block, general register R_3 is incremented by the number of third-operand bytes processed (if any), general register $R_3 + 1$ is decremented by the same amount, and the instruction completes by setting condition code 2.
- The LAAD flag is one, and the number of bytes remaining in the third operand is between 1 and 15. In this case, the following is performed:
  - A copy of the remaining short block is padded on the right with sufficient binary zeros to form a full block that is hashed using GHASH.
  - General register R_3 is incremented by the number of third-operand bytes processed, and the third-operand length in general register $R_3 + 1$ is set to zero.

  Processing of the last block of the third operand is illustrated in Figure 7-85.



Symbol	Explanation
<n>	Length of item in bytes
A	Last (short) block of additional authenticated data (in the third operand).
H	Hash subkey
L	Length of last 3rd-operand block in bytes (between 1 and 15).
T	Message-authentication tag (in the parameter block).
V	Number of bytes of zeros to pad the last block: 16 – L.
Z	Zeros.

Figure 7-85. GHASH Processing of Last (Short) Block

- The LAAD flag is one and the number of bytes remaining in the third operand is zero.

For either of the above two cases (when the LAAD flag is one), it is model dependent whether the instruction completes with condition code 3, or processing continues with the ciphering and hashing of the second operand, as described below.

Ciphering and Hashing: The ciphering-and-hashing process is not performed when either of the following conditions apply. In either of these conditions, general registers R_1, R_2, and R_2+1 are not modified, and it is model dependent whether the CV and T fields in the parameter block are fetched and stored back unmodified.

- The second-operand length in general register $R_2 + 1$ is initially less than 16, and the last-plaintext/ciphertext flag (LPC, bit 55 of general register 0) is zero. In this case, the instruction completes by setting condition code 2.
- The second-operand length is initially zero, and the LPC flag is one. In this case, processing continues as described in "Last Message-Authentication-Tag Hashing and Encryption" on page 7-88.

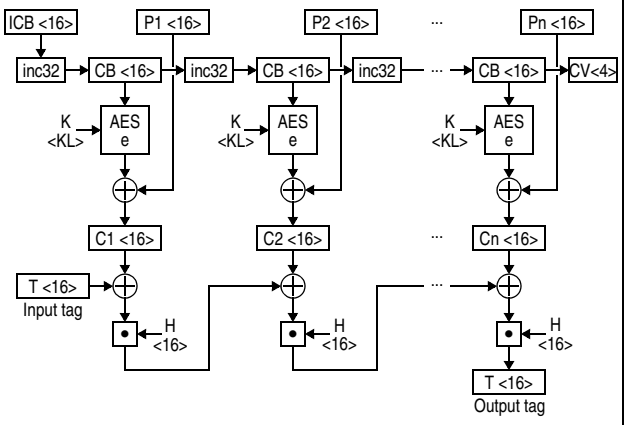
Depending on the M bit (bit 56 of general register 0), each block of the second operand is either encrypted or decrypted using the GCTR function described in Reference [17.] on page xxx. The respective encrypted or decrypted result is placed at the first-operand location, and the encrypted operand is hashed using the GHASH function. The combination of the GCTR and GHASH processing is described as the GCM function in Reference [17.] on page xxx.

Conceptually, the result is obtained as if processing starts at the left end of the first and second operands and proceeds to the right, block by block, as illustrated in Figure 7-86 and Figure 7-87. However, depending on the model, a unit of operation may process multiple blocks of the first and second operands in parallel; thus, the blocks may not necessarily be accessed in left-to-right order. Furthermore, multiple accesses may be made to a block, and in the case of the encryption operation, a first-operand block may be re-fetched after it is stored.

The GCTR function uses a 16-byte initial-counter block (ICB) formed from the concatenation of the leftmost 12 bytes of the initial-counter-value (J_0) on the left with the four-byte counter value (CV) on the right. GCTR also uses the key field (either directly from the parameter block for the KMA-GCM-AES functions, or the decrypted key for the KMA-GCM-encrypted-AES functions). For each block that is ciphered by GCTR, the counter value (CV) is incremented by one; a carry out of bit position 0 of the counter value is ignored. The GCTR function then uses a 16-byte counter block (CB) formed from the leftmost 12 bytes of J_0 concatenated with the incremented counter value.

The GHASH function uses the encrypted data (that is, the encrypted first-operand result when M is 0 or the second operand when M is one), the tag (T) field from either the parameter block or from the preceding step, and the hash subkey.

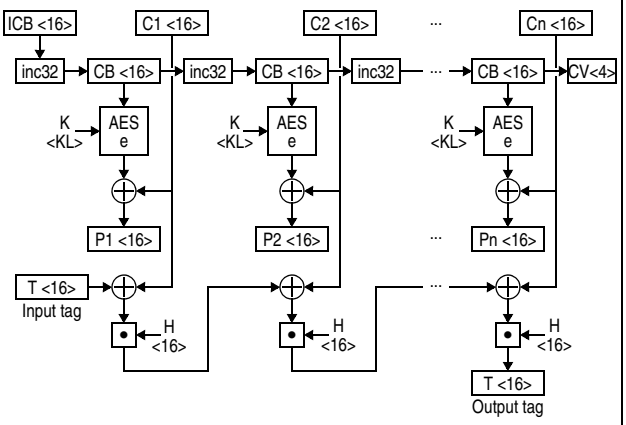
When the M bit is zero (that is, when the second operand is being encrypted), GCM processing of full blocks is illustrated in Figure 7-86.



Symbol	Explanation
<n>	Length of item in bytes
CB	Counter block formed from the concatenation of the leftmost 12 bytes of J_0 with the incremented counter value (CV)
Cn	Block n of the ciphertext (first operand).
CV	Counter value stored into the parameter block.
H	Hash subkey.
ICB	Initial counter block comprising the leftmost 12 bytes of the initial-counter value (J_0) concatenated with the 4-byte counter value (CV).
inc32	Incrementing of the 32-bit counter value (CV); a carry out of the leftmost bit position is ignored.
K	Key value.
KL	Key length (see Figure 7-79).
Pn	Block n of the plaintext (second operand).
T	Message-authentication tag (in the parameter block).

Figure 7-86. Combined Ciphering and Hashing of Full Blocks for Encryption

When the M bit is one (that is, when the second operand is being decrypted), GCM processing of full blocks is illustrated in Figure 7-87.



Explanation

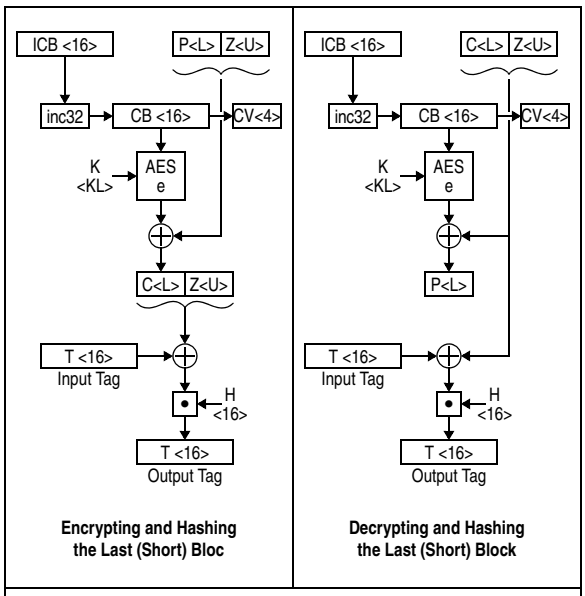
Cn Block n of the ciphertext (second operand).
Pn Block n of the plaintext (first operand).
Other symbols are the same as in Figure 7-86.

Figure 7-87. Combined Ciphering and Hashing of Full Blocks for Decryption

The ciphering-and-hashing process for either encryption or decryption continues until any of the following is true:

- A CPU-determined number of blocks that is less than the length of the second operand has been processed. In this case, the current counter value is placed into the CV field of the parameter block, the message-authentication tag computed thus far is placed into the T field of the parameter block, general registers R_1 and R_2 are incremented by the number of second-operand bytes processed, general register $R_2 + 1$ is decremented by the same amount, and the instruction completes by setting condition code 3.
- The last-plaintext/ciphertext flag (LPC, bit 55 of general register 0) is zero, and number of bytes remaining in the second operand is less than 16. In this case, the current counter value is placed into the CV field of the parameter block, the message-authentication tag computed thus far (if any) is placed into the T field of the parameter block, general registers R_1 and R_2 are incremented by the number of second-operand bytes processed (if any), general register $R_2 + 1$ is decremented by the same amount, and the instruction completes by setting condition code 2.
- The LPC flag is one, and the number of bytes remaining in the second operand is between 1 and 15. In this case, the following is performed:
  - A copy of the remaining bytes of the second operand is padded on the right with sufficient binary zeros to form a full block that is ciphered using the GCTR algorithm, and the leftmost bytes of the resulting encrypted or decrypted block are placed at the first-operand location. The number of bytes placed at the first-operand location is the same as the number of bytes remaining in the second operand (that is, less than 16).
  - The GHASH algorithm is then applied to the ciphertext. When M is zero, the input to the GHASH algorithm consists of a copy of the short block stored into the first-operand location, padded on the right with sufficient binary zeros to form a full block. When M is one, the input to the GHASH algorithm is the same as the input to the GCTR algorithm (that is, a copy of the remaining bytes of the second operand, padded on the right with sufficient binary zeros to form a full block).
  - The current counter value is placed into the CV field of the parameter block, the resulting tag value is placed in the parameter block, general registers R_1 and R_2 are incremented by the number of second-operand bytes processed, and general register $R_2 + 1$ is set to zero.

GCM processing of the last block of the second operand is illustrated in Figure 7-88.



Symbol	Explanation
<n>	Length of item in bytes
CB	Counter block formed from the concatenation of the leftmost 12 bytes of J_0 with the incremented counter value (CV)
C	Last (short) block of the ciphertext (first operand when M=0, second operand when M=1).
CV	Counter value (in the parameter block)
H	Hash subkey
ICB	Initial counter block comprising the leftmost 12 bytes of the initial-counter value (J_0) concatenated with the 4-byte counter value (CV).
inc32	Incrementing of the 32-bit counter value (CV); a carry out of the leftmost bit position is ignored.
K	Key value.
KL	Key length (see Figure 7-79).
L	Length of last second-operand block in bytes (between 1 and 15).
P	Last (short) block n of the plaintext (second operand when M=0, first operand when M=1).
T	Message-authentication tag (in the parameter block).
U	Number of bytes of zeros to pad the last block: 16 – L.
Z	Zeros.

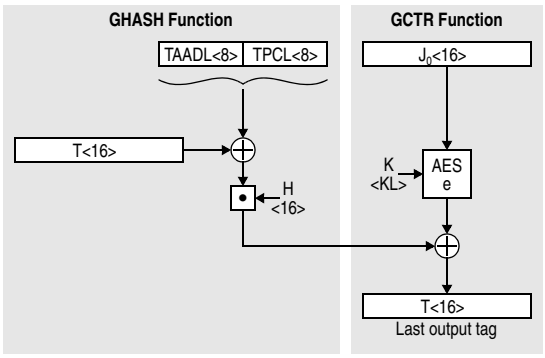
Figure 7-88. Ciphering and Hashing of the Last (Short) Block.

- The LPC flag is one, and the number of bytes remaining in the second operand is zero.

For either of the above two cases (when the LPC flag is one), it is model dependent whether the instruction completes with condition code 3 or continues processing with the last message-authentication-tag hashing and encryption, as described below.

Last Message-Authentication-Tag Hashing and Encryption: A 128-bit value comprising the concatenation of the 64-bit total-AAD-length (TAADL) and total-plaintext-or-ciphertext-length (TPCL) fields from the parameter block is hashed using the GHASH function. The GHASH function uses the concatenated lengths field, the tag (T) field as computed in the ciphering-and-hashing operation, and the hash subkey.

The resulting 128-bit output of GHASH is then processed by the GCTR algorithm. Note, unlike the ciphering-and-hashing operation, the input counter to GCTR is the initial-counter-block (J_0) field from the parameter block. The resulting 128-bit value replaces the tag (T) field in the parameter block, and the instruction completes with condition code 0. Figure 7-89 illustrates the hashing and encryption of the last tag value.



Symbol	Explanation
<n>	Length of item in bytes
H	Hash subkey
J_0	Initial counter value (from the parameter block)
K	Key value.
KL	Key length (see Figure 7-79).
T	Message-authentication tag (in the parameter block).
TAADL	Total length of the message's additional-authenticated data (AAD) in bits (from the parameter block).
TPCL	Total length of the message's plaintext or ciphertext in bits (from the parameter block).

Figure 7-89. Hashing and Encrypting the Last Tag

Common Operation: The detection of conditions resulting in condition code 3 depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

The results are unpredictable if either of the following is true:

- The third operand overlaps destructively with any portion of the parameter block that may be updated during AAD processing.
- The LAAD flag is one, and any of the following is true:
  - The second operand overlaps destructively with any portion of the parameter block that may be updated during ciphering and hashing.
  - The first operand overlaps destructively with any portion of the parameter block that may be accessed during ciphering and hashing.
  - The first operand overlaps destructively with the second operand, but the operands do not designate the same location.

Operands are said to overlap destructively when a location would be used as a source after data would have been moved into it, assuming processing to be performed from left to right.

Figure 7-90 illustrates the various condition codes set by the instruction, and the resulting second- and third-operand lengths based on the LAAD and LPC flags.

Condition Code	LAAD	LPC	3rd-operand length	2nd-operand length
0	1	1	0	0
1	—	—	Unchanged†	Unchanged†
2	0	—	< 16	Unchanged
	1	0	0	< 16
3	0	—	>= 16	Unchanged
	1	0	0	>= 16
	1 ‡	1 ‡	0 ‡	0 ‡

Explanation:

—	Not applicable
†	Also, first-, second-, and third-operand addresses in general registers R_1, R_2, and R_3 are unchanged
‡	It is model dependent whether CC3 is set for these conditions (which are identical to the CC0 conditions).
LAAD	Last-additional-authenticated-data flag, bit 54 of general register 0
LPC	Last-plaintext/ciphertext flag, bit 55 of general register 0

Figure 7-90. Condition Codes and Resulting Operand Lengths, based on LAAD and LPC

Store-type access exceptions may be recognized for any location in the parameter block, even though only the CV, T, and H fields are actually stored by the instruction.

A PER storage-alteration event may be recognized both for the first-operand location and for the portion of the parameter block that is stored. A PER zero-address-detection event may be recognized for the first-, second-, and third-operand locations and for the parameter block (including the reserved field of the parameter block). When PER events are detected for one or more of these locations, it is unpredictable which location is identified in the PER access identification (PAID) and PER ASCE ID (AI).

It is unpredictable how many bytes of the first-, second-, or third-operand locations has been processed when a PER storage-alteration event is recognized for the parameter block. When a storage-alteration PER event is recognized for the first-operand location, fewer than 4K additional bytes are stored into the first-operand locations before the event is reported.

When the third-operand length is initially zero, the third operand is not accessed, and the third-operand address and third-operand length in general registers R_3 and $R_3 + 1$, respectively, are not changed. When the second-operand length is initially zero, the second operand is not accessed, and the second-operand address and second-operand length in general registers R_2 and $R_2 + 1$, respectively, are not changed. However, the parameter block may be accessed even when the second- and third-operand lengths are both zero.

When the contents of the R_1 and R_2, fields are the same, the contents of the designated registers are incremented only by the number of bytes processed, not by a multiple of the number of bytes processed.

As observed by this CPU, other CPUs, and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

In certain unusual situations, instruction execution may complete by setting condition code 3 without updating the registers to reflect the last unit of the first, second, and third operands processed. The size of the unit processed in this case depends on the sit-

uation and the model, but is limited such that the portion of the first and second operands which have been processed and not reported do not overlap in storage; and the portion of the first and third operands which have been processed and not reported do not overlap in storage. In all cases, change bits are set and PER storage-alteration events are reported, when applicable, for all first-operand locations processed.

For functions that perform a comparison of the encrypted cryptographic key with the wrapping-key verification-pattern register, it is unpredictable whether access exceptions and PER zero-address-detection events are recognized for the first, second, and third operands when the comparison results in a mismatch and respective operand's length is nonzero.

Access exceptions may be reported for a larger portion of an operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of an operand nor for locations more than 4K bytes beyond the current location being processed.

Special Conditions for KMA

A specification exception is recognized and no other action is taken if any of the following occurs:

1. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
2. The R_1 $R_{2,}$ or R_3 field designates an odd-numbered register or general register 0.
3. The R_3 field designates the same register as either the R_1 or R_2 fields.
4. The function code is nonzero, and the LPC flag is one (indicating that the last blocks of plaintext or ciphertext are being processed), but the LAAD flag is zero (indicating that not all AAD has been processed).

Resulting Condition Code:

0 Normal completion
1 Verification-pattern mismatch
2 Incomplete processing (remaining third-operand length is less than 16 when the LAAD flag is zero, or remaining second-operand length is less than 16 when the LPC flag is zero)
3 Partial completion (model-dependent limit exceeded)

Program Exceptions:

- Access (fetch, operand 2, operand 3, parameter block fields; store, operand 1, counter value, hash subkey, tag)
- Operation (if the message-security-assist extension 8 is not installed)
- Specification
- Transaction constraint

1.-7.	Exceptions with the same priority as the priority of program-interruption conditions for the general case.
8.	Specification exception due to invalid function code or invalid register number.
9.	Specification exception due to LPC flag being one when the LAAD flag is zero (applicable only when the function code is nonzero).
10.A.1	Access exceptions for an access to the parameter block.
10.A.2.	Condition code 1 due to verification-pattern mismatch.
10.B	Access exceptions for an access to the first, second, or third operand.
11.	Condition code 3 due to partial processing of the third operand.
12.	Condition code 2 due to the remaining third-operand length being less than 16 when the LAAD flag is zero.
13.	Condition code 3 due to partial processing of the second operand.
14.	Condition code 2 due to remaining second-operand length being less than 16 when the LPC flag is zero.
15.	Condition code 0 due to normal completion.

Figure 7-91. Priority of Execution: KMA

Programming Notes:

1. The performance of the query function (code 0) may be significantly slower than that of simply examining a location in storage. If the program needs to frequently test for the availability of a

function, it should perform the query function once during initialization; subsequently it should examine the stored results of the query function in memory with an instruction such as TEST UNDER MASK.

1. As observed by this CPU, other CPUs, and the I/O subsystem, inconsistent results may be briefly stored in the first-operand location. See the section "Effects of CPU Retry" on page 11-3 for further details.

2. When processing an individual message, the program should initially set the following fields in the parameter block and not alter these fields until the instruction completes with condition code 0; otherwise, the results will not conform to the GCM standard, as described in Reference [17.] on page xxx.

   a. *Counter Value (CV):* The CV field should be initialized with the rightmost 4 bytes of the initial-counter-value (J_0) field.

   b. *Tag (T):* The tag field should be initialized to zeros.

   c. *The Hash Subkey (H):* For KMA-GCM-AES functions (codes 18-20), the following applies:

      - When the HS flag is one, the program supplies a pre-computed hash subkey in the H field. The subkey comprises 16 bytes of zeros, encrypted using the AES algorithm.
      - When the HS flag is zero, the CPU encrypts 16 bytes of zeros using the AES algorithm and key field in the parameter block, stores the results in the H field, and sets the HS flag to one.

      For KM-GCM-encrypted-AES functions (codes 26-28), the CPU always calculates the hash subkey, and the H field and HS flags are not altered.

   d. *Initial-Counter Value (J_0):* The initial-counter value is derived from an initialization vector (IV) provided by the program. If the program uses a 96-bit IV (as recommended in Reference [17.] on page xxx), then it should store the IV into the leftmost 12 bytes of the J_0 field, and store 00000001 hex in the rightmost bytes of the J_0 field. If the program uses an IV having a different length, then it must supply a 16-byte hashed value of the IV in the J_0 field using the GHASH algorithm (as described in Reference [17]).

   e. *Key Value (K) and Wrapping-Key Verification Pattern (WK_aVP):* For proper ciphering of an individual message, the key must be the same for all executions of the instruction. For KM-encrypted-AES functions, the wrapping-key-verification-pattern field must also remain unchanged.

3. When processing the last plaintext or ciphertext block(s) (that is, when the LPC flag is one), the total-AAD-length (TAADL) and total-plaintext/ciphertext-length (TPCL) fields in the parameter block should contain the total length in bits of the respective AAD and plaintext or ciphertext for the entire message.

4. The program is responsible for comparing the hashed tag of a decrypted message with that of the message when it was encrypted to ensure authenticity of the message.

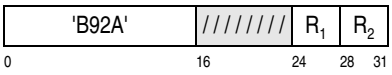
5. The LAAD and LPC flags provide the means by which a message can be ciphered and hashed when not all components of the message are available for a single execution of the instruction. For example, if the plaintext or ciphertext portion of a message spans multiple disk or tape records, and not all blocks of the message have been read into storage, the program can process the earlier blocks of the message by issuing the instruction with the LPC flag set to zero. When the final plaintext or ciphertext blocks of the message are available, the program can then complete the message ciphering by issuing the instruction with the LPC flag set to one.

   When condition code 2 is set due to either or both the LAAD or LPC flag being zero, the general registers containing the operand addresses and lengths, and the parameter block are updated to indicate the progress thus far. However, unlike condition code 3 (where the program can simply branch back to the instruction to continue the operation), the program is responsible for updating the operand addresses and lengths, and the LAAD and LPC flags, as necessary, before branching back to the instruction. If the program simply branches back to the instruction in response to condition code 2, a nonproductive program loop will result.

6. When condition code 3 is set, the general registers containing the operand addresses and lengths, and the parameter block, are usually updated such that the program can simply branch back to the instruction to continue the operation.

CIPHER MESSAGE WITH CIPHER FEEDBACK

KMF R_1,R_2 [RRE]



A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction are ignored.

Bit positions 32-39 of general register 0 contain the length of cipher feedback in bytes (LCFB, also known as “s”), and bit positions 57-63 of general register 0 contain the function code. Figure 7-92 shows the assigned function codes for CIPHER MESSAGE WITH CIPHER FEEDBACK. All other function codes are unassigned. For cipher functions, bit 56 is the modifier bit which specifies whether an encryption or a decryption operation is to be performed. The modifier bit is ignored for all other functions. Bits 0-31 and 40-55 of general register 0 are ignored.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register 1 constitute the address.

The function codes for CIPHER MESSAGE WITH CIPHER FEEDBACK are as follows.

Code	Function	Parm. Block Size (bytes)	Data Block Size[1] (bytes)
0	KMF-Query	16	—
1	KMF-DEA	16	8
2	KMF-TDEA-128	24	8
3	KMF-TDEA-192	32	8
9	KMF-Encrypted-DEA	40	8
10	KMF-Encrypted-TDEA-128	48	8
11	KMF-Encrypted-TDEA-192	56	8
18	KMF-AES-128	32	16
19	KMF-AES-192	40	16
20	KMF-AES-256	48	16
26	KMF-Encrypted-AES-128	64	16
27	KMF-Encrypted-AES-192	72	16
28	KMF-Encrypted-AES-256	80	16

Explanation:

1 The size of ciphertext and plaintext segments (s) range from 1 to the data-block size, depending on the value of the LCFB in bits 32-39 of general register 0.

— Not applicable

Figure 7-92. Function Codes for CIPHER MESSAGE WITH CIPHER FEEDBACK

All other function codes are unassigned.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R_1, R_2, and $R_2 + 1$ are ignored for the query function.

For all other functions, the second operand is ciphered as specified by the function code using a cryptographic key and an initial chaining value in the parameter block, and the result is placed in the first-operand location. The chaining value is updated as part of the operation.

The R_1 field designates a general register and must designate an even-numbered register other than general register 0; otherwise, a specification exception is recognized. The R_2 field designates an even-odd pair of general registers and must designate an

even-numbered register other than general register 0; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first and second operands is specified by the contents of the R_1 and R_2 general registers, respectively. The number of bytes in the second-operand location is specified in general register $R_2 + 1$ and must be a multiple of the cipher-feedback length in bits 32-39 of general register 0; otherwise, a specification exception is recognized. The first operand is the same length as the second operand.

As part of the operation, the addresses in general registers R_1 and R_2 are incremented by the number of bytes processed, and the length in general register $R_2 + 1$ is decremented by the same number. The formation and updating of the addresses and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the addresses of the first and second operands, respectively, and the contents of bit positions 0-39 are ignored; bits 40-63 of the updated addresses replace the corresponding bits in general registers R_1 and R_2, carries out of bit position 40 of the updated address are ignored, and the contents of bit positions 32-39 of general registers R_1 and R_2 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general registers R_1 and R_2 constitute the addresses of the first and second operands, respectively, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated addresses replace the corresponding bits in general registers R_1 and R_2, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general registers R_1 and R_2 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general registers R_1 and R_2 constitute the addresses of the first and second operands, respectively; bits 0-63 of the updated addresses replace the contents of general registers R_1 and R_2, and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_2 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the first and second operands, and the contents of bit positions 0-31 are ignored; bits 32-63 of the updated value replace the corresponding bits in general register $R_2 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the first and second operands; and the updated value replaces the contents of general register $R_2 + 1$.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, R_2, and $R_2 + 1$, always remain unchanged.

Figure 7-93 on page 7-93 shows the contents of the general registers just described.

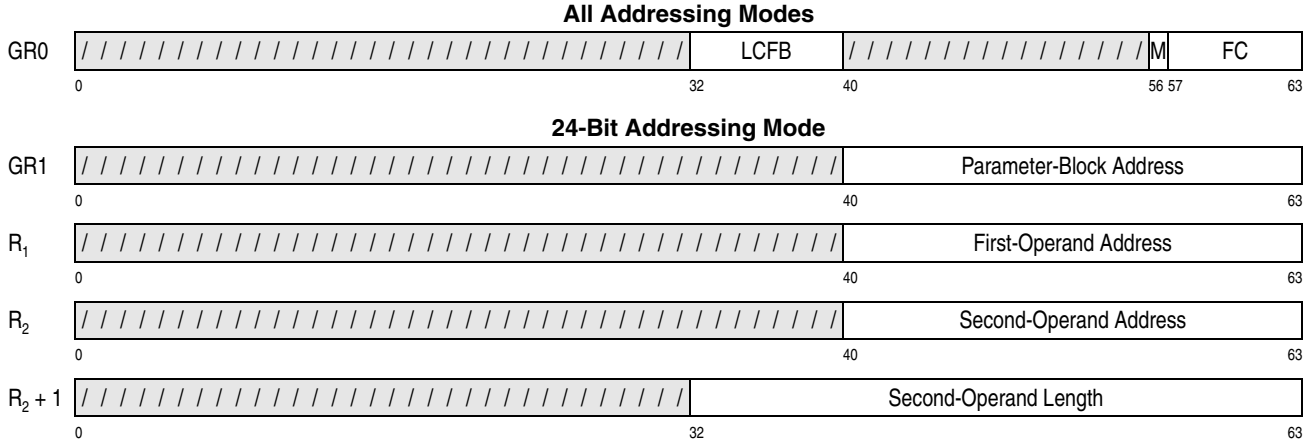


Figure 7-93. General Register Assignment for KMF (Part 1 of 2)

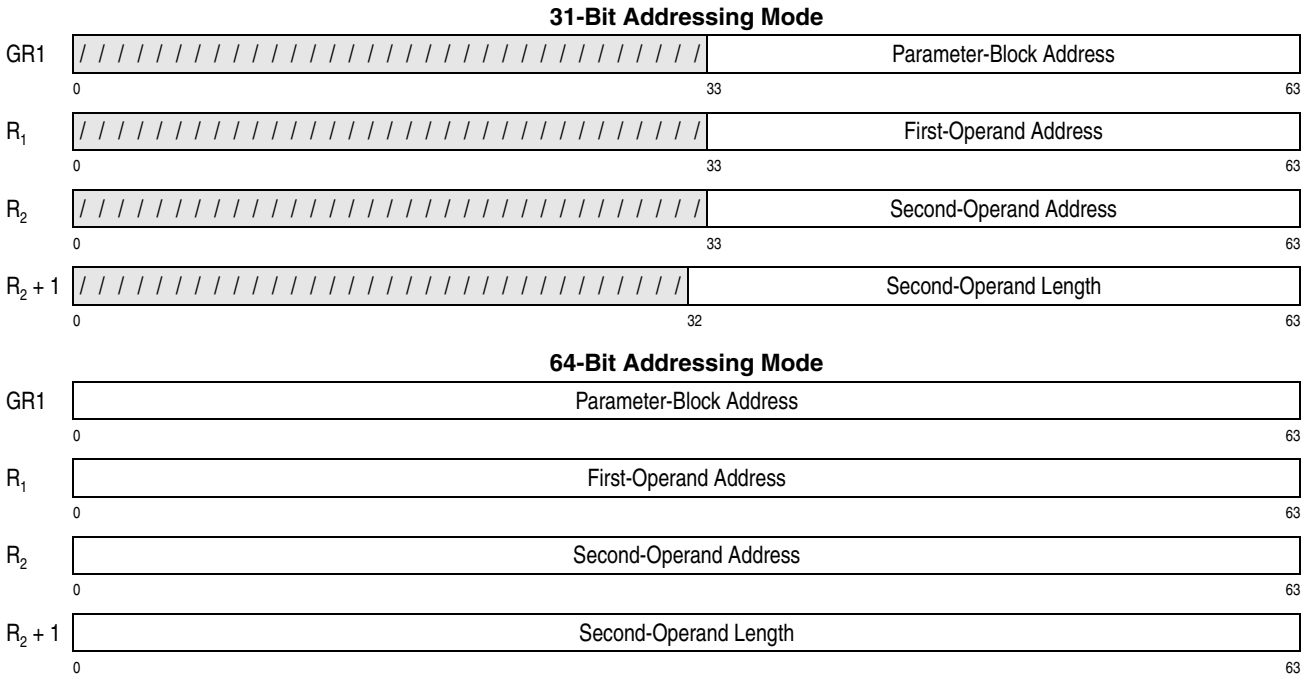


Figure 7-93. General Register Assignment for KMF (Part 2 of 2)

In the access-register mode, access registers 1, R_1, and R_2 specify the address spaces containing the parameter block, first, and second operands, respectively.

The result is obtained as if processing starts at the left end of both the first and second operands and proceeds to the right, block by block. The operation is ended when the number of bytes in the second operand as specified in general register $R_2 + 1$ have been processed and placed at the first-operand location (called normal completion) or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

The results in the first-operand location and the chaining-value field are unpredictable if any of the following situations occur:

1. The cryptographic-key field or the encrypted cryptographic-key field overlaps any portion of the first operand.
2. The chaining-value field overlaps any portion of the first operand or the second operand.
3. The first and second operands overlap destructively. Operands are said to overlap destructively when the first-operand location would be used as a source after data would have been moved into it, assuming processing to be performed from left to right and one byte at a time.

When the operation ends due to normal completion, condition code 0 is set and the resulting value in $R_2 + 1$ is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in $R_2 + 1$ is nonzero.

A PER storage-alteration event may be recognized both for the first-operand location and for the portion of the parameter block that is stored. A PER zero-address-detection event may be recognized for the first- and second-operand locations and for the parameter block. When PER events are detected for one or more of these locations, it is unpredictable which location is identified in the PER access identification (PAID) and PER ASCE ID (AI).

When a storage-alteration PER event is recognized, fewer than 4K additional bytes are stored into the first-operand locations before the event is reported.

When the second-operand length is initially zero, the parameter block, first, and second operands are not accessed, general registers R_1, R_2, and $R_2 + 1$ are not changed, and condition code 0 is set.

When the contents of the R_1 and R_2 fields are the same, the contents of the designated registers are incremented only by the number of bytes processed, not by twice the number of bytes processed.

As observed by this CPU, other CPUs, and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

In certain unusual situations, instruction execution may complete by setting condition code 3 without updating the registers and chaining value to reflect the last unit of the first and second operands processed. The size of the unit processed in this case depends on the situation and the model, but is limited such that the portion of the first and second operands which have been processed and not reported do not overlap in storage. In all cases, change bits are set and PER storage-alteration events are reported, when applicable, for all first-operand locations processed.

For functions that perform a comparison of the encrypted cryptographic key with the wrapping-key verification-pattern register, it is unpredictable whether access exceptions and PER zero-address-detection events are recognized for the first and second operands when the comparison results in a mismatch.

Access exceptions may be reported for a larger portion of an operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of an operand nor for locations more than 4K bytes beyond the current location being processed.

Symbols Used in Function Descriptions

The following symbols are used in the subsequent description of the CIPHER MESSAGE WITH CIPHER FEEDBACK functions. For data-encryption-algorithm (DEA) functions, the DEA-key-parity bit in each byte of the DEA key is ignored, and the operation proceeds normally, regardless of the DEA-key parity of the key.

Further description of the data-encryption algorithm may be found in Reference [13.] on page xxx. Further description of the AES standard may be found in Reference [14.] on page xxx. Further description of the cipher-feedback mode of encryption and decryption may be found in Reference [16.] on page xxx. Further description of the cipher-feedback mode of encryption and decryption may be found in Reference [16.] on page xxx.

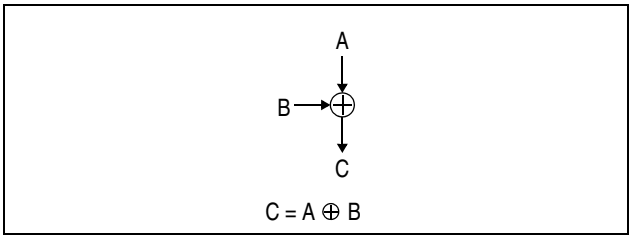
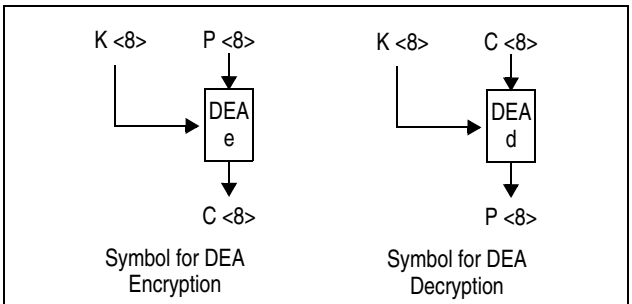
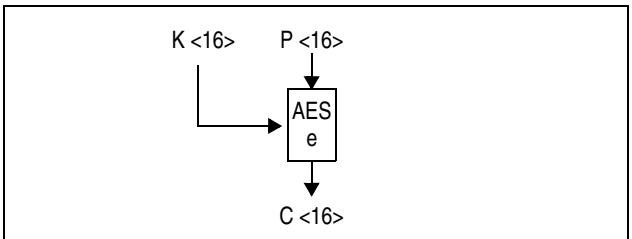


Figure 7-94. Symbol For Bit-Wise Exclusive OR



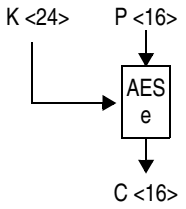
Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-95. Symbols for DEA Encryption and Decryption



Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

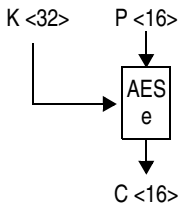
Figure 7-96. Symbols for AES-128 Encryption



Symbol for AES-192 Encryption

Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-97. Symbols for AES-192 Encryption



Symbol for AES-256 Encryption

Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-98. Symbols for AES-256 Encryption

KMF-Query (Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the function has the following format:

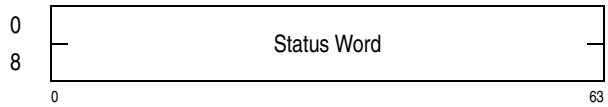


Figure 7-99. Parameter Block for KMF-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the CIPHER MESSAGE WITH CIPHER FEEDBACK instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KMF-Query function completes; condition codes 1 and 3 are not applicable to this function.

In the following function descriptions, the term *input block* refers to the 8- or 16-byte input to the DEA or AES algorithms, respectively. Similarly, the term *output block* refers to the 8- or 16-byte results from the respective algorithms. The term *segment* refers to the s-byte portions of the first and second operands.

KMF-DEA (Function Code 1)

KMF-Encrypted-DEA (Function Code 9)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KMF-DEA function has the following format:

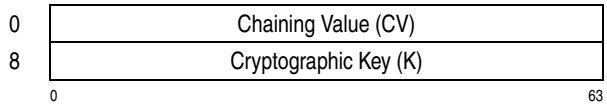


Figure 7-100. Parameter Block for KMF-DEA

For the KMF-DEA function, the chaining value is in byte offset 0-7 of the parameter block and the cryptographic key is in byte offsets 8-15 of the parameter block.

The parameter block used for the KMF-Encrypted-DEA function has the following format:

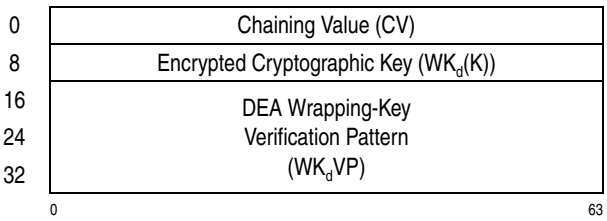


Figure 7-101. Parameter Block for KMF-Encrypted-DEA

For the KMF-Encrypted-DEA function, the contents of byte offsets 16-39 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by set-

ting condition code 1. If they match, byte offsets 0-7 of the parameter block contain the chaining value, and the contents of byte offsets 8-15 of the parameter block are deciphered using the DEA wrapping key to obtain the 64-bit cryptographic key. (See the section, “Protection of Cryptographic Keys” on page 7-424 for details.)

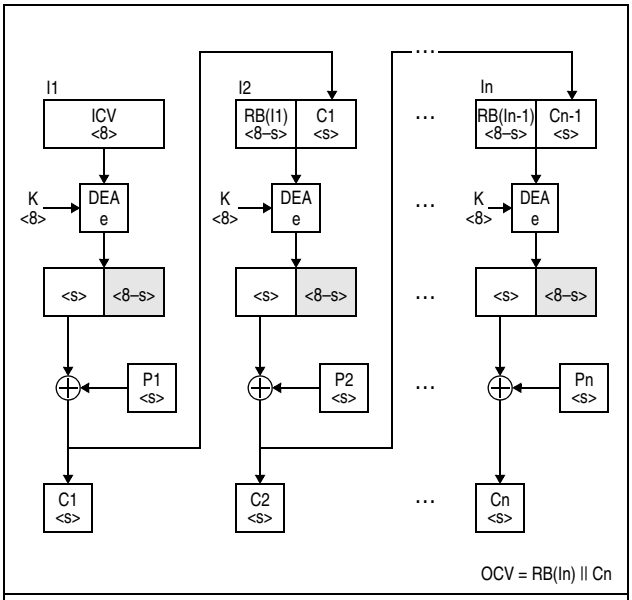
The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The s-byte plaintext segments (P1, P2, …, Pn) in operand 2 are enciphered using the DEA-encryption algorithm with the 64-bit cryptographic key and the 64-bit chaining value, where s is the length of cipher feedback in bytes (LCFB).

The first input block to the DEA-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. The s leftmost bytes of each output block are exclusive-ORed with the corresponding s-byte plaintext segment to form a s-byte ciphertext segment. The remaining (8 - s) bytes of each output block are ignored. Each ciphertext segment is concatenated to the right of the (8 - s) rightmost bytes of the previous input block to form the next input block.

The process is repeated with the successive input blocks until a ciphertext segment is produced for every plaintext segment, or until a CPU-determined number of ciphertext segments have been produced.

The ciphertext segments (C1, C2, …, Cn) are stored in operand 1. The final next-input block is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-50.



Symbol	Explanation
||	Concatenation
Ij	Input block j to DEA
RB(Ij)	Rightmost (8-s) bytes of input block j
Note:	The rightmost (8–s) bytes of the output block from the DEA-encipher operation are ignored.

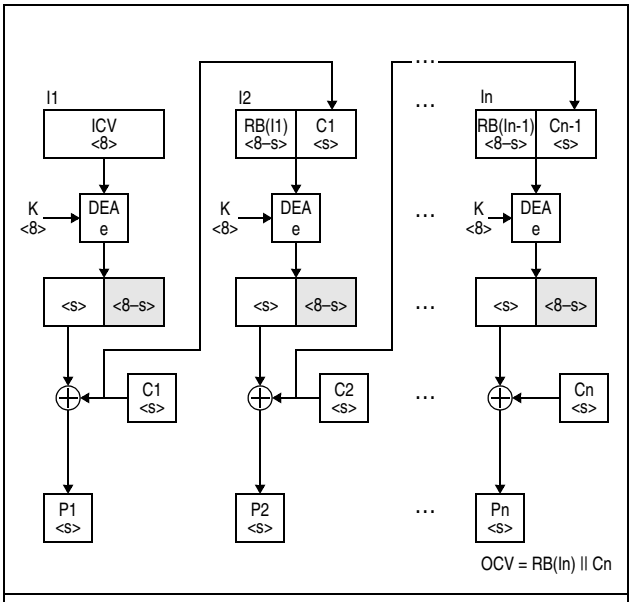
Figure 7-102. KMF-DEA Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The s-byte ciphertext segments (C1, C2, …, Cn) in operand 2 are deciphered using the DEA-encryption algorithm with the 64-bit cryptographic key and the 64-bit chaining value, where s is the length of cipher feedback in bytes.

The first input block to the DEA-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. The s leftmost bytes of each output block are exclusive-ORed with the corresponding s-byte ciphertext segment to form a s-byte plaintext segment. The remaining (8 - s) bytes of each output block are ignored. Each ciphertext segment is concatenated to the right of the (8 - s) rightmost bytes of the previous input block to form the next input block.

The process is repeated with the successive input blocks until a plaintext segment is produced for every ciphertext segment, or until a CPU-determined number of plaintext segments have been produced.

The plaintext segments (P1, P2, …, Pn) are stored in operand 1. The final next-input block (OCV) is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-103.



Symbol	Explanation
||	Concatenation
Ij	Input block j to DEA
RB(Ij)	Rightmost (8-s) bytes of input block j
Note:	The rightmost (8–s) bytes of the output block from the DEA-decipher operation are ignored.

Figure 7-103. KMF-DEA Decipher Operation

KMF-TDEA-128 (Function Code 2)

KMF-Encrypted-TDEA-128 (Function Code 10)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KMF-TDEA-128 function has the following format:

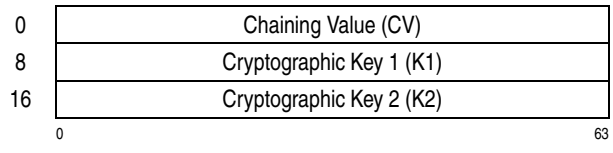


Figure 7-104. Parameter Block for KMF-TDEA-128

For the KMF-TDEA-128 function, the chaining value is in byte offset 0-7 of the parameter block and the cryptographic key is in byte offsets 8-23 of the parameter block.

The parameter block used for the KMF-Encrypted-TDEA-128 function has the following format:

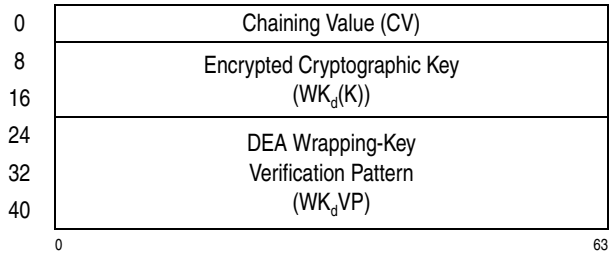


Figure 7-105. Parameter Block for KMF-Encrypted-TDEA-128

For the KMF-Encrypted-TDEA-128 function, the contents of byte offsets 24-47 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-7 of the parameter block contain the chaining value, and the contents of byte offsets 8-23 of the parameter block are deciphered using the DEA wrapping key to obtain the 128-bit cryptographic key. (See the section, “Protection of Cryptographic Keys” on page 7-424 for details.)

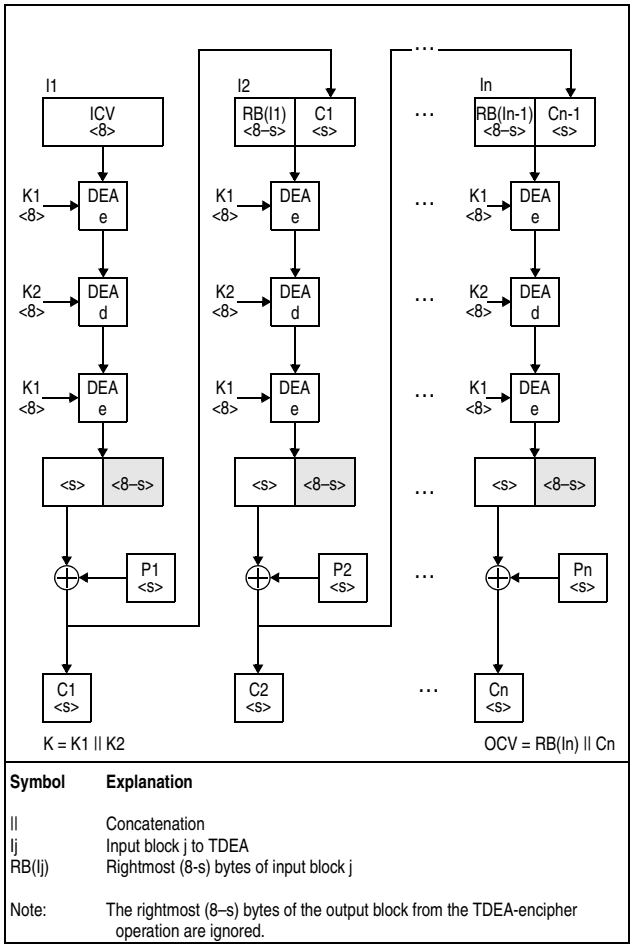
The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The s-byte plaintext segments (P1, P2, …, Pn) in operand 2 are enciphered using the TDEA-encryption algorithm with the 128-bit cryptographic key and the 64-bit chaining value, where s is the length of cipher feedback in bytes.

The first input block to the TDEA-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. The s leftmost bytes of each output block are exclusive-ORed with the corresponding s-byte plaintext segment to form a s-byte ciphertext segment. The remaining (8 - s) bytes of each output block are ignored. Each ciphertext segment is concatenated to the right of the (8 - s) rightmost bytes of the previous input block to form the next input block.

The process is repeated with the successive input blocks until a ciphertext segment is produced for every plaintext segment, or until a CPU-determined number of ciphertext segments have been produced.

The ciphertext segments (C1, C2, …, Cn) are stored in operand 1. The final next-input block is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-106.



Symbol	Explanation
||	Concatenation
Ij	Input block j to TDEA
RB(Ij)	Rightmost (8-s) bytes of input block j
Note:	The rightmost (8–s) bytes of the output block from the TDEA-encipher operation are ignored.

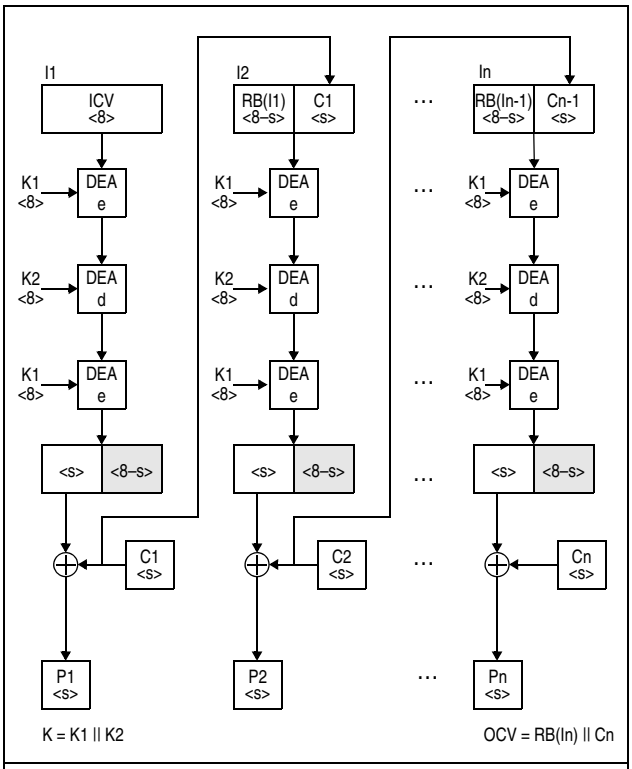
Figure 7-106. KMF-TDEA-128 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The s-byte ciphertext segments (C1, C2, …, Cn) in operand 2 are deciphered using the TDEA-encryption algorithm with the 128-bit cryptographic key and the 64-bit chaining value, where s is the length of cipher feedback in bytes.

The first input block to the TDEA-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. The s leftmost bytes of each output block are exclusive-ORed with the corresponding s-byte ciphertext segment to form a s-byte plaintext segment. The remaining (8 - s) bytes of each output block are ignored. Each ciphertext segment is concatenated to the right of the (8 - s) rightmost bytes of the previous input block to form the next input block.

The process is repeated with the successive input blocks until a plaintext segment is produced for every ciphertext segment, or until a CPU-determined number of plaintext segments have been produced.

The plaintext segments (P1, P2, …, Pn) are stored in operand 1. The final next-input block (OCV) is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-107.



Symbol	Explanation
||	Concatenation
Ij	Input block j to TDEA
RB(Ij)	Rightmost (8-s) bytes of input block j
Note:	The rightmost (8–s) bytes of the output block from the TDEA-decipher operation are ignored.

Figure 7-107. KMF-TDEA-128 Decipher Operation

KMF-TDEA-192 (Function Code 3)

KMF-Encrypted-TDEA-192 (Function Code 11)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KMF-TDEA-192 function has the following format:

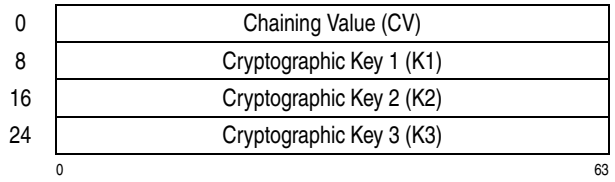


Figure 7-108. Parameter Block for KMF-TDEA-192

For the KMF-TDEA-192 function, the chaining value is in byte offset 0-7 of the parameter block and the cryptographic key is in byte offsets 8-31 of the parameter block.

The parameter block used for the KMF-Encrypted-TDEA-192 function has the following format:

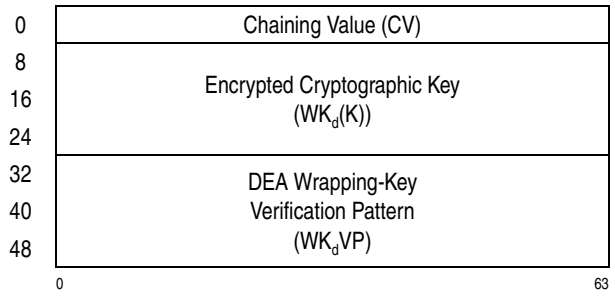


Figure 7-109. Parameter Block for KMF-Encrypted-TDEA-192

For the KMF-Encrypted-TDEA-192 function, the contents of byte offsets 32-55 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-7 of the parameter block contain the chaining value, and the contents of byte offsets 8-31 of the parameter block are deciphered using the DEA wrapping key to obtain the 192-bit cryptographic key. (See the section, “Protection of Cryptographic Keys” on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The s-byte plaintext segments (P1, P2, …, Pn) in operand 2 are enciphered using the TDEA-encryption algorithm with the 192-bit cryptographic key and the 64-bit chaining value, where s is the length of cipher feedback in bytes.

The first input block to the TDEA-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. The s leftmost bytes of each output block are exclusive-ORed with the corresponding s-byte plaintext segment to form a s-byte ciphertext segment. The remaining (8 - s) bytes of each output block are ignored. Each ciphertext segment is concatenated to the right of the (8 - s) rightmost bytes of the previous input block to form the next input block.

The process is repeated with the successive input blocks until a ciphertext segment is produced for every plaintext segment, or until a CPU-determined number of ciphertext segments have been produced.

The ciphertext segments (C1, C2, …, Cn) are stored in operand 1. The final next-input block (OCV) is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-110.

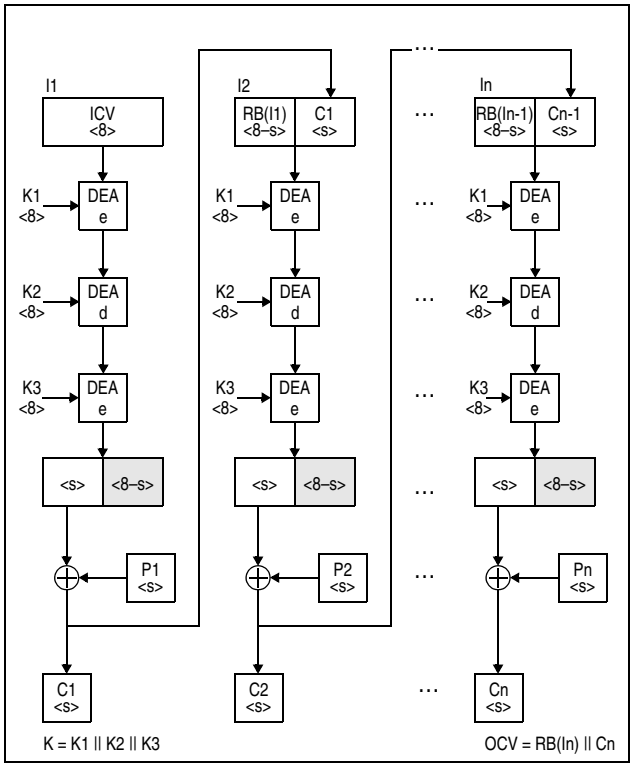


Figure 7-110. KMF-TDEA-192 Encipher Operation

Symbol	Explanation
||	Concatenation
Ij	Input block j to TDEA
RB(Ij)	Rightmost (8-s) bytes of input block j
Note:	The rightmost (8–s) bytes of the output block from the TDEA-encipher operation are ignored.

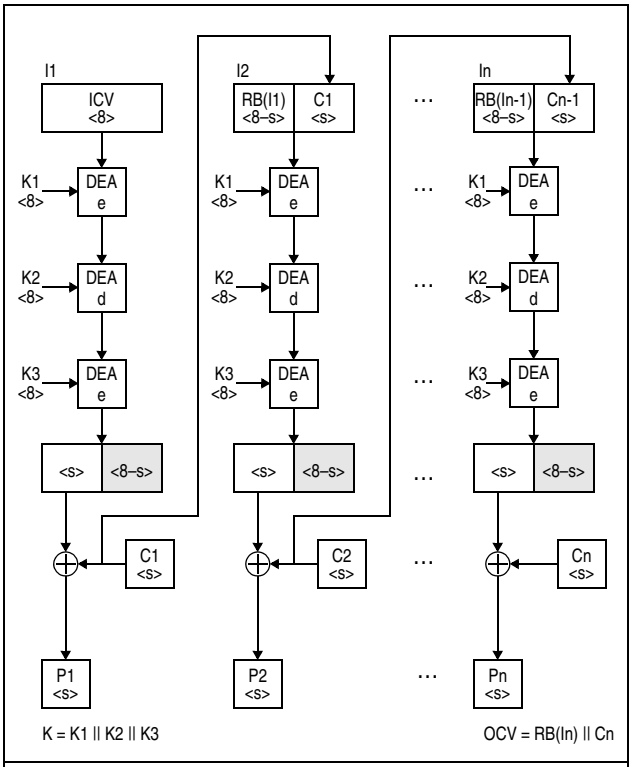
Figure 7-110. KMF-TDEA-192 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The s-byte ciphertext segments (C1, C2, …, Cn) in operand 2 are deciphered using the TDEA-encryption algorithm with the 192-bit cryptographic key and the 64-bit chaining value, where s is the length of cipher feedback in bytes.

The first input block to the TDEA-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. The s leftmost bytes of each output block are exclusive-ORed with the corresponding s-byte ciphertext segment to form a s-byte plaintext segment. The remaining (8 - s) bytes of each output block are ignored. Each ciphertext segment is concatenated to the right of the (8 - s) rightmost bytes of the previous input block to form the next input block.

The process is repeated with the successive input blocks until a plaintext segment is produced for every ciphertext segment, or until a CPU-determined number of plaintext segments have been produced.

The plaintext segments (P1, P2, …, Pn) are stored in operand 1. The final next-input block is stored into the chaining-value field of the parameter block.The operation is shown in Figure 7-111.



Symbol	Explanation
||	Concatenation
Ij	Input block j to TDEA
RB(Ij)	Rightmost (8-s) bytes of input block j
Note:	The rightmost (8–s) bytes of the output block from the TDEA-decipher operation are ignored.

Figure 7-111. KMF-TDEA-192 Decipher Operation

KMF-AES-128 (Function Code 18)

KMF-Encrypted-AES-128 (Function Code 26)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KMF-AES-128 function has the following format:

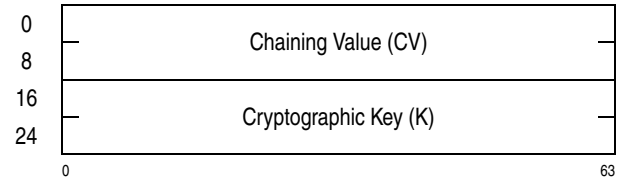


Figure 7-112. Parameter Block for KMF-AES-128

For the KMF-AES-128 function, the chaining value is in byte offset 0-15 of the parameter block and the cryptographic key is in byte offsets 16-31 of the parameter block.

The parameter block used for the KMF-Encrypted-AES-128 function has the following format:

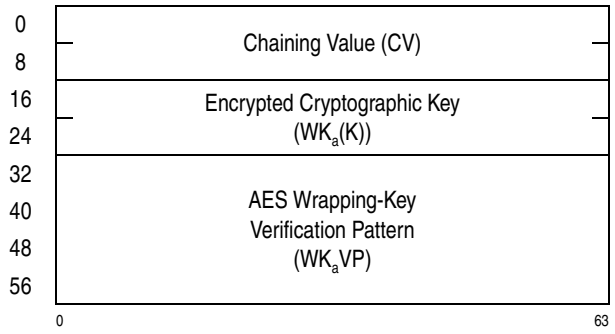


Figure 7-113. Parameter Block for KMF-Encrypted-AES-128

For the KMF-Encrypted-AES-128 function, the contents of byte offsets 32-63 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-15 of the parameter block contain the chaining value, and the contents of byte offsets 16-31 of the parameter block are deciphered using the AES wrapping key to obtain the 128-bit cryptographic key. (See the section "Protection of Cryptographic Keys" on page 7-424 for details.)

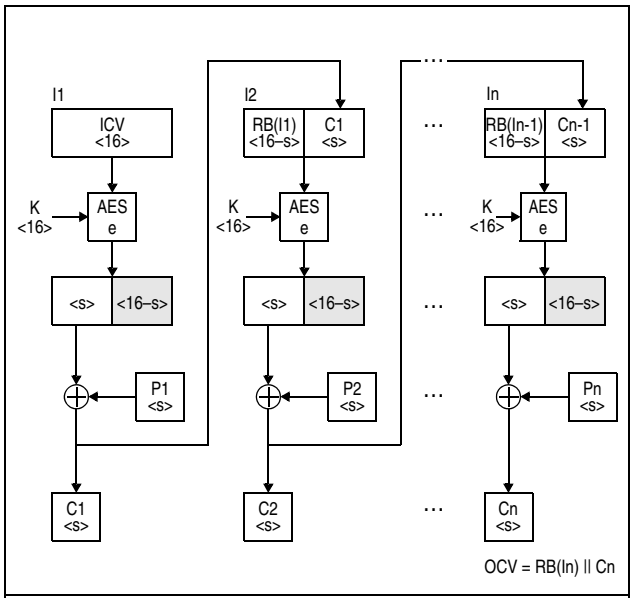
The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The s-byte plaintext segments (P1, P2, …, Pn) in operand 2 are enciphered using the AES-encryption algorithm with the 128-bit cryptographic key and the 128-bit chaining value, where s is the length of cipher feedback in bytes.

The first input block to the AES-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. The s leftmost bytes of each output block are exclusive-ORed with the corresponding s-byte plaintext segment to form a s-byte ciphertext segment. The remaining (16 - s) bytes of each output block are ignored. Each ciphertext segment is concatenated to the right of the (16 - s) rightmost bytes of the previous input block to form the next input block.

The process is repeated with the successive input blocks until a ciphertext segment is produced for every plaintext segment, or until a CPU-determined number of ciphertext segments have been produced.

The ciphertext segments (C1, C2, …, Cn) are stored in operand 1. The final next-input block is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-114.



Symbol	Explanation
||	Concatenation
Ij	Input block j to AES
RB(Ij)	Rightmost (16-s) bytes of input block j
Note:	The rightmost (16–s) bytes of the output block from the AES-encipher operation are ignored.

Figure 7-114. KMF-AES-128 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The s-byte ciphertext segments (C1, C2, …, Cn) in operand 2 are deciphered using the AES-encryption algorithm with the 128-bit cryptographic key and the 128-bit chaining value, where s is the length of cipher feedback in bytes.

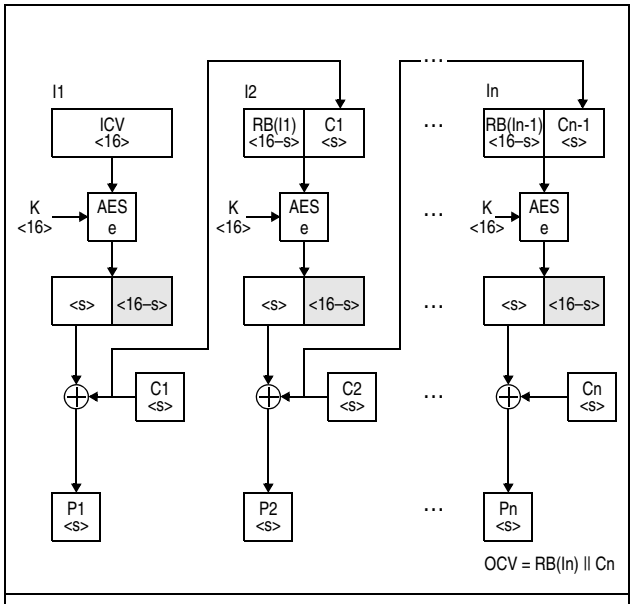
The first input block to the AES-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. The s leftmost bytes of each output block are exclusive-ORed with the corresponding s-

byte ciphertext segment to form a s-byte plaintext segment. The remaining (16 - s) bytes of each output block are ignored. Each ciphertext segment is concatenated to the right of the (16 - s) rightmost bytes

of the previous input block to form the next input block.

The process is repeated with the successive input blocks until a plaintext segment is produced for every ciphertext segment, or until a CPU-determined number of plaintext segments have been produced.

The plaintext segments (P1, P2, …, Pn) are stored in operand 1. The final next-input block (OCV) is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-115.



Symbol	Explanation
||	Concatenation
Ij	Input block j to AES
RB(Ij)	Rightmost (16-s) bytes of input block j
Note:	The rightmost (16–s) bytes of the output block from the AES-encipher operation are ignored.

Figure 7-115. KMF-AES-128 Decipher Operation

KMF-AES-192 (Function Code 19)

KMF-Encrypted-AES-192 (Function Code 27)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KMF-AES-192 function has the following format:

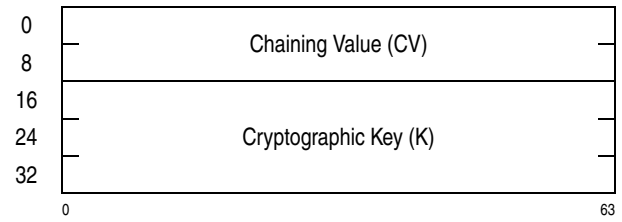


Figure 7-116. Parameter Block for KMF-AES-192

For the KMF-AES-192 function, the chaining value is in byte offset 0-15 of the parameter block and the cryptographic key is in byte offsets 16-39 of the parameter block.

The parameter block used for the KMF-Encrypted-AES-192 function has the following format:

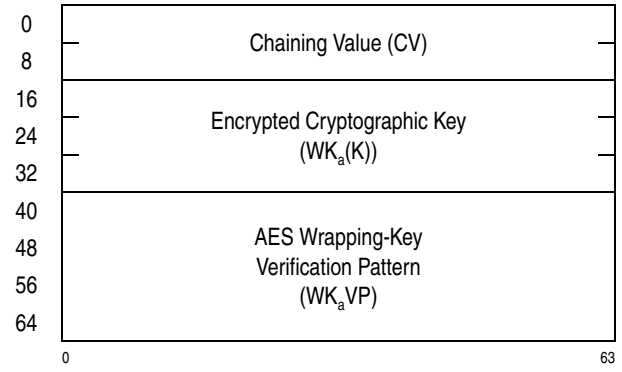


Figure 7-117. Parameter Block for KMF-Encrypted-AES-192

For the KMF-Encrypted-AES-192 function, the contents of byte offsets 40-71 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-15 of the parameter block contain the chaining value, and the contents of byte offsets 16-39 of the parameter block are deciphered using the AES wrapping key to obtain the 192-bit cryptographic key. (See the section “Protection of Cryptographic Keys” on page 7-424 for details.)

The following description applies to both functions.

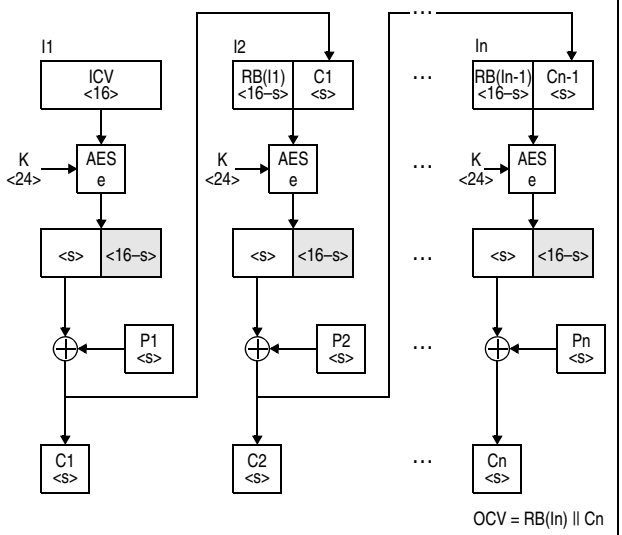
When the modifier bit in general register 0 is zero, an encipher operation is performed. The s-byte plaintext

segments (P1, P2, …, Pn) in operand 2 are enciphered using the AES-encryption algorithm with the 192-bit cryptographic key and the 128-bit chaining value, where s is the length of cipher feedback in bytes.

The first input block to the AES-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. The s leftmost bytes of each output block are exclusive-ORed with the corresponding s-byte plaintext segment to form a s-byte ciphertext segment. The remaining (16 - s) bytes of each output block are ignored. Each ciphertext segment is concatenated to the right of the (16 - s) rightmost bytes of the previous input block to form the next input block.

The process is repeated with the successive input blocks until a ciphertext segment is produced for every plaintext segment, or until a CPU-determined number of ciphertext segments have been produced.

The ciphertext segments (C1, C2, …, Cn) are stored in operand 1. The final next-input block (OCV) is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-118.



Symbol	Explanation
||	Concatenation
Ij	Input block j to AES
RB(Ij)	Rightmost (16-s) bytes of input block j
Note:	The rightmost (16–s) bytes of the output block from the AES-encipher operation are ignored.

Figure 7-118. KMF-AES-192 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The s-byte ciphertext segments (C1, C2, …, Cn) in operand 2 are deciphered using the AES-encryption algorithm with the 192-bit cryptographic key and the 128-bit chaining value, where s is the length of cipher feedback in bytes.

The first input block to the AES-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. The s leftmost bytes of each output block are exclusive-ORed with the corresponding s-byte ciphertext segment to form a s-byte plaintext segment. The remaining (16 - s) bytes of each output block are ignored. Each ciphertext segment is concatenated to the right of the (16 - s) rightmost bytes of the previous input block to form the next input block.

The process is repeated with the successive input blocks until a plaintext segment is produced for every ciphertext segment, or until a CPU-determined number of plaintext segments have been produced.

The plaintext segments (P1, P2, …, Pn) are stored in operand 1. The final next-input block (OCV) is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-119.

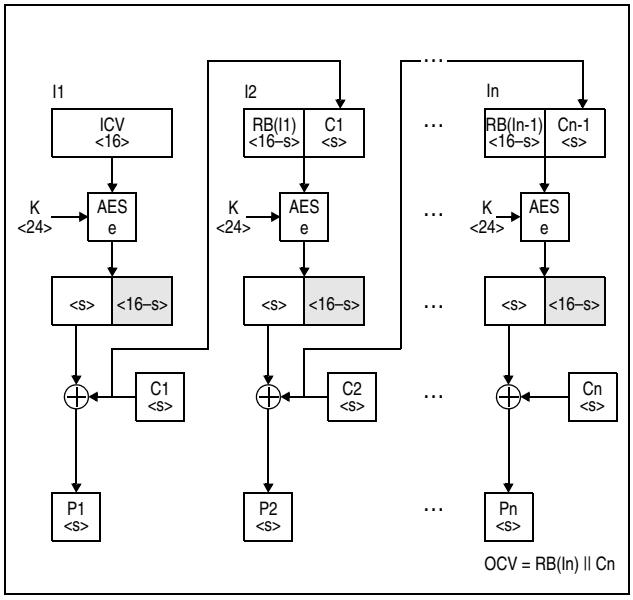


Figure 7-119. KMF-AES-192 Decipher Operation

Symbol	Explanation
||	Concatenation
Ij	Input block j to AES
RB(Ij)	Rightmost (16-s) bytes of input block j
Note:	The rightmost (16–s) bytes of the output block from the AES-decipher operation are ignored.

Figure 7-119. KMF-AES-192 Decipher Operation

KMF-AES-256 (Function Code 20)

KMF-Encrypted-AES-256 (Function Code 28)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-7 on page 7-55.

The parameter block used for the KMF-AES-256 function has the following format:

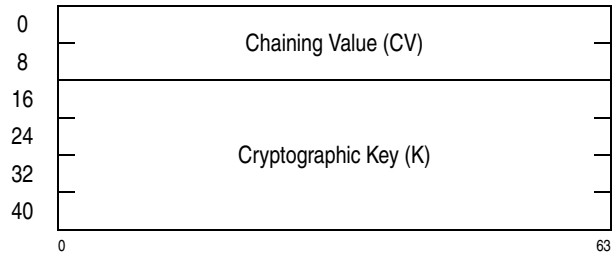


Figure 7-120. Parameter Block for KMF-AES-256

For the KMF-AES-256 function, the chaining value is in byte offset 0-15 of the parameter block and the cryptographic key is in byte offsets 16-47 of the parameter block.

The parameter block used for the KMF-Encrypted-AES-256 function has the following format:

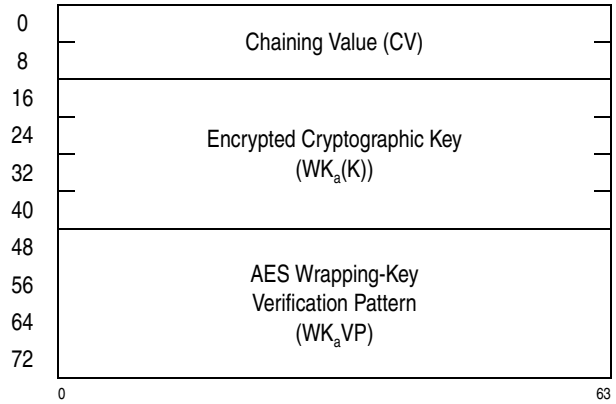


Figure 7-121. Parameter Block for KMF-Encrypted-AES-256

For the KMF-Encrypted-AES-256 function, the contents of byte offsets 48-79 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-15 of the parameter block contain the chaining value, and the contents of byte offsets 16-47 of the parameter block are deciphered using the AES wrapping key to obtain the 256-bit cryptographic key. (See the section "Protection of Cryptographic Keys" on page 7-424 for details.)

The following description applies to both functions.

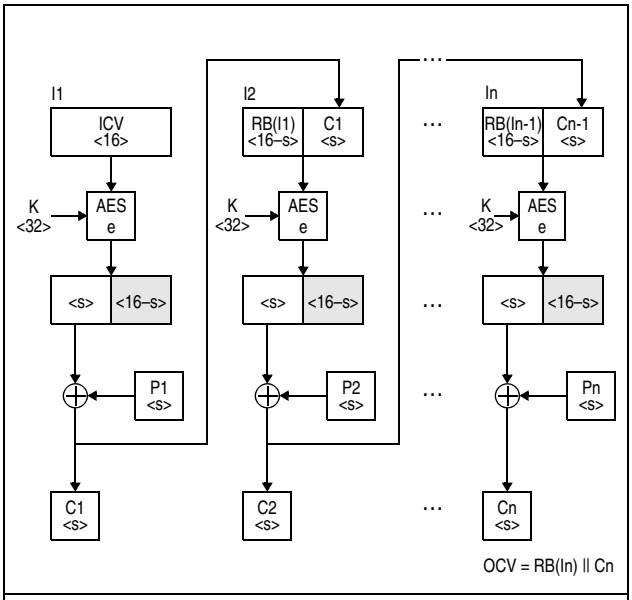
When the modifier bit in general register 0 is zero, an encipher operation is performed. The s-byte plaintext segments (P1, P2, …, Pn) in operand 2 are enciphered using the AES-encryption algorithm with the 256-bit cryptographic key and the 128-bit chaining value, where s is the length of cipher feedback in bytes.

The first input block to the AES-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. The s leftmost bytes of each output block are exclusive-ORed with the corresponding s-byte plaintext segment to form a s-byte ciphertext segment. The remaining (16 - s) bytes of each output block are ignored. Each ciphertext segment is concatenated to the right of the (16 - s) rightmost bytes of the previous input block to form the next input block.

The process is repeated with the successive input blocks until a ciphertext segment is produced for every plaintext segment, or until a CPU-determined number of ciphertext segments have been produced.

The ciphertext segments (C1, C2, …, Cn) are stored in operand 1. The final next-input block (OCV) is

stored into the chaining-value field of the parameter block.The operation is shown in Figure 7-122.



Symbol	Explanation
||	Concatenation
Ij	Input block j to AES
RB(Ij)	Rightmost (16-s) bytes of input block j
Note:	The rightmost (16–s) bytes of the output block from the AES-encipher operation are ignored.

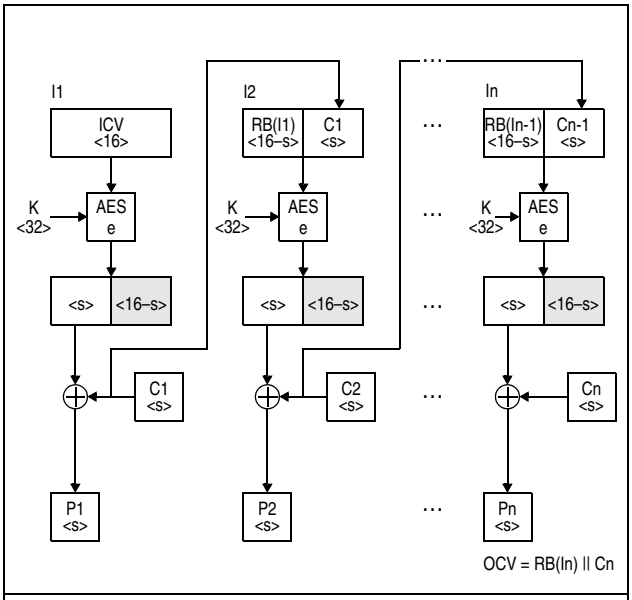
Figure 7-122. KMF-AES-256 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The s-byte ciphertext segments (C1, C2, …, Cn) in operand 2 are deciphered using the AES-encryption algorithm with the 256-bit cryptographic key and the 128-bit chaining value, where s is the length of cipher feedback in bytes.

The first input block to the AES-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. The s leftmost bytes of each output block are exclusive-ORed with the corresponding s-byte ciphertext segment to form a s-byte plaintext segment. The remaining (16 - s) bytes of each output block are ignored. Each ciphertext segment is concatenated to the right of the (16 - s) rightmost bytes of the previous input block to form the next input block.

The process is repeated with the successive input blocks until a plaintext segment is produced for every ciphertext segment, or until a CPU-determined number of plaintext segments have been produced.

The plaintext segments (P1, P2, …, Pn) are stored in operand 1. The final next-input block is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-123.



Symbol	Explanation
||	Concatenation
Ij	Input block j to AES
RB(Ij)	Rightmost (16-s) bytes of input block j
Note:	The rightmost (16–s) bytes of the output block from the AES-decipher operation are ignored.

Figure 7-123. KMF-AES-256 Decipher Operation

Special Conditions

A specification exception is recognized and no other action is taken if any of the following occurs:

1. For DEA or TDEA functions, bits 32-39 of general register 0 specify a value that is zero or greater than 8.
2. For AES functions, bits 32-39 of general register 0 specify a value that is zero or greater than 16.
3. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
4. The R_1 or R_2 field designates an odd-numbered register or general register 0.
5. The second operand length is not a multiple of the length of cipher feedback. This specification-exception condition does not apply to the query functions.

Resulting Condition Code:

0 Normal completion
1 Verification-pattern mismatch
2 --
3 Partial completion

Program Exceptions:

- Access (fetch, operand 2, cryptographic key, and wrapping-key verification pattern; store, operand 1; fetch and store, chaining value)
- Operation (if the message-security-assist extension 4 is not installed)
- Specification
- Transaction constraint

1.-6.	Exceptions with the same priority as the priority of program-interruption conditions for the general case.
7.A	Access exceptions for second instruction halfword.
7.B	Operation exception.
7.C	Transaction constraint
8.	Specification exception due to invalid function code or invalid register number.
9.	Specification exception due to invalid operand length.
10.	Condition code 0 due to second-operand length originally zero.
11.A.1	Access exceptions for an access to the parameter block.
11.A.2.	Condition code 1 due to verification-pattern mismatch.
11.B	Access exceptions for an access to the first, or second operand.
12.	Condition code 0 due to normal completion (second-operand length originally nonzero, but stepped to zero).
13.	Condition code 3 due to partial completion (second-operand length still nonzero).

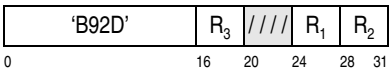
Figure 7-124. Priority of Execution: KMF

Programming Notes:

1. The performance of the query function (code 0) may be significantly slower than that of simply examining a location in storage. If the program needs to frequently test for the availability of a function, it should perform the query function once during initialization; subsequently it should examine the stored results of the query function in memory with an instruction such as TEST UNDER MASK.
2. See the programming notes for CIPHER MESSAGE WITH CHAINING.
3. In earlier versions of the architecture, CIPHER MESSAGE WITH CIPHER FEEDBACK was known as CIPHER MESSAGE WITH CFB.
4. As observed by this CPU, other CPUs, and the I/O subsystem, inconsistent results may be briefly stored in the first-operand location. See the section “Effects of CPU Retry” on page 11-3 for further details.

CIPHER MESSAGE WITH COUNTER

KMCTR R_1,R_3,R_2 [RRF-b]



A function specified by the function code in general register 0 is performed.

Bits 20-23 of the instruction are ignored.

Bit positions 57-63 of general register 0 contain the function code. Figure 7-125 shows the assigned function codes for CIPHER MESSAGE WITH COUNTER. All other function codes are unassigned. For cipher functions, bit 56 is the modifier bit which specifies whether an encryption or a decryption operation is to be performed. The modifier bit is ignored for all other functions. All other bits of general register 0 are ignored.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents

of bit positions 0-63 of general register 1 constitute the address.

The function codes for CIPHER MESSAGE WITH COUNTER are as follows.

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
0	KMCTR-Query	16	—
1	KMCTR-DEA	8	8
2	KMCTR-TDEA-128	16	8
3	KMCTR-TDEA-192	24	8
9	KMCTR-Encrypted-DEA	32	8
10	KMCTR-Encrypted-TDEA-128	40	8
11	KMCTR-Encrypted-TDEA-192	48	8
18	KMCTR-AES-128	16	16
19	KMCTR-AES-192	24	16
20	KMCTR-AES-256	32	16
26	KMCTR-Encrypted-AES-128	48	16
27	KMCTR-Encrypted-AES-192	56	16
28	KMCTR-Encrypted-AES-256	64	16

Explanation:

— Not applicable

Figure 7-125. Function Codes for CIPHER MESSAGE WITH COUNTER

All other function codes are unassigned.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R_1, R_2, $R_2 + 1$, and R_3 are ignored for the query function.

For all other functions, the second operand is ciphered as specified by the function code using a cryptographic key and counter values, and the result is placed in the first-operand location.

The R_1 and R_3 fields each designate a general register and must designate an even-numbered register other than general register 0; otherwise, a specification exception is recognized. The R_2 field designates an even-odd pair of general registers and must designate an even-numbered register other than general register 0; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first, second, and third operands is specified by the contents of the R_1, R_2, and R_3 general registers, respectively. The number of bytes in the second-operand location is specified in general register $R_2 + 1$. The first operand and the third operand are the same length as the second operand.

As part of the operation, the addresses in general registers R_1, R_2, and R_3 are incremented by the number of bytes processed, and the length in general register $R_2 + 1$ is decremented by the same number. The formation and updating of the addresses and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1, R_2, and R_3 constitute the addresses of the first, second, and third operands, respectively, and the contents of bit positions 0-39 are ignored; bits 40-63 of the updated addresses replace the corresponding bits in general registers R_1, R_2, and R_3, carries out of bit position 40 of the updated address are ignored, and the contents of bit positions 32-39 of general registers R_1, R_2, and R_3 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general registers R_1, R_2, and R_3 constitute the addresses of the first, second, and third operands, respectively, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated addresses replace the corresponding bits in general registers R_1, R_2, and R_3, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general registers R_1, R_2, and R_3 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general registers R_1, R_2, and R_3 constitute the addresses of the first, second, and third operands, respectively; bits 0-63 of the updated addresses replace the contents of general registers R_1, R_2, and R_3, and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_2 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the first, second, and third operands, and the contents of bit positions 0-31 are ignored; bits 32-63 of the updated value replace the corresponding bits in general register $R_2 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the first, second, and third operands; the updated value replaces the contents of general register $R_2 + 1$.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, R_2, $R_2 + 1$, and R_3 always remain unchanged.

Figure 7-126 on page 7-109 shows the contents of the general registers just described.

Figure 7-126. General Register Assignment for KMCTR

In the access-register mode, access registers 1, R_1, R_2, and R_3 specify the address spaces containing the parameter block, first, second, and third operands, respectively.

The result is obtained as if processing starts at the left end of the first, second, and third operands and proceeds to the right, block by block. The operation is ended when the number of bytes in the second oper-

and as specified in general register $R_2 + 1$ have been processed and placed at the first-operand location (called normal completion) or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

The results in the first-operand location are unpredictable if any of the following situations occurs:

1. The cryptographic-key field or the encrypted cryptographic-key field overlaps any portion of the first operand.
2. The first and second operands overlap destructively.
3. The first and third operands overlap destructively.

Operands are said to overlap destructively when the first-operand location would be used as a source after data would have been moved into it, assuming processing to be performed from left to right and one byte at a time.

When the operation ends due to normal completion, condition code 0 is set and the resulting value in $R_2 + 1$ is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in $R_2 + 1$ is nonzero.

A PER storage-alteration event may be recognized both for the first-operand location and for the portion of the parameter block that is stored. A PER zero-address-detection event may be recognized for the first- , second-, and third-operand locations and for the parameter block. When PER events are detected for one or more of these locations, it is unpredictable which location is identified in the PER access identification (PAID) and PER ASCE ID (AI).

When a storage-alteration PER event is recognized, fewer than 4K additional bytes are stored into the first-operand locations before the event is reported.

When the second-operand length is initially zero, the parameter block, first, second, and third operands are not accessed, general registers R_1, R_2, $R_2 + 1$, and R_3 are not changed, and condition code 0 is set.

When the contents of any two or all three of the R_1, R_2, and R_3 fields are the same, the contents of the designated registers are incremented only by the number of bytes processed, not by a multiple of the number of bytes processed.

As observed by this CPU, other CPUs, and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

In certain unusual situations, instruction execution may complete by setting condition code 3 without updating the registers to reflect the last unit of the first, second, and third operands processed. The size of the unit processed in this case depends on the situation and the model, but is limited such that the portion of the first and second operands which have been processed and not reported do not overlap in storage; and the portion of the first and third operands which have been processed and not reported do not overlap in storage. In all cases, change bits are set and PER storage-alteration events are reported, when applicable, for all first-operand locations processed.

For functions that perform a comparison of the encrypted cryptographic key with the wrapping-key verification-pattern register, it is unpredictable whether access exceptions and PER zero-address-detection events are recognized for the first, second, and third operands when the comparison results in a mismatch.

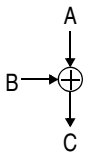
Access exceptions may be reported for a larger portion of an operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of an operand nor for locations more than 4K bytes beyond the current location being processed.

Symbols Used in Function Descriptions

The following symbols are used in the subsequent description of the CIPHER MESSAGE WITH COUNTER functions. For data-encryption-algorithm (DEA) functions, the DEA-key-parity bit in each byte of the DEA key is ignored, and the operation pro-

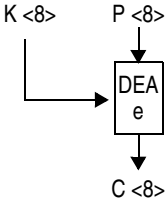
ceeds normally, regardless of the DEA-key parity of the key.

Further description of the data-encryption algorithm may be found in Reference [13.] on page xxx. Further description of the AES standard may be found in Reference [14.] on page xxx. Further description of the counter mode of encryption and decryption may be found in Reference [16.] on page xxx.



$C = A \oplus B$

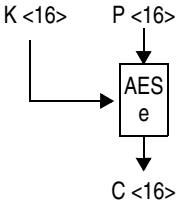
Figure 7-127. Symbol For Bit-Wise Exclusive OR



Symbol for DEA Encryption

Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

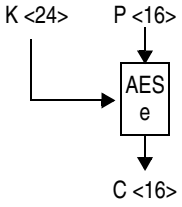
Figure 7-128. Symbols for DEA Encryption



Symbol for AES-128 Encryption

Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

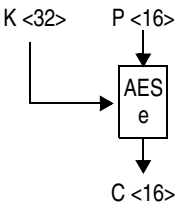
Figure 7-129. Symbols for AES-128 Encryption



Symbol for AES-192 Encryption

Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-130. Symbols for AES-192 Encryption



Symbol for AES-256 Encryption

Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-131. Symbols for AES-256 Encryption

KMCTR-Query (Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-126 on page 7-109.

The parameter block used for the function has the following format:

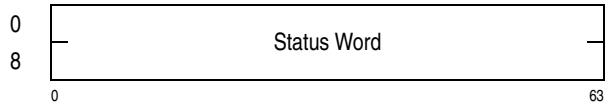


Figure 7-132. Parameter Block for KMCTR-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the CIPHER MESSAGE WITH COUNTER instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KMCTR-Query function completes; condition codes 1 and 3 are not applicable to this function.

KMCTR-DEA (Function Code 1)

KMCTR-Encrypted-DEA (Function Code 9)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-126 on page 7-109.

The parameter block used for the KMCTR-DEA function has the following format:

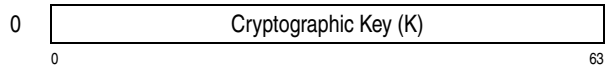


Figure 7-133. Parameter Block for KMCTR-DEA

For the KMCTR-DEA function, the cryptographic key is in byte offsets 0-7 of the parameter block.

The parameter block used for the KMCTR-Encrypted-DEA function has the following format:

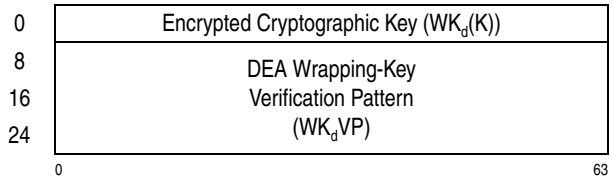


Figure 7-134. Parameter Block for KMCTR-Encrypted-DEA

For the KMCTR-Encrypted-DEA function, the contents of byte offsets 8-31 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 0-7 of the parameter block are deciphered using the DEA wrapping key to obtain the 64-bit cryptographic key. (See the section “Protection of Cryptographic Keys” on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the DEA algorithm with the 64-bit cryptographic key and using 8-byte counter-value blocks (R1, R2, …, Rn) in operand 3. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The operation is shown in Figure 7-135.

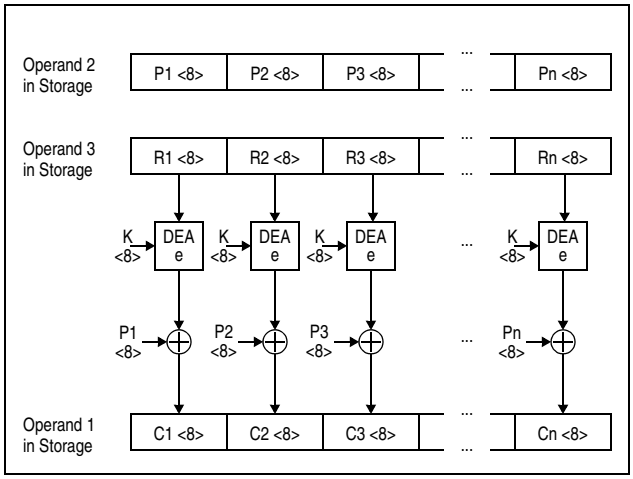


Figure 7-135. KMCTR DEA Encipher Operation Using 64-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the DEA algorithm with the 64-bit cryptographic key and using 8-byte counter-value blocks (R1, R2, …, Rn) in operand 3. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The operation is shown in Figure 7-136.

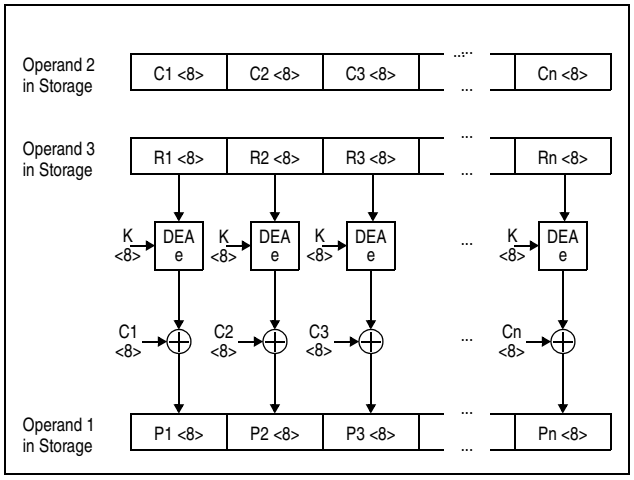


Figure 7-136. KMCTR DEA Decipher Operation Using 64-Bit Key

KMCTR-TDEA-128 (Function Code 2)

KMCTR-Encrypted-TDEA-128 (Function Code 10)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-126 on page 7-109.

The parameter block used for the KMCTR-TDEA-128 function has the following format:

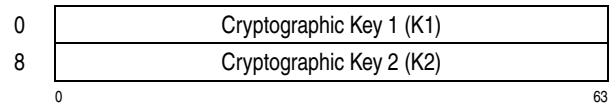


Figure 7-137. Parameter Block for KMCTR-TDEA-128

For the KMCTR-TDEA-128 function, the cryptographic key is in byte offsets 0-15 of the parameter block.

The parameter block used for the KMCTR-Encrypted-TDEA-128 function has the following format:

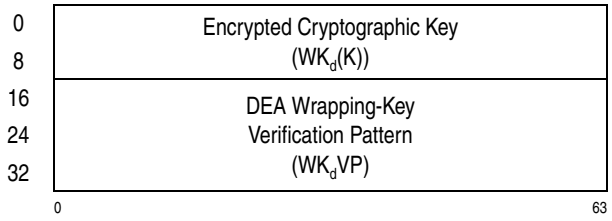


Figure 7-138. Parameter Block for KMCTR Encrypted-TDEA-128

For the KMCTR-Encrypted-TDEA-128 function, the contents of byte offsets 16-39 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 0-15 of the parameter block are deciphered using the DEA wrapping key to obtain the 128-bit cryptographic key, K = K1 || K2, where K1 is the leftmost 64 bits of K and K2 is the rightmost 64 bits of K. (See the section “Protection of Cryptographic Keys” on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the TDEA (triple DEA) algorithm with the two 64-bit cryptographic keys and using 8-byte counter-value blocks (R1, R2, …, Rn) in operand 3. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining.

The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The operation is shown in Figure 7-139.

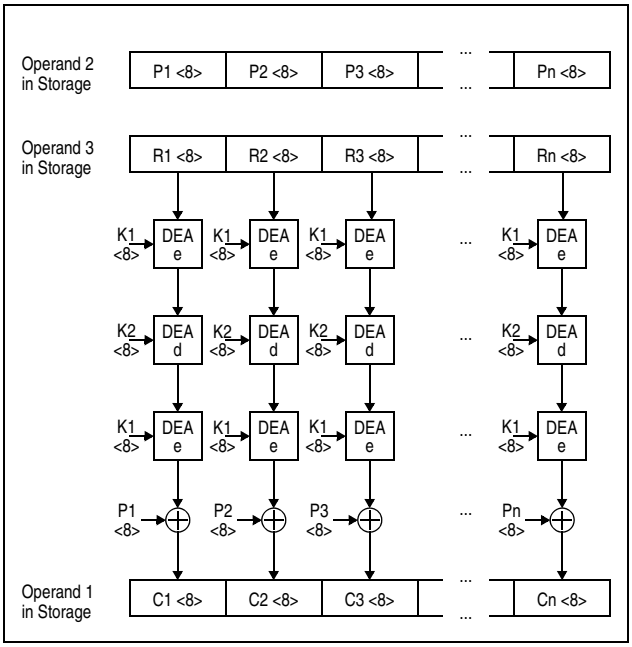


Figure 7-139. KMCTR TDEA Encipher Operation Using 128-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the TDEA algorithm with the two 64-bit cryptographic keys and using 8-byte counter-value blocks (R1, R2, ..., Rn) in operand 3. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The operation is shown in Figure 7-140.

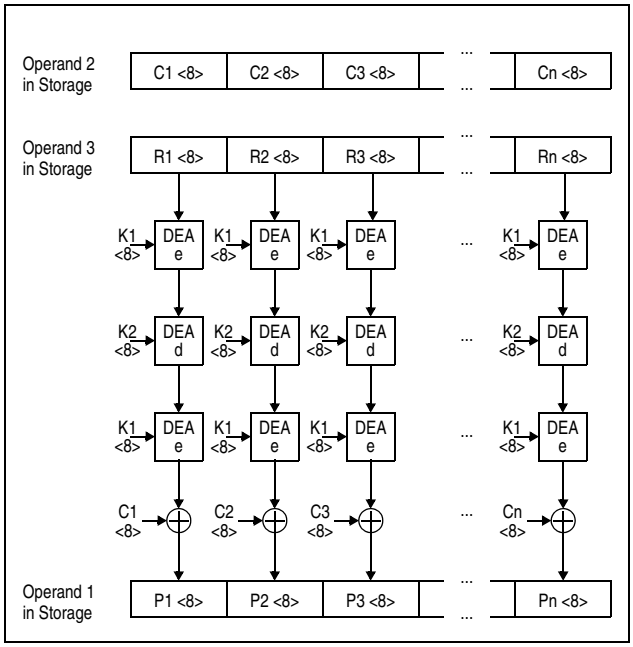


Figure 7-140. KMCTR TDEA Decipher Operation Using 128-Bit Key

KMCTR-TDEA-192 (Function Code 3)

KMCTR-Encrypted-TDEA-192 (Function Code 11)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-126 on page 7-109.

The parameter block used for the KMCTR-TDEA-192 function has the following format:

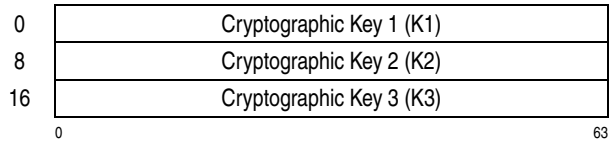


Figure 7-141. Parameter Block for KMCTR-TDEA-192

For the KMCTR-TDEA-192 function, the cryptographic key is in byte offsets 0-23 of the parameter block.

The parameter block used for the KMCTR-Encrypted-TDEA-192 function has the following format:

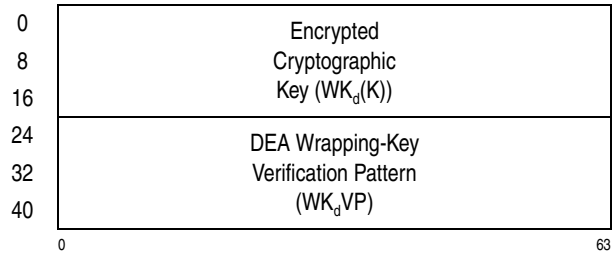


Figure 7-142. Parameter Block for KMCTR-Encrypted-TDEA-192

For the KMCTR-Encrypted-TDEA-192 function, the contents of byte offsets 24-47 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 0-23 of the parameter block are deciphered using the DEA wrapping key to obtain the 192-bit cryptographic key, K = K1 || K2 || K3, where K1 is the leftmost 64 bits of K, K2 is the middle 64 bits of K, and K3 is the rightmost 64 bits of K. (See the section “Protection of Cryptographic Keys” on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the TDEA algorithm with the three 64-bit cryptographic keys and using 8-byte counter-value blocks (R1, R2, …, Rn) in operand 3. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The operation is shown in Figure 7-143.

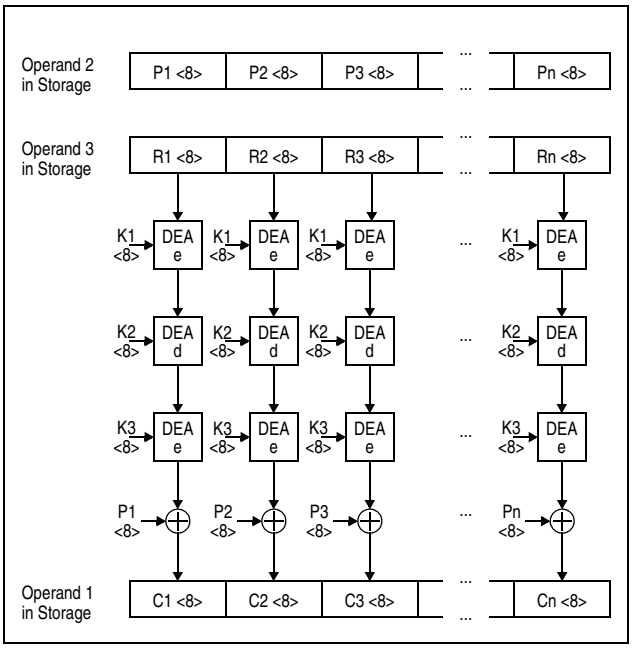


Figure 7-143. KMCTR TDEA Encipher Operation Using 192-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the TDEA algorithm with the three 64-bit cryptographic keys and using 8-byte counter-value blocks (R1, R2, …, Rn) in operand 3. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining.

The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The operation is shown in Figure 7-144.

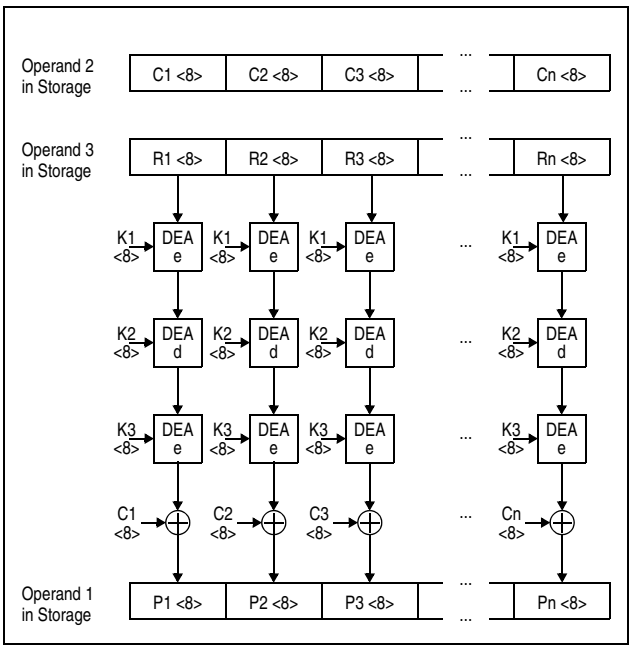


Figure 7-144. KMCTR TDEA Decipher Operation Using 192-Bit Key

KMCTR-AES-128 (Function Code 18)

KMCTR-Encrypted-AES-128 (Function Code 26)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-126 on page 7-109.

The parameter block used for the KMCTR-AES-128 function has the following format:

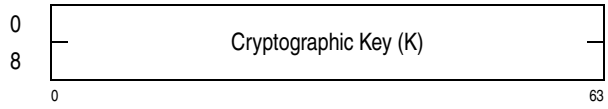


Figure 7-145. Parameter Block for KMCTR-AES-128

For the KMCTR-AES-128 function, the cryptographic key is in byte offsets 0-15 of the parameter block.

The parameter block used for the KMCTR-encrypted-AES-128 function has the following format:

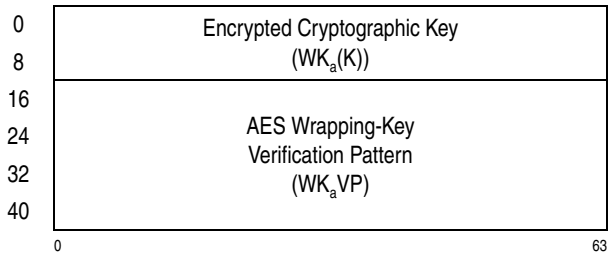


Figure 7-146. Parameter Block for KMCTR-Encrypted-AES-128

For the KMCTR-Encrypted-AES-128 function, the contents of byte offsets 16-47 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 0-15 of the parameter block are deciphered using the AES wrapping key to obtain the 128-bit cryptographic key, K. (See the section "Protection of Cryptographic Keys" on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES algorithm with the 128-bit cryptographic key and using 16-byte counter-value blocks (R1, R2, …, Rn) in operand 3. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The

ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The operation is shown in Figure 7-147.

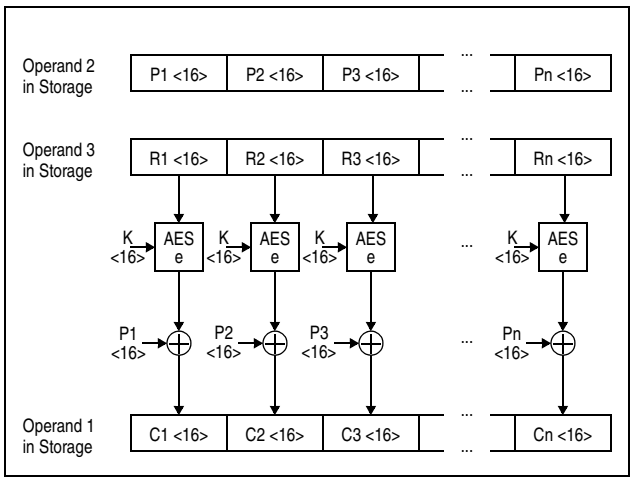


Figure 7-147. KMCTR AES Encipher Operation Using 128-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES algorithm with the 128-bit cryptographic key and using 16-byte counter-value blocks (R1, R2, …, Rn) in operand 3. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The operation is shown in Figure 7-148.

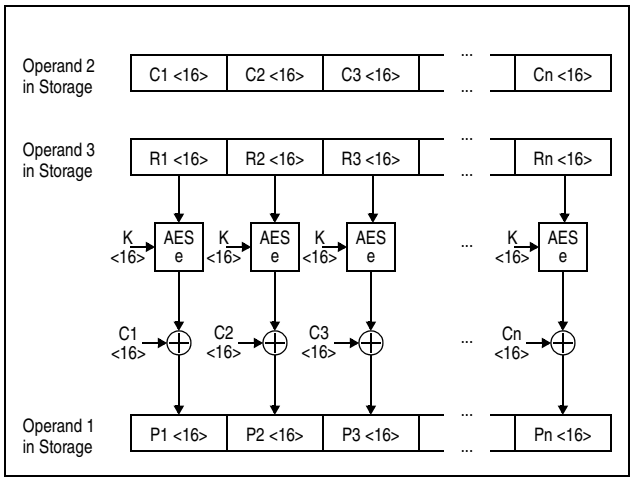


Figure 7-148. KMCTR AES Decipher Operation Using 128-Bit Key

KMCTR-AES-192 (Function Code 19)

KMCTR-Encrypted-AES-192 (Function Code 27)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-126 on page 7-109.

The parameter block used for the KMCTR-AES-192 function has the following format:

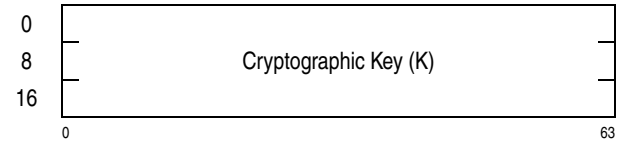


Figure 7-149. Parameter Block for KMCTR-AES-192

For the KMCTR-AES-192 function, the cryptographic key is in byte offsets 0-23 of the parameter block.

The parameter block used for the KMCTR-Encrypted-AES-192 function has the following format:

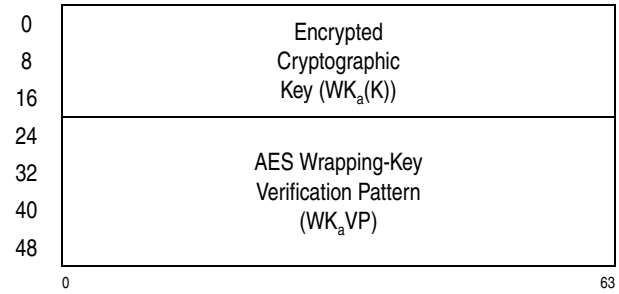


Figure 7-150. Parameter Block for KMCTR-Encrypted-AES-192

For the KMCTR-Encrypted-AES-192 function, the contents of byte offsets 24-55 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 0-23 of the parameter block are deciphered using the AES wrapping key to obtain the 192-bit cryptographic key, K. (See the section “Protection of Cryptographic Keys” on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES algorithm with the 192-bit cryptographic key and using 16-byte counter-value blocks (R1, R2, …, Rn) in operand 3. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The operation is shown in Figure 7-151.

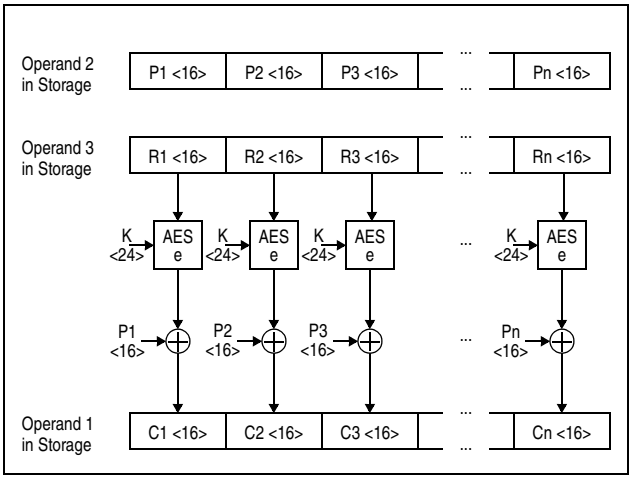


Figure 7-151. KMCTR AES Encipher Operation Using 192-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES algorithm with the 192-bit cryptographic key and using 16-byte counter-value blocks (R1, R2, …, Rn) in operand 3. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The operation is shown Figure 7-152.

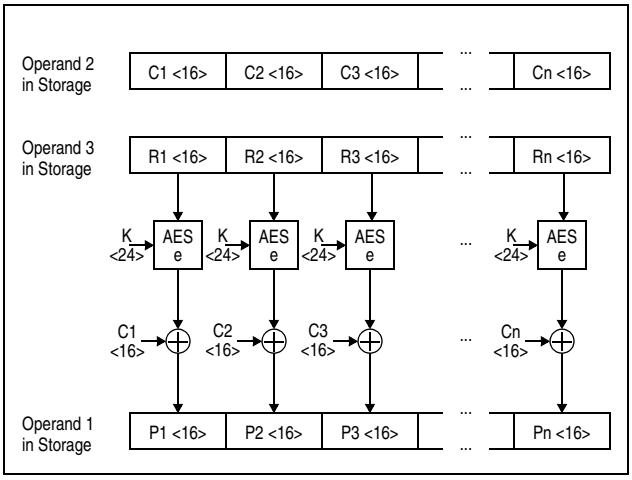


Figure 7-152. KMCTR AES Decipher Operation Using 192-Bit Key

KMCTR-AES-256 (Function Code 20)

KMCTR-Encrypted-AES-256 (Function Code 28)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-126 on page 7-109.

The parameter block used for the KMCTR-AES-256 function has the following format:

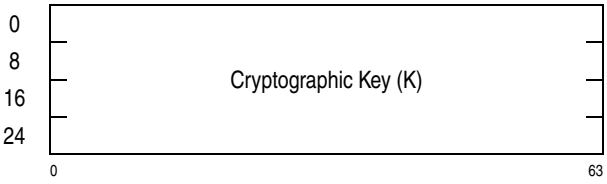


Figure 7-153. Parameter Block for KMCTR-AES-256

For the KMCTR-AES-256 function, the cryptographic key is in byte offsets 0-31 of the parameter block.

The parameter block used for the KMCTR-Encrypted-AES-256 function has the following format:

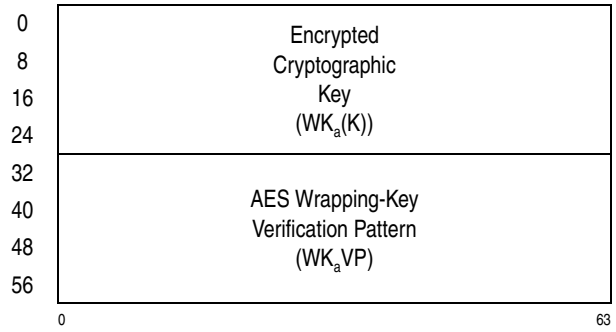


Figure 7-154. Parameter Block for KMCTR-Encrypted-AES-256

For the KMCTR-Encrypted-AES-256 function, the contents of byte offsets 32-63 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, the contents of byte offsets 0-31 of the parameter block are deciphered using the AES wrapping key to obtain the 256-bit cryptographic key, K. (See the section "Protection of Cryptographic Keys" on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES algorithm with the 256-bit cryptographic key and using 16-byte counter-value blocks (R1, R2, …, Rn) in operand 3. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The operation is shown in Figure 7-155.

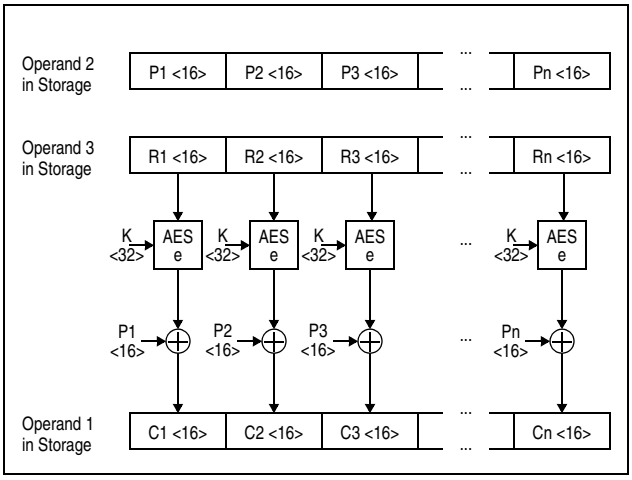


Figure 7-155. KMCTR AES Encipher Operation Using 256-Bit Key

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES algorithm with the 256-bit cryptographic key and using 16-byte counter-value blocks (R1, R2, …, Rn) in operand 3. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The operation is shown in Figure 7-156.

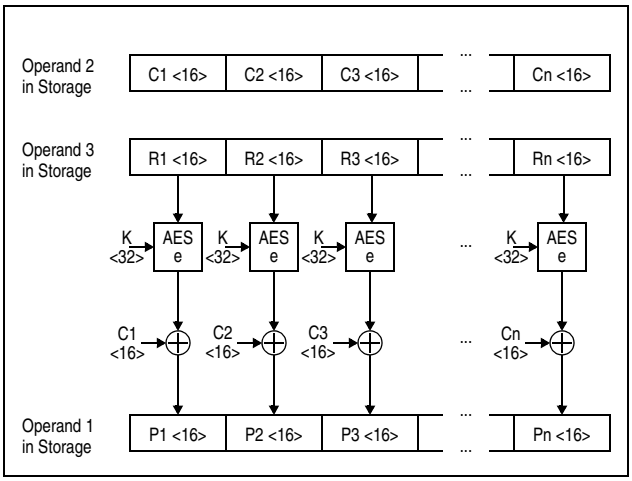


Figure 7-156. KMCTR AES Decipher Operation Using 256-Bit Key

Special Conditions for KMCTR

A specification exception is recognized and no other action is taken if any of the following occurs:

1. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
2. The R_1 R_2, or R_3 field designates an odd-numbered register or general register 0.
3. The second operand length is not a multiple of the data block size of the designated function (see Figure 7-125 on page 7-108 to determine the data block sizes for CIPHER MESSAGE WITH COUNTER functions).This specification-exception condition does not apply to the query functions.

Resulting Condition Code:

0 Normal completion
1 Verification-pattern mismatch
2 --
3 Partial completion

Program Exceptions:

- Access (fetch, operand 2, operand 3, cryptographic key, and wrapping-key verification pattern; store, operand 1)
- Operation (if the message-security-assist extension 4 is not installed)
- Specification
- Transaction constraint

1.-6.	Exceptions with the same priority as the priority of program-interruption conditions for the general case.
7.A	Access exceptions for second instruction halfword.
7.B	Operation exception.
7.C	Transaction constraint
8.	Specification exception due to invalid function code or invalid register number.
9.	Specification exception due to invalid operand length.
10.	Condition code 0 due to second-operand length originally zero.
11.A.1	Access exceptions for an access to the parameter block.
11.A.2.	Condition code 1 due to verification-pattern mismatch.
11.B	Access exceptions for an access to the first, second, or third operand.
12.	Condition code 0 due to normal completion (second-operand length originally nonzero, but stepped to zero).
13.	Condition code 3 due to partial completion (second-operand length still nonzero).

Figure 7-157. Priority of Execution: KMCTR

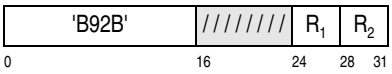
Figure 7-157. Priority of Execution: KMCTR (Continued)

Programming Notes:

1. The performance of the query function (code 0) may be significantly slower than that of simply examining a location in storage. If the program needs to frequently test for the availability of a function, it should perform the query function once during initialization; subsequently it should examine the stored results of the query function in memory with an instruction such as TEST UNDER MASK.
2. As observed by this CPU, other CPUs, and the I/O subsystem, inconsistent results may be briefly stored in the first-operand location. See the section “Effects of CPU Retry” on page 11-3 for further details.

CIPHER MESSAGE WITH OUTPUT FEEDBACK

KMO R_1,R_2 [RRE]



A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction are ignored.

Bit positions 57-63 of general register 0 contain the function code. Figure 7-158 shows the assigned function codes for CIPHER MESSAGE WITH OUTPUT FEEDBACK. All other function codes are unassigned. For cipher functions, bit 56 is the modifier bit which specifies whether an encryption or a decryption operation is to be performed. The modifier bit is ignored for all other functions. All other bits of general register 0 are ignored.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register 1 constitute the address.

The function codes for CIPHER MESSAGE WITH OUTPUT FEEDBACK are as follows.

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
0	KMO-Query	16	—
1	KMO-DEA	16	8
2	KMO-TDEA-128	24	8
3	KMO-TDEA-192	32	8
9	KMO-Encrypted-DEA	40	8
10	KMO-Encrypted-TDEA-128	48	8
11	KMO-Encrypted-TDEA-192	56	8
18	KMO-AES-128	32	16
19	KMO-AES-192	40	16
20	KMO-AES-256	48	16
26	KMO-Encrypted-AES-128	64	16
27	KMO-Encrypted-AES-192	72	16
28	KMO-Encrypted-AES-256	80	16

Explanation:

— Not applicable

Figure 7-158. Function Codes for CIPHER MESSAGE WITH OUTPUT FEEDBACK

All other function codes are unassigned.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R_1, R_2, and $R_2 + 1$ are ignored for the query function.

For all other functions, the second operand is ciphered as specified by the function code using a cryptographic key and an initial chaining value in the parameter block, and the result is placed in the first-operand location. The chaining value is updated as part of the operation.

The R_1 field designates a general register and must designate an even-numbered register other than general register 0; otherwise, a specification exception is recognized. The R_2 field designates an even-odd pair of general registers and must designate an even-numbered register other than general register 0; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first and second operands is specified by the contents of the R_1 and R_2 general registers, respectively. The number of bytes in the second-operand location is specified in general register $R_2 + 1$. The first operand is the same length as the second operand.

As part of the operation, the addresses in general registers R_1 and R_2 are incremented by the number of bytes processed, and the length in general register $R_2 + 1$ is decremented by the same number. The formation and updating of the addresses and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the addresses of the first and second operands, respectively, and the contents of bit positions 0-39 are ignored; bits 40-63 of the updated addresses replace the corresponding bits in general registers R_1 and R_2, carries out of bit position 40 of the updated address are ignored, and the contents of bit positions 32-39 of general registers R_1 and R_2 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general registers R_1 and R_2 constitute the addresses of the first and second operands, respectively, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated addresses replace the corresponding bits in general registers R_1 and R_2, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general registers R_1 and R_2 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general registers R_1 and R_2 constitute the addresses of the first and second operands, respectively; bits 0-63 of the updated addresses replace the contents of general registers R_1 and R_2, and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_2 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the first and second

operands, and the contents of bit positions 0-31 are ignored; bits 32-63 of the updated value replace the corresponding bits in general register $R_2 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the first and second operands; and the updated value replaces the contents of general register $R_2 + 1$.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, R_2, and $R_2 + 1$, always remain unchanged.

Figure 7-159 on page 7-122 shows the contents of the general registers just described.

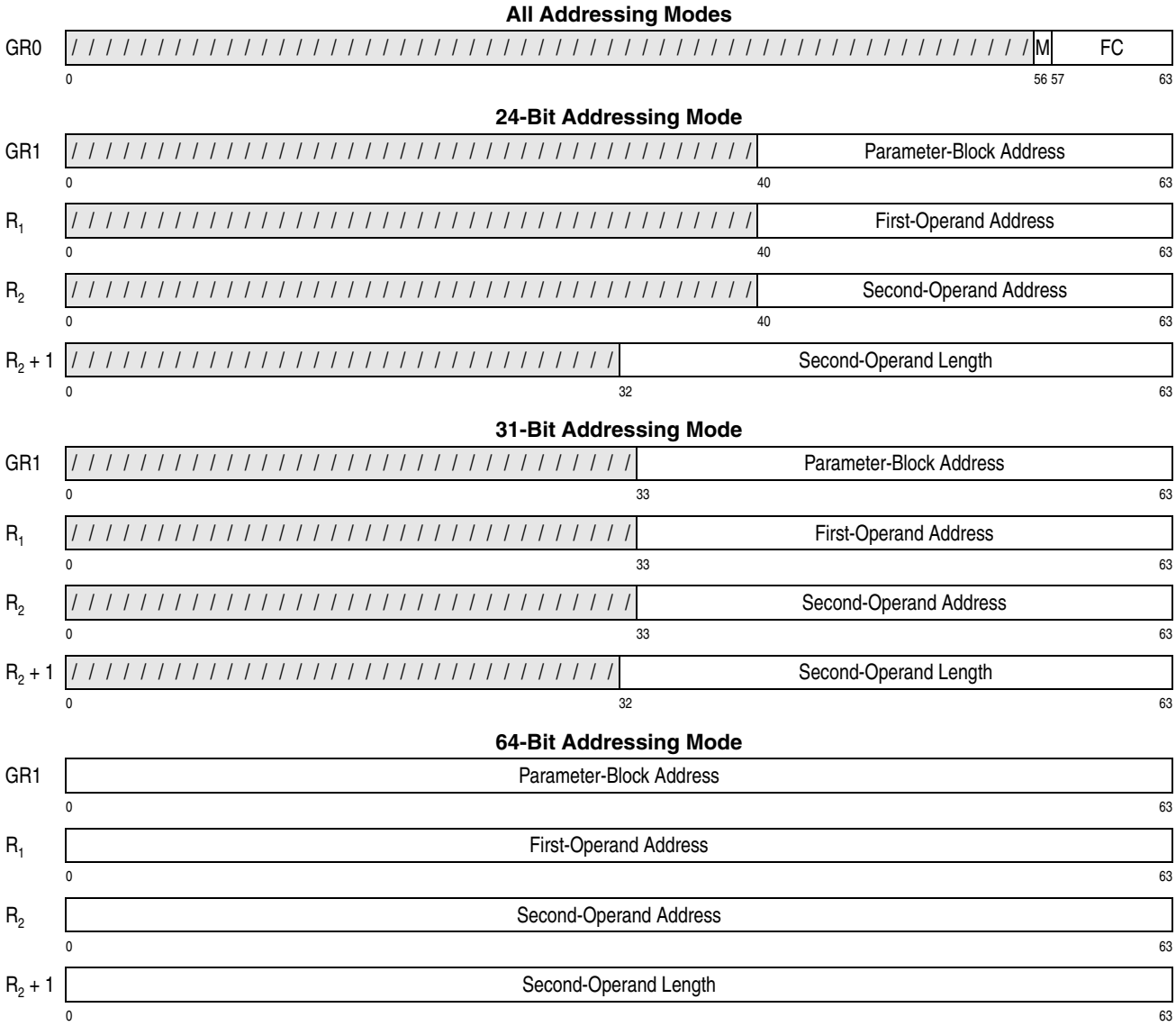


Figure 7-159. General Register Assignment for KMO

In the access-register mode, access registers 1, R_1, and R_2 specify the address spaces containing the parameter block, first, and second operands, respectively.

The result is obtained as if processing starts at the left end of both the first and second operands and proceeds to the right, block by block. The operation is ended when the number of bytes in the second operand as specified in general register $R_2 + 1$ have been processed and placed at the first-operand location

(called normal completion) or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

The results in the first-operand location and the chaining-value field are unpredictable if any of the following situations occur:

1. The cryptographic-key field or the encrypted cryptographic-key field overlaps any portion of the first operand.
2. The chaining-value field overlaps any portion of the first operand or the second operand.
3. The first and second operands overlap destructively. Operands are said to overlap destructively when the first-operand location would be used as a source after data would have been moved into it, assuming processing to be performed from left to right and one byte at a time.

When the operation ends due to normal completion, condition code 0 is set and the resulting value in $R_2 + 1$ is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in $R_2 + 1$ is nonzero.

A PER storage-alteration event may be recognized both for the first-operand location and for the portion of the parameter block that is stored. A PER zero-address-detection event may be recognized for the first- and second-operand locations and for the parameter block. When PER events are detected for one or more of these locations, it is unpredictable which location is identified in the PER access identification (PAID) and PER ASCE ID (AI).

When a storage-alteration PER event is recognized, fewer than 4K additional bytes are stored into the first-operand locations before the event is reported.

When the second-operand length is initially zero, the parameter block, first, and second operands are not accessed, general registers R_1, R_2, and $R_2 + 1$ are not changed, and condition code 0 is set.

When the contents of the R_1 and R_2 fields are the same, the contents of the designated registers are incremented only by the number of bytes processed, not by twice the number of bytes processed.

As observed by this CPU, other CPUs, and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

In certain unusual situations, instruction execution may complete by setting condition code 3 without updating the registers and chaining value to reflect the last unit of the first and second operands processed. The size of the unit processed in this case depends on the situation and the model, but is limited such that the portion of the first and second operands which have been processed and not reported do not overlap in storage. In all cases, change bits are set and PER storage-alteration events are reported, when applicable, for all first-operand locations processed.

For functions that perform a comparison of the encrypted cryptographic key with the wrapping-key verification-pattern register, it is unpredictable whether access exceptions and PER zero-address-detection events are recognized for the first and second operands when the comparison results in a mismatch.

Access exceptions may be reported for a larger portion of an operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of an operand nor for locations more than 4K bytes beyond the current location being processed.

Symbols Used in Function Descriptions

The following symbols are used in the subsequent description of the CIPHER MESSAGE WITH OUTPUT FEEDBACK functions. For data-encryption-algorithm (DEA) functions, the DEA-key-parity bit in each byte of the DEA key is ignored, and the operation proceeds normally, regardless of the DEA-key parity of the key.

Further description of the data-encryption algorithm may be found in Reference [13.] on page xxx. Further description of the AES standard may be found in Reference [14.] on page xxx. Further description of the

output-feedback mode of encryption and decryption may be found in Reference [16.] on page xxx.

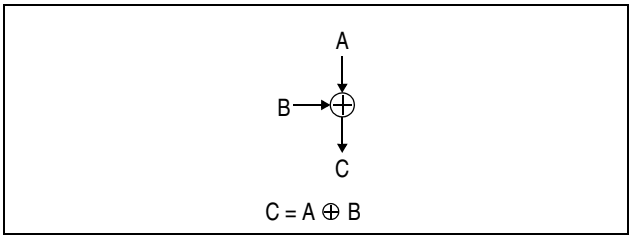
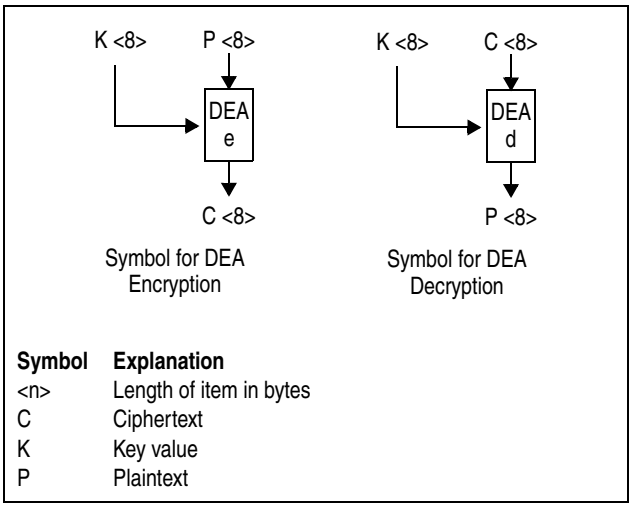
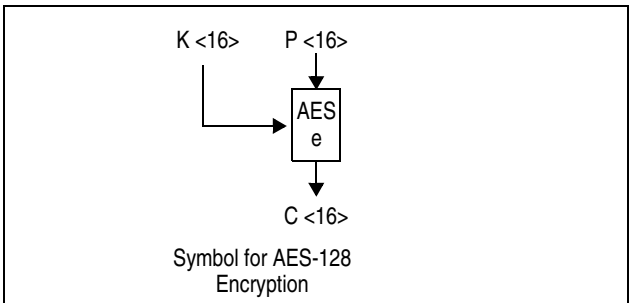


Figure 7-160. Symbol For Bit-Wise Exclusive OR



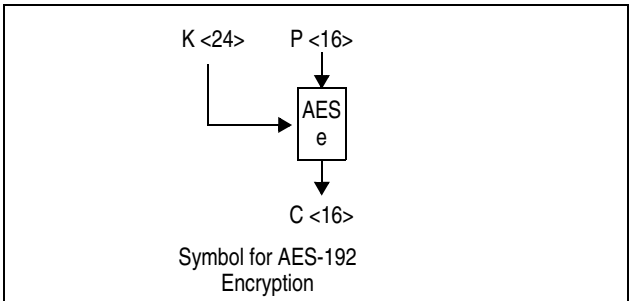
Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-161. Symbols for DEA Encryption and Decryption



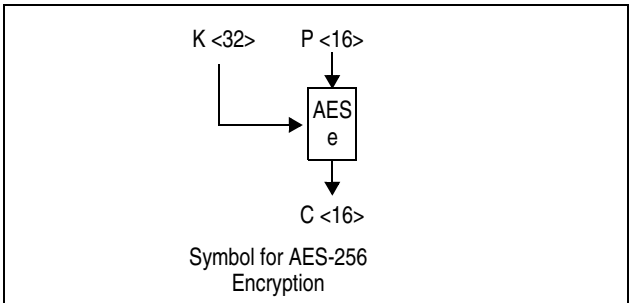
Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-162. Symbols for AES-128 Encryption



Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-163. Symbols for AES-192 Encryption



Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-164. Symbols for AES-256 Encryption

KMO-Query (Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-159 on page 7-122.

The parameter block used for the function has the following format:

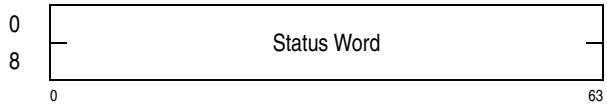


Figure 7-165. Parameter Block for KMO-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the CIPHER MESSAGE WITH OUTPUT FEEDBACK instruction. When a bit

is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KMO-Query function completes; condition codes 1 and 3 are not applicable to this function.

KMO-DEA (Function Code 1)

KMO-Encrypted-DEA (Function Code 9)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-159 on page 7-122.

The parameter block used for the KMO-DEA function has the following format:

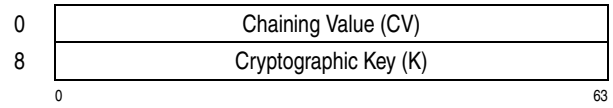


Figure 7-166. Parameter Block for KMO-DEA

For the KMO-DEA function, the chaining value is in byte offset 0-7 of the parameter block and the cryptographic key is in byte offsets 8-15 of the parameter block.

The parameter block used for the KMO-Encrypted - DEA function has the following format:

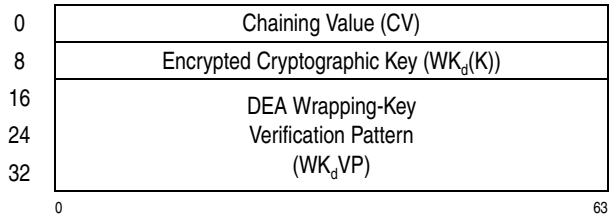


Figure 7-167. Parameter Block for KMO-Encrypted-DEA

For the KMO-Encrypted-DEA function, the contents of byte offsets 16-39 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-7 of the parameter block contain the chaining value, and the contents of byte offsets 8-15 of the parameter block are deciphered using the DEA wrapping key to obtain the 64-bit cryptographic key. (See the section “Protection of Cryptographic Keys” on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the DEA-encryption algorithm with the 64-bit cryptographic key and the 64-bit chaining value.

The first input block to the DEA-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. Each output block is exclusive-ORed with the corresponding plaintext block to form a ciphertext block. Each output block is also used as the next input block.

The process is repeated with the successive input blocks until a ciphertext block is produced for every plaintext block, or until a CPU-determined number of ciphertext blocks have been produced.

The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The next input block, called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-168.

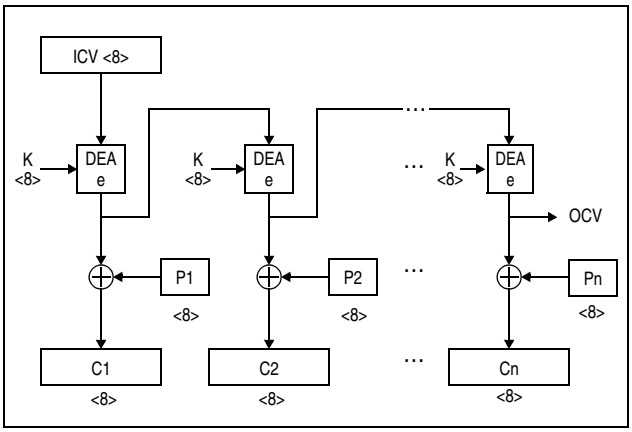


Figure 7-168. KMO-DEA Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the DEA-encryption algorithm with the 64-bit cryptographic key and the 64-bit chaining value.

The first input block to the DEA-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. Each output block is exclusive-ORed

with the corresponding ciphertext block to form a plaintext block. Each output block is also used as the next input block.

The process is repeated with the successive input blocks until a plaintext block is produced for every ciphertext block, or until a CPU-determined number of plaintext blocks have been produced.

The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The next input block, called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-169.

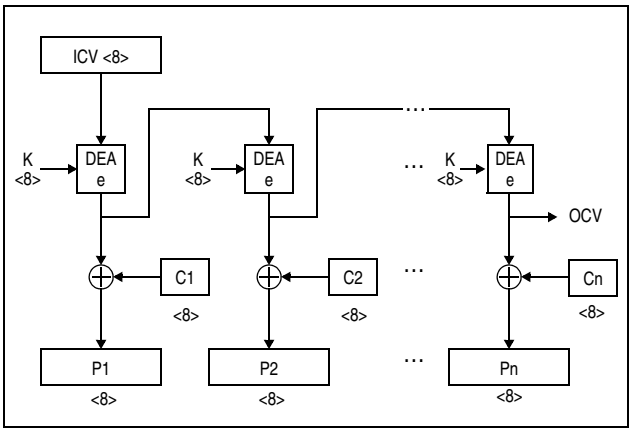


Figure 7-169. KMO-DEA Decipher Operation

KMO-TDEA-128 (Function Code 2)

KMO-Encrypted-TDEA-128 (Function Code 10)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-159 on page 7-122.

The parameter block used for the KMO-TDEA-128 function has the following format:

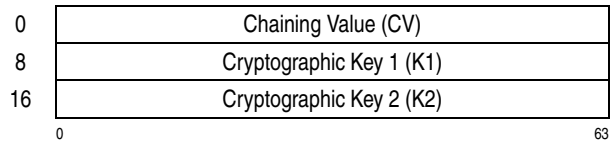


Figure 7-170. Parameter Block for KMO-TDEA-128

For the KMO-TDEA-128 function, the chaining value is in byte offset 0-7 of the parameter block and the cryptographic key is in byte offsets 8-23 of the parameter block.

The parameter block used for the KMO-Encrypted-TDEA-128 function has the following format:

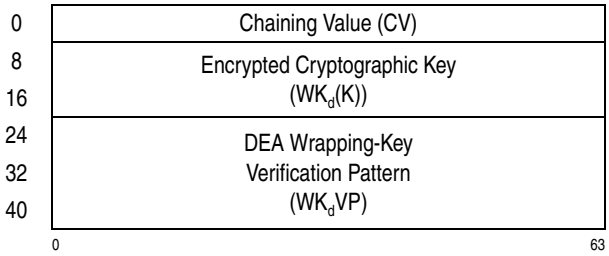


Figure 7-171. Parameter Block for KMO-Encrypted-TDEA-128

For the KMO-Encrypted-TDEA-128 function, the contents of byte offsets 24-47 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-7 of the parameter block contain the chaining value, and the contents of byte offsets 8-23 of the parameter block are deciphered using the DEA wrapping key to obtain the 128-bit cryptographic key. (See the section "Protection of Cryptographic Keys" on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the TDEA-encryption algorithm with the 128-bit cryptographic key and the 64-bit chaining value.

The first input block to the TDEA-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. Each output block is exclusive-ORed with the corresponding plaintext block to form a ciphertext block. Each output block is also used as the next input block.

The process is repeated with the successive input blocks until a ciphertext block is produced for every plaintext block, or until a CPU-determined number of ciphertext blocks have been produced.

The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The next input block, called the output chaining value (OCV), is stored into the chaining-

value field of the parameter block. The operation is shown in Figure 7-172.

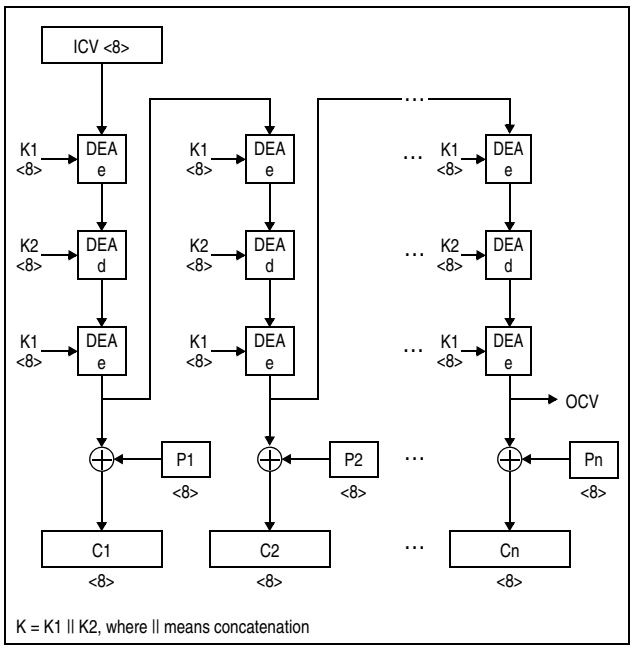


Figure 7-172. KMO-TDEA-128 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the DEA-encryption algorithm with the 128-bit cryptographic key and the 64-bit chaining value.

The first input block to the TDEA-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. Each output block is exclusive-ORed with the corresponding ciphertext block to form a plaintext block. Each output block is also used as the next input block.

The process is repeated with the successive input blocks until a plaintext block is produced for every ciphertext block, or until a CPU-determined number of plaintext blocks have been produced.

The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The next input block, called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-173.

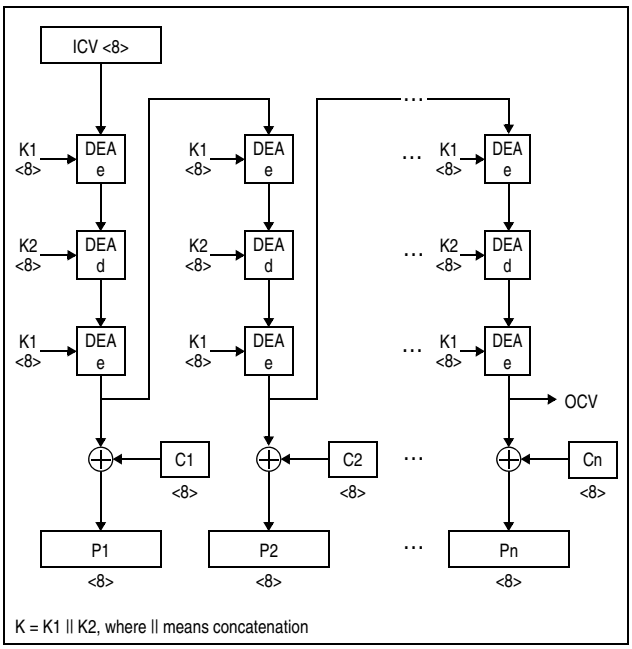


Figure 7-173. KMO-TDEA-128 Decipher Operation

KMO-TDEA-192 (Function Code 3)

KMO-Encrypted-TDEA-192 (Function Code 11)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-159 on page 7-122.

The parameter block used for the KMO-TDEA-192 function has the following format:

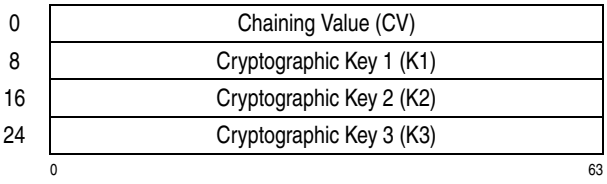


Figure 7-174. Parameter Block for KMO-TDEA-192

For the KMO-TDEA-192 function, the chaining value is in byte offset 0-7 of the parameter block and the cryptographic key is in byte offsets 8-31 of the parameter block.

The parameter block used for the KMO-Encrypted-TDEA-192 function has the following format:

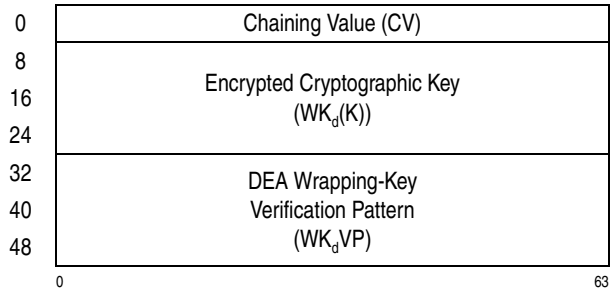


Figure 7-175. Parameter Block for KMO-Encrypted-TDEA-192

For the KMO-Encrypted-TDEA-192 function, the contents of byte offsets 32-55 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-7 of the parameter block contain the chaining value, and the contents of byte offsets 8-31 of the parameter block are deciphered using the DEA wrapping key to obtain the 192-bit cryptographic key. (See the section "Protection of Cryptographic Keys" on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the TDEA-encryption algorithm with the 192-bit cryptographic key and the 64-bit chaining value.

The first input block to the TDEA-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. Each output block is exclusive-ORed with the corresponding plaintext block to form a ciphertext block. Each output block is also used as the next input block.

The process is repeated with the successive input blocks until a ciphertext block is produced for every plaintext block, or until a CPU-determined number of ciphertext blocks have been produced.

The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The next input block, called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. The operation is The operation is shown in Figure 7-176.

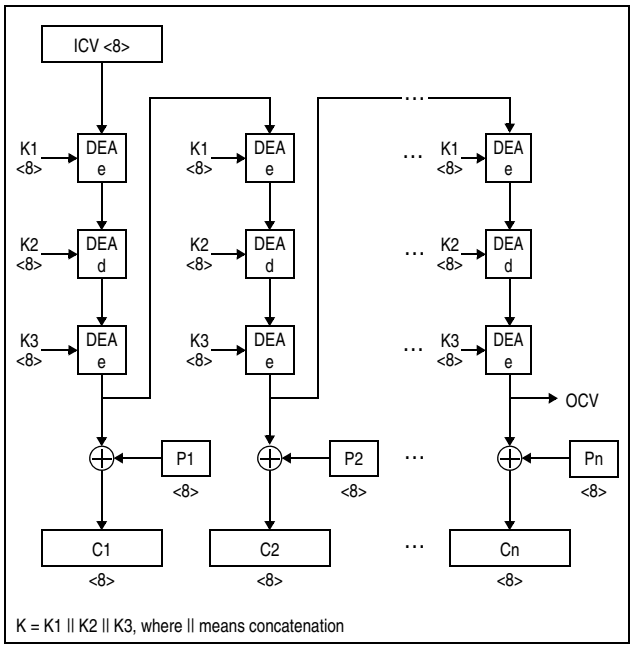


Figure 7-176. KMO-TDEA-192 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the DEA-encryption algorithm with the 192-bit cryptographic key and the 64-bit chaining value.

The first input block to the TDEA-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. Each output block is exclusive-ORed with the corresponding ciphertext block to form a plaintext block. Each output block is also used as the next input block.

The process is repeated with the successive input blocks until a plaintext block is produced for every ciphertext block, or until a CPU-determined number of plaintext blocks have been produced.

The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The next input block, called the output chaining value (OCV), is stored into the chaining-

value field of the parameter block. The operation is shown in Figure 7-177.

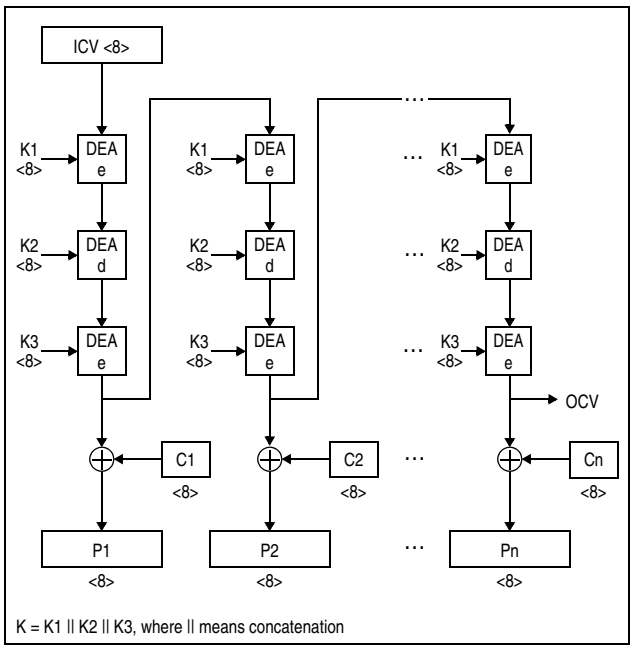


Figure 7-177. KMO-TDEA-192 Decipher Operation

KMO-AES-128 (Function Code 18)

KMO-Encrypted-AES-128 (Function Code 26)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-159 on page 7-122.

The parameter block used for the KMO-AES-128 function has the following format:

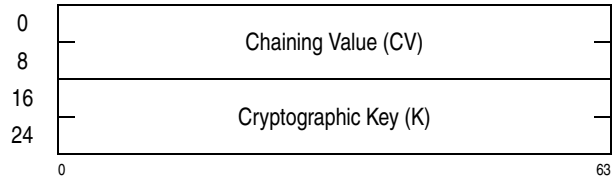


Figure 7-178. Parameter Block for KMO-AES-128

For the KMO-AES-128 function, the chaining value is in byte offset 0-15 of the parameter block and the cryptographic key is in byte offsets 16-31 of the parameter block.

The parameter block used for the KMO-Encrypted-AES-128 function has the following format:

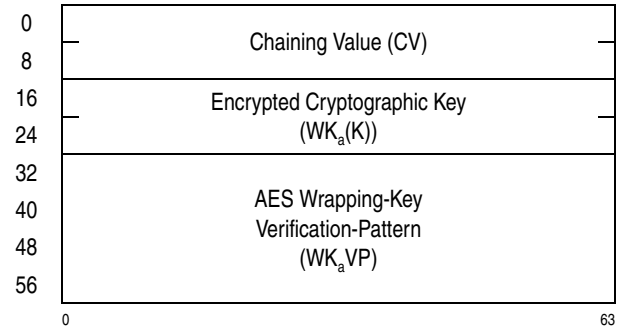


Figure 7-179. Parameter Block for KMO-Encrypted-AES-128

For the KMO-Encrypted-AES-128 function, the contents of byte offsets 32-63 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-15 of the parameter block contain the chaining value, and the contents of byte offsets 16-31 of the parameter block are deciphered using the AES wrapping key to obtain the 128-bit cryptographic key. (See the section “Protection of Cryptographic Keys” on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES-encryption algorithm with the 128-bit cryptographic key and the 128-bit chaining value.

The first input block to the AES-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. Each output block is exclusive-ORed with the corresponding plaintext block to form a ciphertext block. Each output block is also used as the next input block.

The process is repeated with the successive input blocks until a ciphertext block is produced for every plaintext block, or until a CPU-determined number of ciphertext blocks have been produced.

The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The next input block, called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-180.

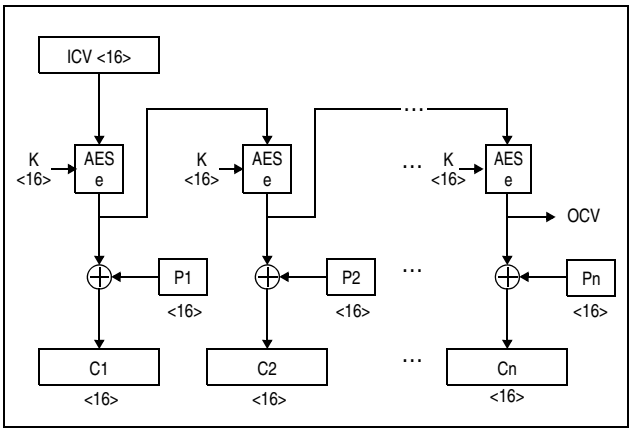


Figure 7-180. KMO-AES-128 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES-encryption algorithm with the 128-bit cryptographic key and the 128-bit chaining value.

The first input block to the AES-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. Each output block is exclusive-ORed with the corresponding ciphertext block to form a plaintext block. Each output block is also used as the next input block.

The process is repeated with the successive input blocks until a plaintext block is produced for every ciphertext block, or until a CPU-determined number of plaintext blocks have been produced.

The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The next input block, called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-181.

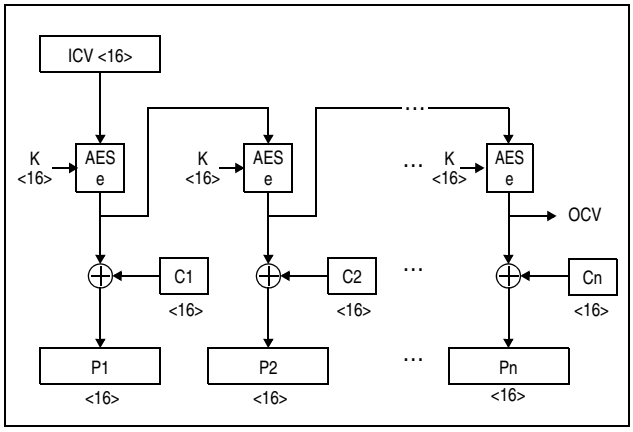


Figure 7-181. KMO-AES-128 Decipher Operation

KMO-AES-192 (Function Code 19)

KMO-Encrypted-AES-192 (Function Code 27)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-159 on page 7-122.

The parameter block used for the KMO-AES-192 function has the following format:

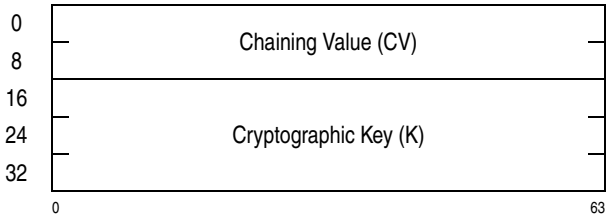


Figure 7-182. Parameter Block for KMO-AES-192

For the KMO-AES-192 function, the chaining value is in byte offset 0-15 of the parameter block and the cryptographic key is in byte offsets 16-39 of the parameter block.

The parameter block used for the KMO-Encrypted-AES-192 function has the following format:

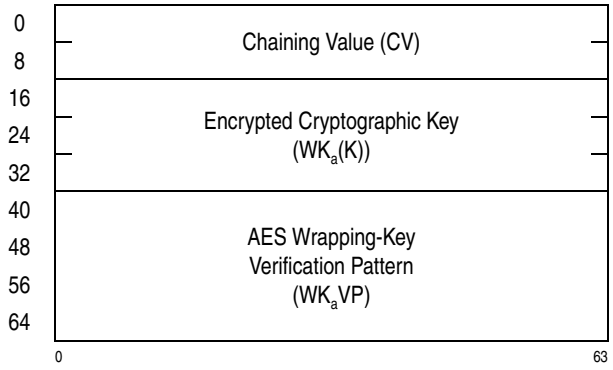


Figure 7-183. Parameter Block for KMO-Encrypted-AES-192

For the KMO-Encrypted-AES-192 function, the contents of byte offsets 40-71 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-15 of the parameter block contain the chaining value, and the contents of byte offsets 16-39 of the parameter block are deciphered using the AES wrapping key to obtain the 192-bit cryptographic key. (See the section "Protection of Cryptographic Keys" on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES-encryption algorithm with the 192-bit cryptographic key and the 128-bit chaining value.

The first input block to the AES-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. Each output block is exclusive-ORed with the corresponding plaintext block to form a ciphertext block. Each output block is also used as the next input block.

The process is repeated with the successive input blocks until a ciphertext block is produced for every plaintext block, or until a CPU-determined number of ciphertext blocks have been produced.

The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The next input block, called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-184.

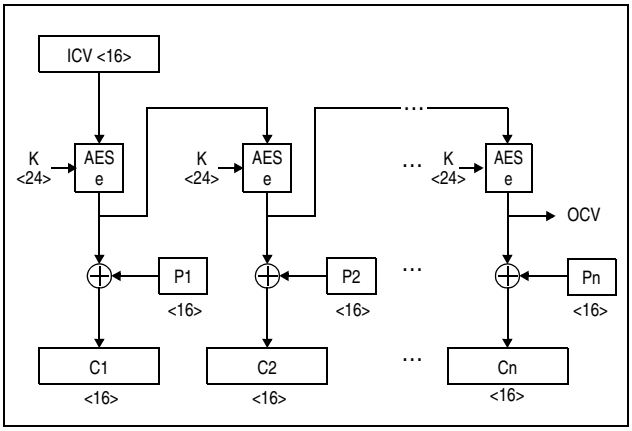


Figure 7-184. KMO-AES-192 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES-encryption algorithm with the 192-bit cryptographic key and the 128-bit chaining value.

The first input block to the AES-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. Each output block is exclusive-ORed with the corresponding ciphertext block to form a plaintext block. Each output block is also used as the next input block.

The process is repeated with the successive input blocks until a plaintext block is produced for every ciphertext block, or until a CPU-determined number of plaintext blocks have been produced.

The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The next input block, called the output chaining value (OCV), is stored into the chaining-

value field of the parameter block. The operation is shown in Figure 7-185.

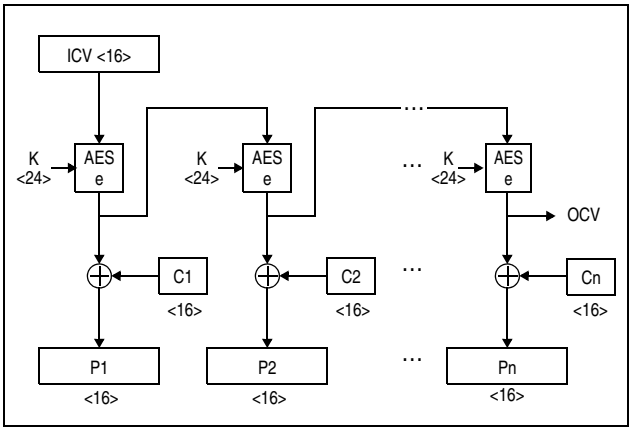


Figure 7-185. KMO-AES-192 Decipher Operation

KMO-AES-256 (Function Code 20)

KMO-Encrypted-AES-256 (Function Code 28)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-159 on page 7-122.

The parameter block used for the KMO-AES-256 function has the following format:

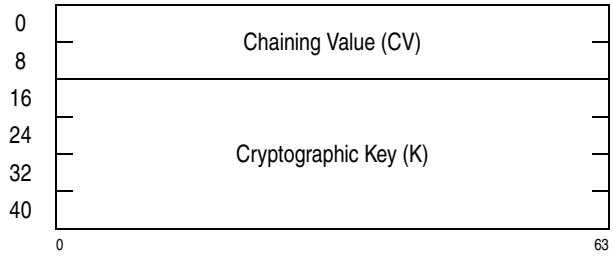


Figure 7-186. Parameter Block for KMO-AES-256

For the KMO-AES-256 function, the chaining value is in byte offset 0-15 of the parameter block and the cryptographic key is in byte offsets 16-47 of the parameter block.

The parameter block used for the KMO-Encrypted-AES-256 function has the following format:

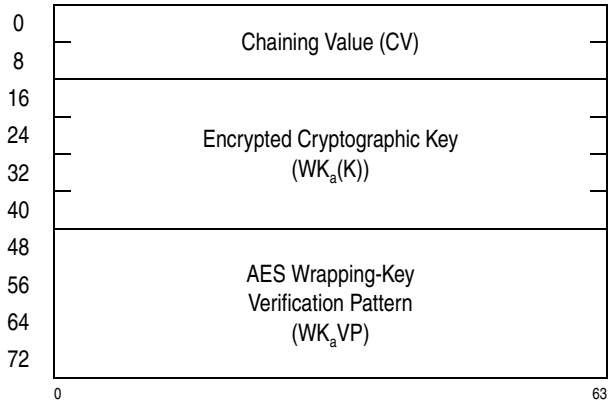


Figure 7-187. Parameter Block for KMO-AES-256

For the KMO-Encrypted-AES-256 function, the contents of byte offsets 48-79 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-15 of the parameter block contain the chaining value, and the contents of byte offsets 16-47 of the parameter block are deciphered using the AES wrapping key to obtain the 256-bit cryptographic key. (See the section “Protection of Cryptographic Keys” on page 7-424 for details.)

The following description applies to both functions.

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 16-byte plaintext blocks (P1, P2, …, Pn) in operand 2 are enciphered using the AES-encryption algorithm with the 256-bit cryptographic key and the 128-bit chaining value.

The first input block to the AES-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. Each output block is exclusive-ORed with the corresponding plaintext block to form a ciphertext block. Each output block is also used as the next input block.

The process is repeated with the successive input blocks until a ciphertext block is produced for every plaintext block, or until a CPU-determined number of ciphertext blocks have been produced.

The ciphertext blocks (C1, C2, …, Cn) are stored in operand 1. The next input block, called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-188.

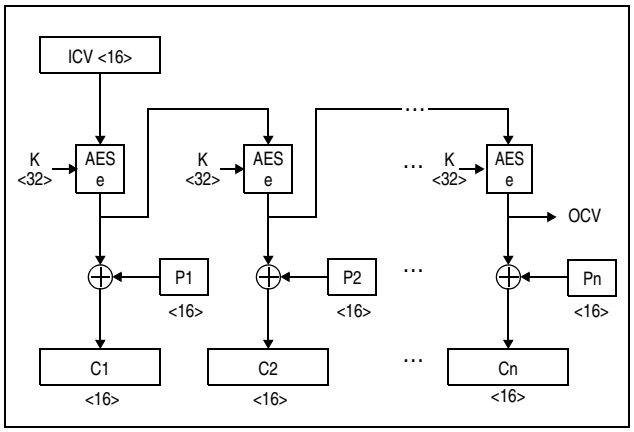


Figure 7-188. KMO-AES-256 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 16-byte ciphertext blocks (C1, C2, …, Cn) in operand 2 are deciphered using the AES-encryption algorithm with the 256-bit cryptographic key and the 128-bit chaining value.

The first input block to the AES-encryption algorithm is the initial chaining value (ICV) in the parameter block. Each input block is enciphered to produce an output block. Each output block is exclusive-ORed with the corresponding ciphertext block to form a plaintext block. Each output block is also used as the next input block.

The process is repeated with the successive input blocks until a plaintext block is produced for every ciphertext block, or until a CPU-determined number of plaintext blocks have been produced.

The plaintext blocks (P1, P2, …, Pn) are stored in operand 1. The next input block, called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. The operation is shown in Figure 7-189.

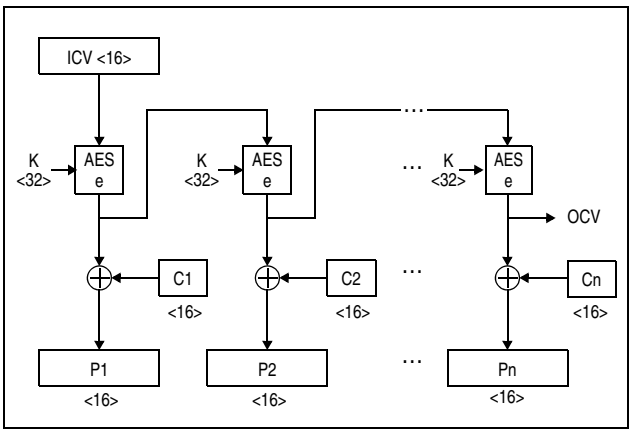


Figure 7-189. KMO-AES-256 Decipher Operation

Special Conditions

A specification exception is recognized and no other action is taken if any of the following occurs:

1. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
2. The R_1 or R_2 field designates an odd-numbered register or general register 0.
3. The second operand length is not a multiple of the data block size of the designated function (see Figure 7-158 on page 7-121 to determine the data block sizes for CIPHER MESSAGE WITH OUTPUT FEEDBACK functions). This specification-exception condition does not apply to the query functions.

Resulting Condition Code:

0 Normal completion
1 Verification-pattern mismatch
2 --
3 Partial completion

Program Exceptions:

- Access (fetch, operand 2, cryptographic key, and wrapping-key verification pattern; store, operand 1; fetch and store, chaining value)
- Operation (if the message-security-assist extension 4 is not installed)
- Specification

- Transaction constraint

1.-6.	Exceptions with the same priority as the priority of program-interruption conditions for the general case.
7.A	Access exceptions for second instruction halfword.
7.B	Operation exception.
7.C	Transaction constraint
8.	Specification exception due to invalid function code or invalid register number.
9.	Specification exception due to invalid operand length.
10.	Condition code 0 due to second-operand length originally zero.
11.A.1	Access exceptions for an access to the parameter block.
11.A.2.	Condition code 1 due to verification-pattern mismatch.
11.B	Access exceptions for an access to the first or second operand.
12.	Condition code 0 due to normal completion (second-operand length originally nonzero, but stepped to zero).
13.	Condition code 3 due to partial completion (second-operand length still nonzero).

Figure 7-190. Priority of Execution: KMO

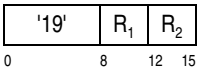
Programming Notes:

1. The performance of the query function (code 0) may be significantly slower than that of simply examining a location in storage. If the program needs to frequently test for the availability of a function, it should perform the query function once during initialization; subsequently it should examine the stored results of the query function in memory with an instruction such as TEST UNDER MASK.
2. See the programming notes for CIPHER MESSAGE WITH CHAINING.
3. In earlier versions of the architecture, CIPHER MESSAGE WITH OUTPUT FEEDBACK was known as CIPHER MESSAGE WITH OFB.
4. As observed by this CPU, other CPUs, and the I/O subsystem, inconsistent results may be briefly stored in the first-operand location. See the section “Effects of CPU Retry” on page 11-3 for further details.

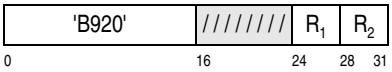
COMPARE

Register-and-register formats:

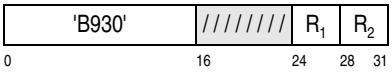
CR R_1,R_2 [RR]



CGR R_1,R_2 [RRE]

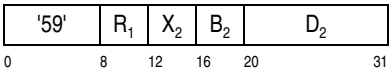


CGFR R_1,R_2 [RRE]

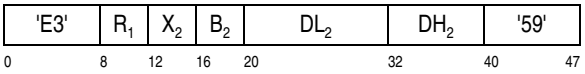


Register-and-storage formats:

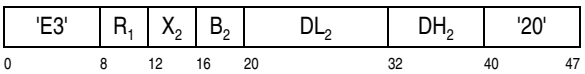
C $R_1,D_2(X_2,B_2)$ [RX-a]



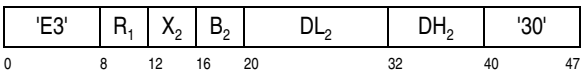
CY $R_1,D_2(X_2,B_2)$ [RXY-a]



CG $R_1,D_2(X_2,B_2)$ [RXY-a]

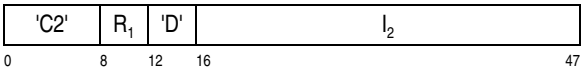


CGF $R_1,D_2(X_2,B_2)$ [RXY-a]

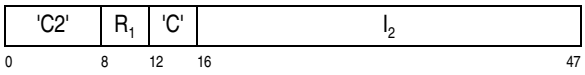


COMPARE IMMEDIATE

CFI R_1,I_2 [RIL-a]

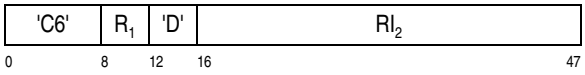


CGFI R_1,I_2 [RIL-a]

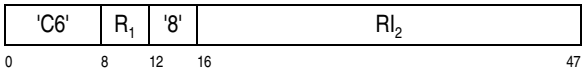


COMPARE RELATIVE LONG

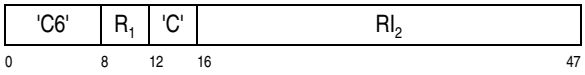
CRL R_1,RI_2 [RIL-b]



CGRL R_1,RI_2 [RIL-b]



CGFRL R_1,RI_2 [RIL-b]



The first operand is compared with the second operand, and the result is indicated in the condition code.

For COMPARE (CR, C, CY), COMPARE IMMEDIATE (CFI), and COMPARE RELATIVE LONG (CRL), the operands are treated as 32-bit signed binary integers. For COMPARE (CGR, CG) and COMPARE RELATIVE LONG (CGRL), they are treated as 64-bit signed binary integers. For COMPARE (CGFR, CGF), COMPARE IMMEDIATE (CGFI), and COMPARE RELATIVE LONG (CGFRL), the second operand is treated as a 32-bit signed binary integer, and the first operand is treated as a 64-bit signed binary integer.

The displacement for COMPARE (C) is treated as a 12-bit unsigned binary integer. The displacement for COMPARE (CY, CG, and CGF) is treated as a 20-bit signed binary integer.

For COMPARE RELATIVE LONG, the contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the address of the second operand in storage. When DAT is on, the second operand is accessed using the same addressing-space mode as that used to access the instruction. When DAT is off, the second operand is accessed using a real address.

For COMPARE RELATIVE LONG (CRL, CGFRL), the second operand must be aligned on a word boundary, and for COMPARE RELATIVE LONG (CGRL), the second operand must be aligned on a doubleword boundary; otherwise, a specification exception is recognized.

Resulting Condition Code:

0 Operands equal
1 First operand low
2 First operand high
3 --

Program Exceptions:

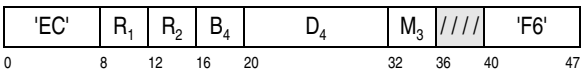
- Access (operand 2 of C, CY, CG, CGF, CRL, CGRL, and CGFRL only)
- Operation (CY, if the long-displacement facility is not installed; CFI and CGFI, if the extended-immediate facility is not installed; CGFRL, CGRL, and CRL, if the general-instructions-extension facility is not installed)
- Specification (CGFRL, CGRL, and CRL only)

Programming Notes:

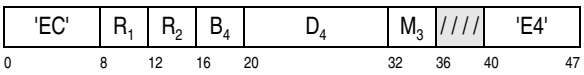
1. For COMPARE RELATIVE LONG, the second operand must be aligned on an integral boundary corresponding to the operand's size.
2. When COMPARE RELATIVE LONG is the target of an execute-type instruction, the second-operand address is relative to the target address.

COMPARE AND BRANCH

CRB $R_1,R_2,M_3,D_4(B_4)$ [RRS]

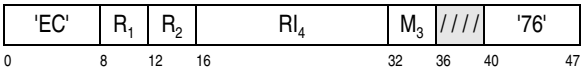


CGRB $R_1,R_2,M_3,D_4(B_4)$ [RRS]

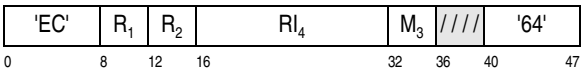


COMPARE AND BRANCH RELATIVE

CRJ R_1,R_2,M_3,RI_4 [RIE-b]

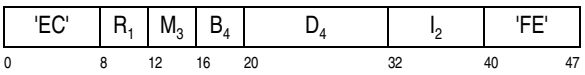


CGRJ R_1,R_2,M_3,RI_4 [RIE-b]

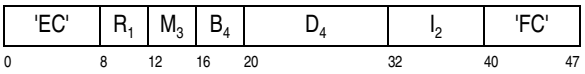


COMPARE IMMEDIATE AND BRANCH

CIB $R_1,I_2,M_3,D_4(B_4)$ [RIS]

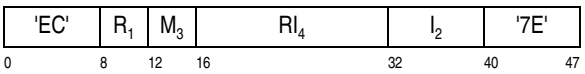


CGIB $R_1,I_2,M_3,D_4(B_4)$ [RIS]

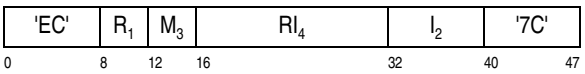


COMPARE IMMEDIATE AND BRANCH RELATIVE

CIJ R_1,I_2,M_3,RI_4 [RIE-c]



CGIJ R_1,I_2,M_3,RI_4 [RIE-c]



The first operand is compared with the second operand. If the mask bit in the M_3 field corresponding to the comparison result is one, the instruction address in the current PSW is replaced by the branch address specified by the fourth operand; otherwise, normal instruction sequencing proceeds with the updated instruction address.

For COMPARE AND BRANCH (CRB), COMPARE AND BRANCH RELATIVE (CRJ), COMPARE IMMEDIATE AND BRANCH (CIB) and COMPARE IMMEDIATE AND BRANCH RELATIVE (CIJ), the first operand is treated as a 32-bit signed binary integer. For COMPARE AND BRANCH (CGRB), COMPARE AND BRANCH RELATIVE (CGRJ), COMPARE IMMEDIATE AND BRANCH (CGIB) and COMPARE IMMEDIATE AND BRANCH RELATIVE (CGIJ), the first operand is treated as a 64-bit signed binary integer.

For COMPARE AND BRANCH (CRB) and COMPARE AND BRANCH RELATIVE (CRJ), the second operand is treated as a 32-bit signed binary integer. For COMPARE AND BRANCH (CGRB) and COMPARE AND BRANCH RELATIVE (CGRJ), the second operand is treated as a 64-bit signed binary integer. For COMPARE IMMEDIATE AND BRANCH and COMPARE IMMEDIATE AND BRANCH RELATIVE, the second operand is treated as an 8-bit signed binary integer.

The comparison results and corresponding M_3 bits are as follows:

Comparison Result	M_3 Bit
Equal	0
First operand low	1
First operand high	2

Bit 3 of the M_3 field is reserved and should be zero; otherwise, the program may not operate compatibly in the future.

For COMPARE AND BRANCH and COMPARE IMMEDIATE AND BRANCH, the fourth operand is used as the branch address. For COMPARE AND BRANCH RELATIVE and COMPARE IMMEDIATE AND BRANCH RELATIVE, the contents of the RI_4 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the branch address.

Condition Code: The code remains unchanged.

Program Exceptions:

- Operation (if the general-instructions-extension facility is not installed).
- Transaction constraint

Programming Notes:

1. When bit positions 0-2 of the M_3 field contain zeros, the instruction acts as a NOP, however this is not the preferred instruction with which to create a NOP. When bit positions 0-2 of the M_3 field contain 111 binary, a branch always occurs.

2. When COMPARE AND BRANCH RELATIVE or COMPARE IMMEDIATE AND BRANCH RELATIVE are the target of an execute-type instruction, the branch is relative to the target address. See “Branch-Address Generation” on page 5-12.

3. The high-level assembler (HLASM) provides extended-mnemonic formats for all of the multiple-operation instructions which perform a compare (arithmetic or logical) followed by a branch or trap. In the extended-mnemonic format, the mask field is not explicitly coded; rather the branch mask is implicitly indicated by the extended mnemonic.

   The extended-mnemonic names are formed by concatenating the base mnemonic, followed by one of the suffix characters, as shown below.

Suffix Chars	Meaning	Mask Field
E	Equal	8
H	First operand high	2
L	First operand low	4
NE	Not equal	6
NH	First operand not high	12
NL	First operand not low	10

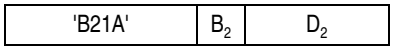
   Consider the following example of a COMPARE IMMEDIATE AND BRANCH RELATIVE (CGIJ) instruction that is coded using an extended mnemonic:

```
        CGIJNE 10,-123,SKIP
          ⋮
SKIP    DS     0H     (128 bytes past CGIJNE)
```

   The implied mask field corresponding to the mnemonic suffix (NE) is 6, thus the generated code is ECA60040857C hex.

COMPARE AND FORM CODEWORD

CFC $D_2(B_2)$ [S]



General register 2 contains an index, which is used along with contents of general registers 1 and 3 to designate the starting addresses of two fields in storage, called the first and third operands. The first and third operands are logically compared, and a codeword is formed in general register 2 for use in sort/merge algorithms.

The second-operand address is not used to address data. Bits 49-62 of the second-operand address, with one rightmost and one leftmost zero appended, are used as a 16-bit index limit. Bit 63 of the second-operand address is the operand-control bit; the operand-control bit is applicable when the first and third operands are unequal. When bit 63 is zero, the codeword is formed from the one's complement of the high operand; when bit 63 is one, the codeword is formed from the low operand. The remainder of the second-operand address is ignored.

General registers 1 and 3 contain the base addresses of the first and third operands. Bits 48-63 of general register 2 are used as an index for addressing both the first and third operands. General registers 1, 2, and 3 must all initially contain even values; otherwise, a specification exception is recognized.

In the access-register mode, access register 1 specifies the address space containing the first and third operands.

The size of the units by which the first and third operands are compared, the size of the resulting codeword, and the participation of bits 0-31 of general registers 1, 2, and 3 in the operation depend on the addressing mode. In the 24-bit or 31-bit addressing mode, the comparison unit is two bytes, the codeword is four bytes, and bits 0-31 of the registers are ignored and remain unchanged. In the 64-bit addressing mode, the comparison unit is six bytes, the codeword is eight bytes, and bits 0-31 of the registers are used in and may be changed by the operation.

Operation in the 24-Bit or 31-Bit Addressing Mode

The operation consists in comparing the first and third operands halfword by halfword and incrementing the index until an unequal pair of halfwords is found or the index exceeds the index limit. This proceeds in units of operation, between which interruptions may occur.

At the start of a unit of operation, the index, bits 48-63 of general register 2, is logically compared with the index limit. If the index is larger, the instruction is completed by placing bits 32-63 of general register 3, with bit 32 set to one, in bit positions 32-63 of general register 2, and by setting condition code 0.

If the index is less than or equal to the index limit, the index is applied to the first-operand and third-oper-

and base addresses to locate the current pair of halfwords to be compared. The index, with 48 leftmost zeros appended, and bits 32-63 of general register 1, with 32 leftmost zeros appended, are added to form a 64-bit intermediate value. A carry out of bit position 32, if any, is ignored. The address of the current first-operand halfword is generated from the intermediate value by following the normal rules for operand address generation. The address of the current third-operand halfword is formed in the same manner by adding bits 32-63 of general register 3 and the index.

The current first-operand and third-operand halfwords are logically compared. If they are equal, the contents of general register 2 are incremented by 2, and a unit of operation ends.

If the compare values are unequal, the contents of general register 2 are incremented by 2 and then shifted left logically by 16 bit positions. The shifting occurs only within bit positions 32-63. If the operand-control bit is zero, (1) the one's complement of the higher halfword is placed in bit positions 48-63 of general register 2, and (2) if operand 1 was higher, bits 32-63 of general registers 1 and 3 are interchanged. If the operand-control bit is one, (1) the lower halfword is placed in bit positions 48-63 of general register 2, and (2) if operand 1 was lower, bits 32-63 of general registers 1 and 3 are interchanged. Condition code 2 is set if general registers 1 and 3 are interchanged; otherwise, condition code 1 is set.

For the purpose of recognizing access exceptions, operand 1 and operand 3 are both considered to have a length equal to 2 more than the value of the index limit minus the index.

Operation in the 64-bit Addressing Mode

The operation consists in comparing the first and third operands in units of six bytes at a time and incrementing the index until an unequal pair of six-byte units is found or the index exceeds the index limit. This proceeds in units of operation, between which interruptions may occur.

At the start of a unit of operation, the index, bits 48-63 of general register 2, is logically compared with the index limit. If the index is larger, the instruction is completed by placing bits 0-63 of general register 3, with bit 0 set to one, in bit positions 0-63 of general register 2, and by setting condition code 0.

If the index is less than or equal to the index limit, the index is applied to the first-operand and third-operand base addresses to locate the current pair of six-byte units to be compared. The index, with 48 leftmost zeros appended, and bits 0-63 of general register 1 are added to form the 64-bit address of the current first-operand six-byte unit. A carry out of bit position 0, if any, is ignored. The address of the current third-operand six-byte unit is formed in the same manner by adding bits 0-63 of general register 3 and the index.

The current first-operand and third-operand six-byte units are logically compared. If they are equal, the contents of general register 2 are incremented by 6, and a unit of operation ends.

If the compare values are unequal, the contents of general register 2 are incremented by 6 and then shifted left logically by 48 bit positions. If the operand-control bit is zero, (1) the one's complement of the higher six-byte unit is placed in bit positions 16-63 of general register 2, and (2) if operand 1 was higher, bits 0-63 of general registers 1 and 3 are interchanged. If the operand-control bit is one, (1) the lower six-byte unit is placed in bit positions 16-63 of general register 2, and (2) if operand 1 was lower, bits 0-63 of general registers 1 and 3 are interchanged. Condition code 2 is set if general registers 1 and 3 are interchanged; otherwise, condition code 1 is set.

For the purpose of recognizing access exceptions, operand 1 and operand 3 are both considered to have a length equal to 6 more than the value of the index limit minus the index. However, depending on the model, access exceptions may or may not be recognized for the portion of a storage operand to the right of the first unequal byte when the operand is not needed to form the codeword in general register 2.

Specifications Independent of Addressing Mode

The condition code is unpredictable if the instruction is interrupted.

When the index is initially larger than the index limit, access exceptions are not recognized for the storage operands. For operands longer than 4K bytes, access exceptions are not recognized more than 4K bytes beyond the byte being processed. Access exceptions are not recognized when a specification-exception condition exists.

If the B_2 field designates general register 2, it is unpredictable whether or not the index limit is recomputed; thus, in this case the operand length is unpredictable. However, in no case can the operands exceed 2^{15} bytes in length.

Resulting Condition Code:

0 Operands equal
1 Operand-control bit zero and operand 1 low, or operand-control bit one and operand 3 low
2 Operand-control bit zero and operand 1 high, or operand-control bit one and operand 3 high
3 --

Program Exceptions:

- Access (fetch, operands 1 and 3)
- Specification
- Transaction constraint

Programming Notes:

1. An example of the use of COMPARE AND FORM CODEWORD is given in “Sorting Instructions” in Appendix A, “Number Representation and Instruction-Use Examples.”

2. The offset of the halfword or six-byte unit (depending on the addressing mode) of the first and third operands at which comparison is to begin should be placed in bit positions 48-63 of general register 2 before executing COMPARE AND FORM CODEWORD. The index limit derived from the second-operand address should be the offset of the last halfword or six-byte unit of the first and third operands for which comparison can be made. When the operands do not compare equal, the leftmost 16 bits of the codeword formed in general register 2 (bits 32-47 of the register in the 24-bit or 31-bit addressing mode, or bits 0-15 in the 64-bit addressing mode) by the execution of COMPARE AND FORM CODEWORD gives the offset of the first halfword or six-byte unit not compared. If the codewords compare equal in an UPDATE TREE operation, bit positions 32-47 of general register 2 in the 24-bit or 31-bit addressing mode, or bit positions 0-15 in the 64-bit addressing mode, will contain the offset at which another COMPARE AND FORM CODEWORD should resume comparison for breaking codeword ties. Operand-control-bit values of zero or one are used for sorting operands in ascending or descending order, respectively. Refer to “Sorting Instructions” on page A-53 for a discussion of the use of codewords in sorting.

3. The condition code indicates the results of comparing operands up to 32,768 bytes long. Equal operands result in a negative codeword in bit positions 32-63 of general register 2 in the 24-bit or 31-bit addressing mode, or in bit positions 0-63 in the 64-bit addressing mode. A negative codeword also results in the 24-bit or 31-bit mode when the index limit is 32,766 and the operands that are compared differ in only their last two bytes, or in the 64-bit mode when the limit is 32,762 and the operands differ in only their last six bytes. If this latter codeword is used by UPDATE TREE, an incorrect result may be indicated in general registers 0 and 1. Therefore, the index limit should not exceed 32,764 in the 24-bit or 31-bit mode, or 32,760 in the 64-bit mode, when the resulting codeword is to be used by UPDATE TREE.

4. Special precautions should be taken if COMPARE AND FORM CODEWORD is made the target of an execute-type instruction. See the programming note concerning interruptible instructions under EXECUTE.

5. Further programming notes concerning interruptible instructions are included in “Interruptible Instructions” in Chapter 5, “Program Execution.”

6. The storage-operand references of COMPARE AND FORM CODEWORD may be multiple-access references. (See “Storage-Operand Consistency”.)

7. Figure 7-191 on page 7-140 and Figure 7-192 on page 7-141 contain summaries of the operation in the 24-bit or 31-bit addressing mode, and Figure 7-193 on page 7-142 and Figure 7-194 on page 7-143 contain summaries of the operation in the 64-bit addressing mode.

Operand-Control Bit	Relation	Resulting Condition Code	Result in GR2 (Bits 32-63)	Result in GR1 (Bits 32-63)	Result in GR3 (Bits 32-63)
0	op1 = op3	0	OGR3b1	–	–
0	op1 < op3	1	X, nop3	–	–
0	op1 > op3	2	X, nop1	OGR3	OGR1
1	op1 = op3	0	OGR3b1	–	–
1	op1 < op3	2	X, top1	OGR3	OGR1
1	op1 > op3	1	X, top3	–	–

Explanation:

– The bits remain unchanged.
OGR1 The original value of GR1 bits 32-63.
OGR3 The original value of GR3 bits 32-63.
OBR3b1 The original value of GR3 bits 32-63 with bit 32 set to one.
X Bits 32-47 of GR2 are 2 more than the index of the first unequal halfword.
nop1 Bits 48-63 of GR2 are the one's complement of the first unequal halfword in operand 1.
nop3 Bits 48-63 of GR2 are the one's complement of the first unequal halfword in operand 3.
top1 Bits 48-63 of GR2 are the first unequal halfword in operand 1.
top3 Bits 48-63 of GR2 are the first unequal halfword in operand 3.

Figure 7-191. Operation of COMPARE AND FORM CODEWORD in the 24-Bit or 31-bit Addressing Mode

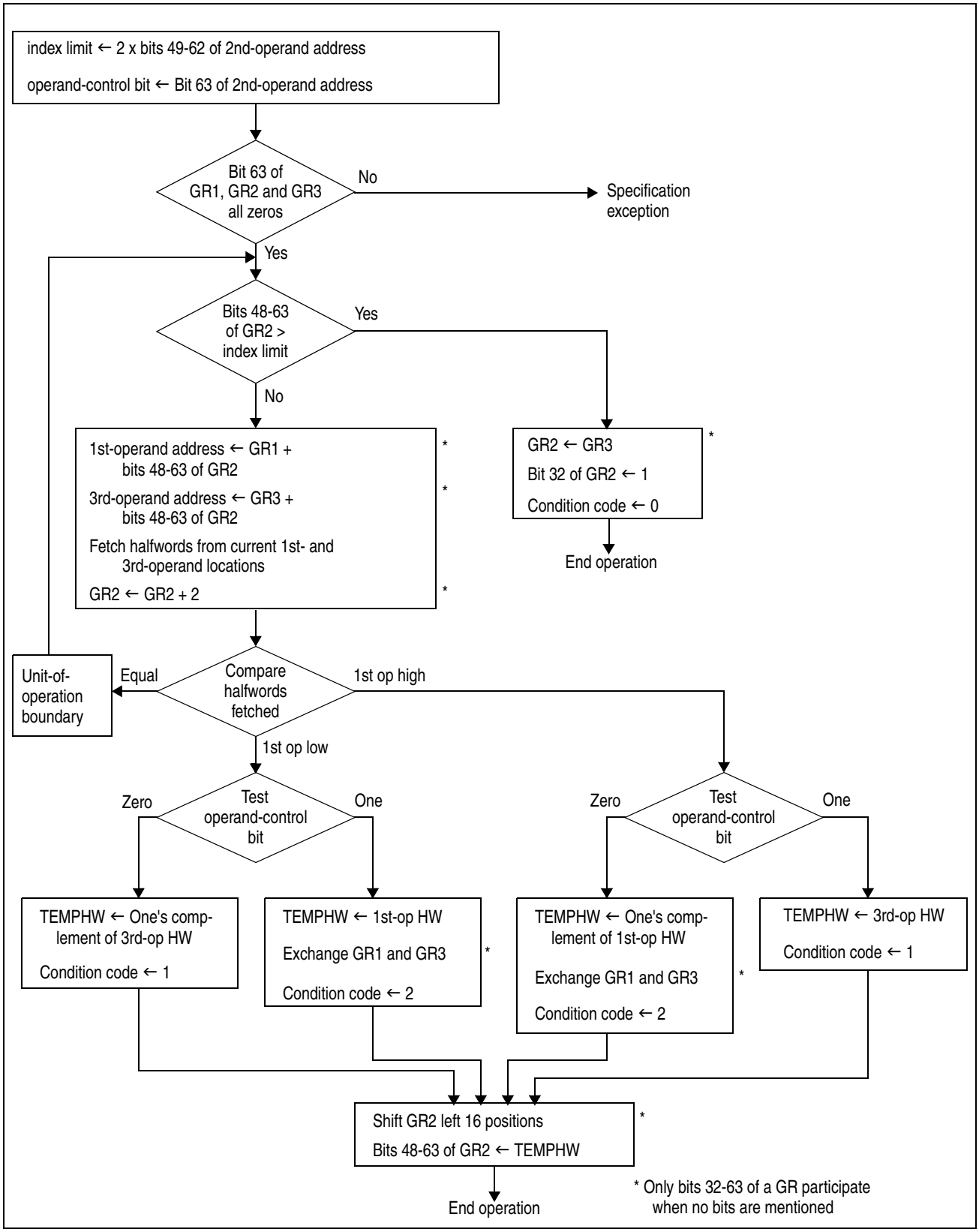


Figure 7-192. Execution of COMPARE AND FORM CODEWORD in the 24-Bit or 31-bit Addressing Mode

Operand-Control Bit	Relation	Resulting Condition Code	Result in GR2 (Bits 0-63)	Result in GR1 (Bits 0-63)	Result in GR3 (Bits 0-63)
0	op1 = op3	0	OGR3b1	–	–
0	op1 < op3	1	X, nop3	–	–
0	op1 > op3	2	X, nop1	OGR3	OGR1
1	op1 = op3	0	OGR3b1	–	–
1	op1 < op3	2	X, top1	OGR3	OGR1
1	op1 > op3	1	X, top3	–	–

Explanation:

– The bits remain unchanged.
OGR1 The original value of GR1 bits 0-63.
OGR3 The original value of GR3 bits 0-63.
OBR3b1 The original value of GR3 bits 0-63 with bit 0 set to one.
X Bits 0-15 of GR2 are 6 more than the index of the first unequal six-byte unit.
nop1 Bits 16-63 of GR2 are the one's complement of the first unequal six-byte unit in operand 1.
nop3 Bits 16-63 of GR2 are the one's complement of the first unequal six-byte unit in operand 3.
top1 Bits 16-63 of GR2 are the first unequal six-byte unit in operand 1.
top3 Bits 16-63 of GR2 are the first unequal six-byte unit in operand 3.

Figure 7-193. Operation of COMPARE AND FORM CODEWORD in the 64-bit Addressing Mode

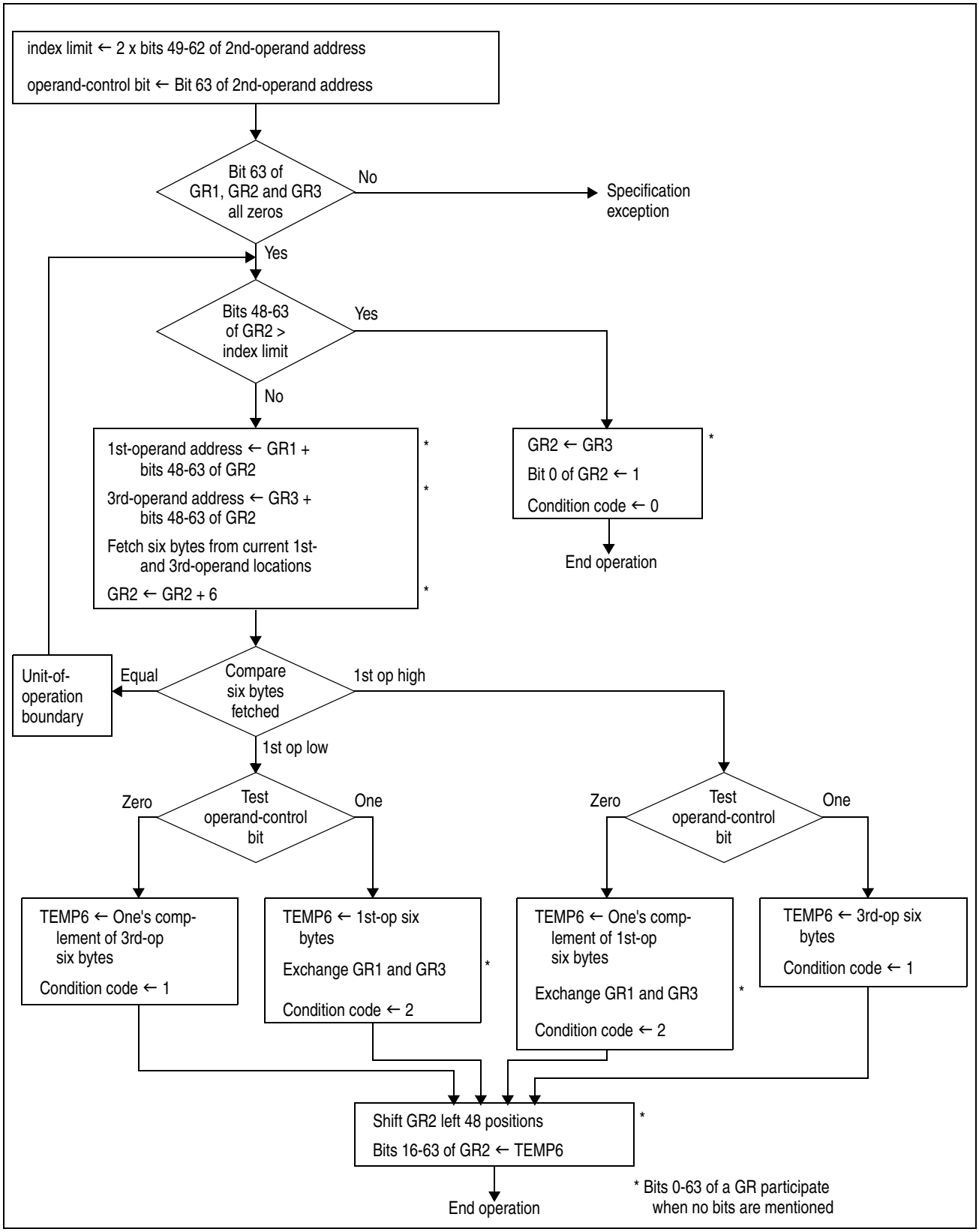
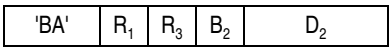


Figure 7-194. Execution of COMPARE AND FORM CODEWORD in the 64-bit Addressing Mode

COMPARE AND SWAP

CS $R_1,R_3,D_2(B_2)$ [RS-a]

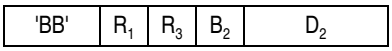


CSY $R_1,R_3,D_2(B_2)$ [RSY-a]

CSG $R_1,R_3,D_2(B_2)$ [RSY-a]

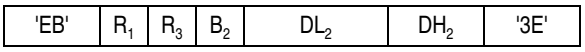
COMPARE DOUBLE AND SWAP

CDS $R_1,R_3,D_2(B_2)$ [RS-a]



CDSY $R_1,R_3,D_2(B_2)$ [RSY-a]

CDSG $R_1,R_3,D_2(B_2)$ [RSY-a]



The first and second operands are compared. If they are equal, the third operand is stored at the second-operand location. If they are unequal, the second operand is loaded into the first-operand location. The result of the comparison is indicated in the condition code.

For COMPARE AND SWAP (CS, CSY), the first and third operands are 32 bits in length, with each operand occupying bit positions 32-63 of a general register. The second operand is a word in storage.

For COMPARE AND SWAP (CSG), the first and third operands are 64 bits in length, with each operand occupying bit positions 0-63 of a general register. The second operand is a doubleword in storage.

For COMPARE DOUBLE AND SWAP (CDS, CDSY), the first and third operands are 64 bits in length. The first 32 bits of an operand occupy bit positions 32-63 of the even-numbered register of an even-odd pair of general registers, and the second 32 bits occupy bit positions 32-63 of the odd-numbered register of the pair. The second operand is a doubleword in storage.

For COMPARE DOUBLE AND SWAP (CDSG), the first and third operands are 128 bits in length. The first 64 bits of an operand occupy bit positions 0-63 of the even-numbered register of an even-odd pair of general registers, and the second 64 bits occupy bit positions 0-63 of the odd-numbered register of the pair. The second operand is a quadword in storage.

When an equal comparison occurs, the third operand is stored at the second-operand location. The fetch of the second operand for purposes of comparison and the store into the second-operand location appear to be a block-concurrent interlocked-update reference as observed by other CPUs.

When the result of the comparison is unequal, the second operand is loaded at the first-operand location, and the second-operand location remains unchanged. However, on some models, the contents may be fetched and subsequently stored back unchanged at the second-operand location. This update appears to be a block-concurrent interlocked-update reference as observed by other CPUs.

A serialization function is performed before the operand is fetched and again after the operation is completed.

The displacement for CS and CDS is treated as a 12-bit unsigned binary integer. The displacement for CSY, CSG, CDSY, and CDSG is treated as a 20-bit signed binary integer.

The second operand of COMPARE AND SWAP (CS, CSY) must be designated on a word boundary. The second operand of COMPARE AND SWAP (CSG) and COMPARE DOUBLE AND SWAP (CDS, CDSY) must be designated on a doubleword boundary. The second operand of COMPARE DOUBLE AND SWAP (CDSG) must be designated on a quadword boundary. The R_1 and R_3 fields for COMPARE DOUBLE AND SWAP must each designate an even-numbered register. Otherwise, a specification exception is recognized.

Resulting Condition Code:

0 First and second operands equal, second operand replaced by third operand
1 First and second operands unequal, first operand replaced by second operand
2 --
3 --

Program Exceptions:

- Access (fetch and store, operand 2)
- Operation (CSY and CDSY, if the long-displacement facility is not installed)
- Specification
- Transaction constraint

Programming Notes:

1. Several examples of the use of the COMPARE AND SWAP and COMPARE DOUBLE AND SWAP instructions are given in Appendix A, "Number Representation and Instruction-Use Examples."

2. Some of the following notes are worded, with respect to operand size, for CS, CSY, CDS, and CDSY. Similar notes, worded for a larger operand size, would apply to CSG and CDSG.

3. COMPARE AND SWAP can be used by CPU programs sharing common storage areas in either a multiprogramming or multiprocessing environment. Two examples are:

   a. By performing the following procedure, a CPU program can modify the contents of a storage location even though the possibility exists that the CPU program may be interrupted by another CPU program that will update the location or that another CPU program may simultaneously update the location. First, the entire word containing the byte or bytes to be updated is loaded into a general register. Next, the updated value is computed and placed in another general register. Then COMPARE AND SWAP is executed with the R_1 field designating the register that contains the original value and the R_3 field designating the register that contains the updated value. If the update has been successful, condition code 0 is set. If the storage location no longer contains the original value, the update has not been successful, the general register designated by the R_1 field of the COMPARE AND SWAP instruction contains the new current value of the storage location, and condition code 1 is set. When condition code 1 is set, the CPU program can repeat the procedure using the new current value.

   b. COMPARE AND SWAP can be used for controlled sharing of a common storage area, including the capability of leaving a message (in a chained list of messages) when the common area is in use. To accomplish this, a word in storage can be used as a control word, with a zero value in the word indicating that the common area is not in use and that no messages exist, a negative value indicating that the area is in use and that no messages exist, and a nonzero positive value indicating that the common area is in use and that the value is the address of the most recent message added to the list. Thus, any number of CPU programs desiring to seize the area can use COMPARE AND SWAP to update the control word to indicate that the area is in use or to add messages to the list. The single CPU program which has seized the area can also safely use COMPARE AND SWAP to remove messages from the list.

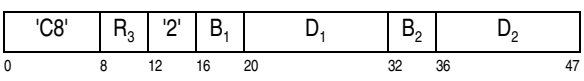
4. COMPARE DOUBLE AND SWAP can be used in a manner similar to that described for COMPARE AND SWAP. In addition, it has another use. Consider a chained list, with a control word used to address the first message in the list, as described in programming note 3.b above. If multiple CPU programs are to be permitted to delete messages by using COMPARE AND SWAP (and not just the single CPU program which has seized the common area), there is a possibility the list will be incorrectly updated. This would occur if, for example, after one CPU program has fetched the address of the most recent message in order to remove the message, another CPU program removes the first two messages and then adds the first message back into the chain. The first CPU program, on continuing, cannot easily detect that the list is changed. By increasing the size of the control word to a doubleword containing both the first message address and a word with a change number that is incremented for each modification of the list, and by using COMPARE DOUBLE AND SWAP to update both fields together, the possibility of the list being

incorrectly updated is reduced to a negligible level. That is, an incorrect update can occur only if the first CPU program is delayed while changes exactly equal in number to a multiple of 2^{32} take place *and* only if the last change places the original message address in the control word.

5. To ensure successful updating of a common storage field by two or more CPUs, all updates must be done by means of an interlocked-update reference (the section "Interlocked-Update References" on page 5-124 lists the instructions that perform an interlocked-update reference). For example, in a configuration where the interlocked-access facility 2 is not installed, if one CPU executes OR IMMEDIATE and another CPU executes COMPARE AND SWAP to update the same byte, the fetch by OR IMMEDIATE may occur either before the fetch by COMPARE AND SWAP or between the fetch and the store by COMPARE AND SWAP, and then the store by OR IMMEDIATE may occur after the store by COMPARE AND SWAP, in which case the change made by COMPARE AND SWAP is lost.
6. For the case of a condition-code setting of 1, COMPARE AND SWAP and COMPARE DOUBLE AND SWAP may or may not, depending on the model, cause any of the following to occur for the second-operand location: a PER storage-alteration event may be recognized; a protection exception for storing may be recognized; and, provided no access exception exists, the change bit may be set to one. Because the contents of storage remain unchanged, the change bit may or may not be one when a PER storage-alteration event is recognized.
7. The performance of CDSG on some models may be significantly slower than that of CSG. When quadword consistency is not required by the program, alternate code sequences should be used.

COMPARE AND SWAP AND STORE

CSST $D_1(B_1),D_2(B_2),R_3$ [SSF]



The first operand is compared with the third operand. If they are equal, the replacement value is stored into the first-operand location, and subsequently, the store value is placed into the second-operand location. If they are not equal, then the first operand is loaded into the third-operand location, and the second operand is not changed. The result is indicated by the condition code.

The size of the first and third operands, the size of the replacement value, and the alignment of the first operand are determined by the contents of a function code (FC) in bits 56-63 of general register 0. The assigned function codes are as follows:

- When the function code is 0, the operands are 32 bits in length. The first operand is a word in storage, and the third operand is in bits 32-63 of general register R_3. The replacement value is in bytes 0-3 of the parameter list.
- When the function code is 1, the operands are 64 bits in length. The first operand is a doubleword in storage, and the third operand is in bits 0-63 of general register R_3. The replacement value is bytes 0-7 of the parameter list.
- When the function code is 2 and the compare-and-swap-and-store facility 2 is installed, the operands are 128 bits in length. The first operand is a quadword in storage, and the third operand is in bits 0-63 of general registers R_3 and $R_3 + 1$. The replacement value is in bytes 0-15 of the parameter list.

The size of the store value and the alignment of the second operand are determined by a store characteristic (SC) in bits 48-55 of general register 0. The store characteristic is expressed as a power of two; the assigned store characteristics are 0, 1, 2, and 3. An SC of 0 means that one byte is stored on a byte boundary; an SC of 1 means that two bytes are stored on a halfword boundary, an SC of 2 means that four bytes are stored on a word boundary, and an SC of 3 means that eight bytes are stored on a doubleword boundary. When the CSST facility 2 is installed, an SC of 4 is also assigned; an SC of 4 means that sixteen bytes are stored on a quadword boundary. The store value begins at byte 16 of the parameter list.

Bits 0-31 of general register 0 are ignored. Bits 32-47 of general register 0 and bits 60-63 of general register 1 are reserved and should contain zeros; otherwise, the program may not operate compatibly in the future.

General register 1 contains the logical address of a parameter list that contains the replacement value and the store value. The parameter list is on a quadword boundary. In the access-register mode, access register 1 specifies the address space containing the parameter list.

The handling of the parameter-list address is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-59 of general register 1, with four zeros appended to the right, constitute the address, and bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-59 of the register, with four zeros appended to the right, constitute the address, and bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-59 of the register, with four zeros appended to the right, constitute the address.

Figure 7-195 illustrates contents of the registers and parameter list just described.

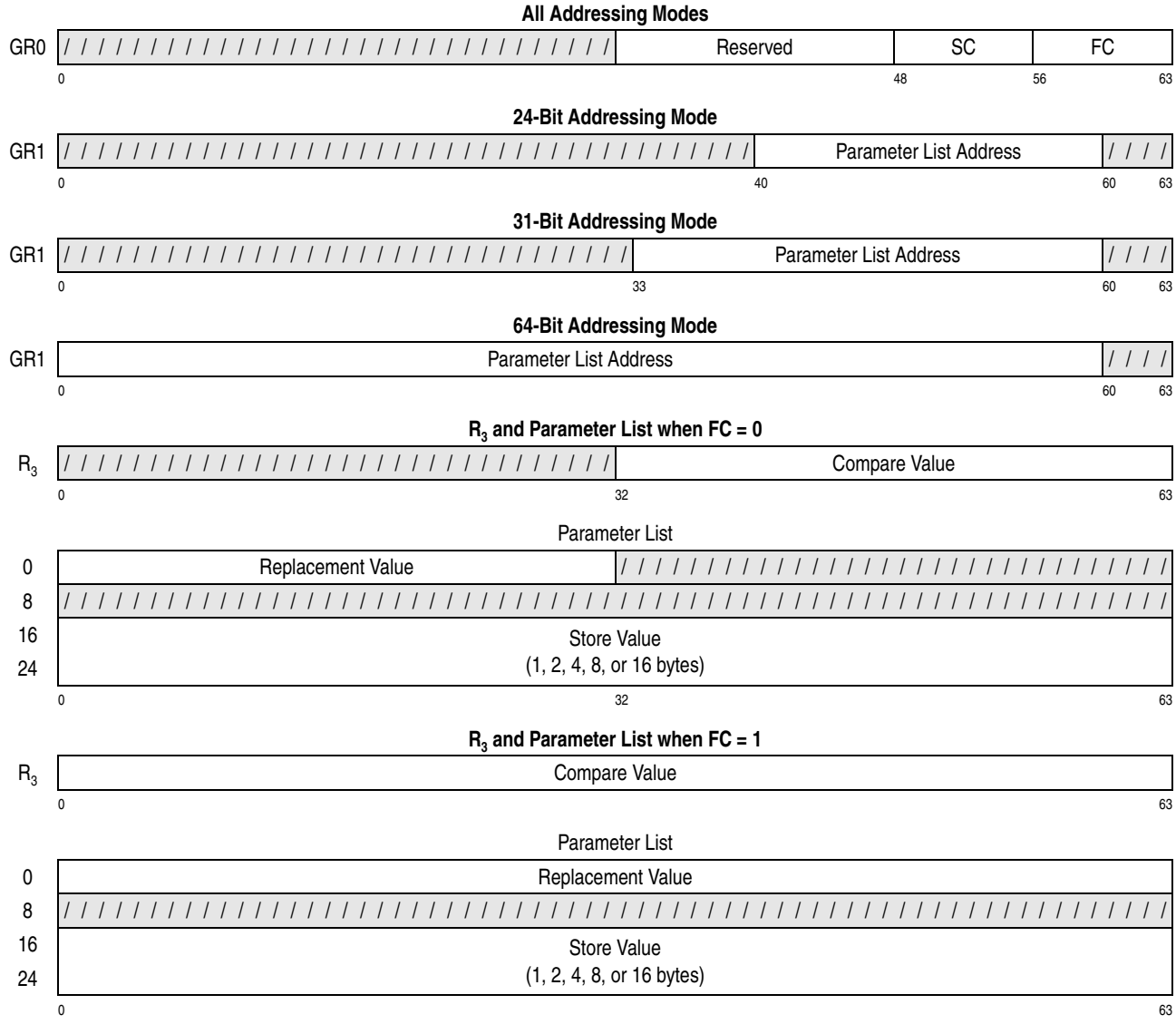


Figure 7-195. Register Contents for COMPARE AND SWAP AND STORE

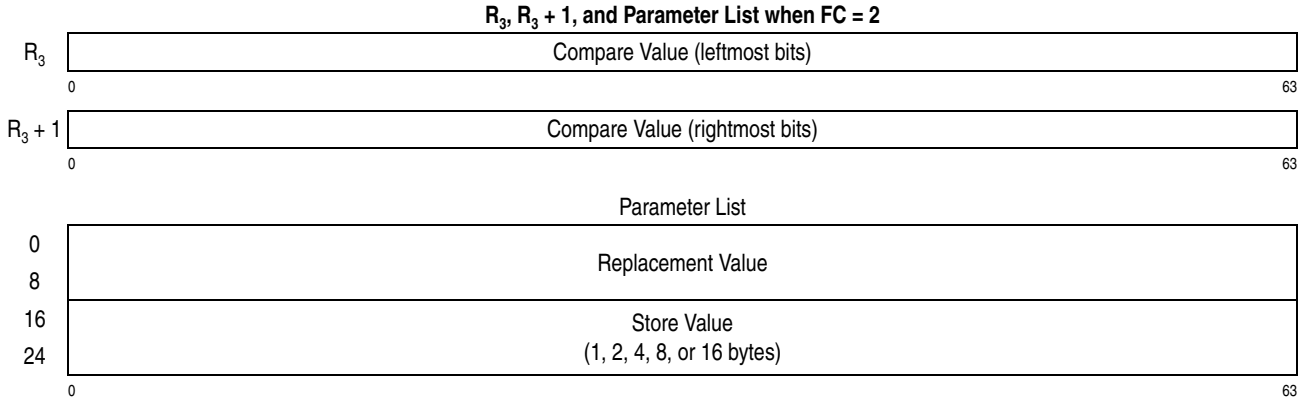


Figure 7-195. Register Contents for COMPARE AND SWAP AND STORE

When an equal comparison occurs, the replacement value is stored at the first-operand location, and the store value is stored at the second-operand location. The fetch of the first operand for purposes of comparison, and the store of the replacement value into the first-operand location, both appear to be a block-concurrent interlocked-update reference as observed by other CPUs. The store of the store value appears to be block-concurrent as observed by other CPUs.

When the result of the comparison is unequal, the first operand is loaded into the third-operand location, and the first operand remains unchanged. However, on some models, the contents of the first operand may be fetched and subsequently stored back unchanged at the first-operand location. This update appears to be a block-concurrent interlocked-update reference as observed by other CPUs.

As observed by this CPU and by other CPUs, all fetches appear to occur before all stores, and the store into the first operand appears to occur before the store into the second operand. Access-exception conditions are recognized for the entire 32-byte parameter list and for the second operand, regardless of whether the first and third operands are or are not equal. For the second operand, PER storage-alteration and zero-address-detection events are recognized, and a change bit is set, only if a store occurs.

A serialization function is performed before the operation begins and again after the operation is completed.

Special Conditions

A specification exception is recognized for any of the following conditions:

- The function code specifies an unassigned value.
- The store characteristic specifies an unassigned value.
- The function code is 0, and the first operand is not designated on a word boundary.
- The function code is 1, and the first operand is not designated on a doubleword boundary.
- The function code is 2, and any of the following is true:
  - The compare-and-swap-and-store facility 2 is not installed.
  - The first operand is not designated on a quadword boundary.
  - The R_3 field does not designate the even-numbered register of an even-odd register pair.
- The second operand is not designated on an integral boundary corresponding to the size of the store value.

For all of the above conditions, and for all addressing and protection exceptions, the operation is suppressed.

Resulting Condition Code:

0 First and third operands equal; first operand replaced by replacement value, and second operand replaced by store value
1 First and third operands unequal; third operand replaced by first operand
2 --
3 --

Program Exceptions:

- Access (fetch, parameter list; fetch and store, operand 1; store, operand 2)
- Operation (if the compare-and-swap-and-store facility is not installed)
- Specification
- Transaction constraint

Programming Notes:

1. COMPARE AND SWAP AND STORE may be used in conjunction with other COMPARE AND SWAP or COMPARE DOUBLE AND SWAP instructions to manipulate locks, queue pointers, or other fields that require interlocked updates.

2. COMPARE AND SWAP AND STORE should not be used to manipulate fields that are also manipulated by PERFORM LOCKED OPERATION.

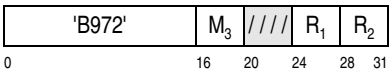
3. The store value is intended to provide a separate "footprint" of the interlocked-update operation in a location apart from the first operand in a single unit of operation.

4. The performance of COMPARE AND SWAP AND STORE may be significantly slower than the that of separate COMPARE AND SWAP, BRANCH ON CONDITION, and STORE instructions.

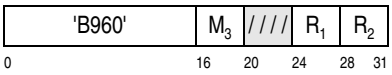
5. COMPARE AND SWAP AND STORE should only be used when an interruption between the compare-and-swap operation and the store operation cannot be tolerated, and other means of disabling for interruptions are not practical.

COMPARE AND TRAP

CRT R_1,R_2,M_3 [RRF-c]

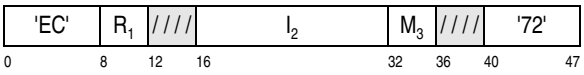


CGRT R_1,R_2,M_3 [RRF-c]

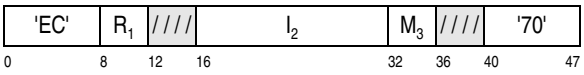


COMPARE IMMEDIATE AND TRAP

CIT R_1,I_2,M_3 [RIE-a]



CGIT R_1,I_2,M_3 [RIE-a]



The first operand is compared with the second operand. If the mask bit in the M_3 field corresponding to the comparison result is one, a compare-and-trap-instruction data exception is recognized, and the operation is completed; otherwise, normal instruction sequencing proceeds with the updated instruction address.

The comparison results and corresponding M_3 bits are as follows:

Comparison Result	M_3 Bit
Equal	0
First operand low	1
First operand high	2

Bit 3 of the M_3 field is reserved and should be zero; otherwise, the program may not operate compatibly in the future.

For COMPARE AND TRAP (CRT) and COMPARE IMMEDIATE AND TRAP (CIT), the first operand is treated as a 32-bit signed binary integer. For COMPARE AND TRAP (CGRT) and COMPARE IMMEDIATE AND TRAP (CGIT), the first operand is treated as a 64-bit signed binary integer.

For COMPARE IMMEDIATE AND TRAP, the second operand is treated as a 16-bit signed binary integer.

For COMPARE AND TRAP (CRT), the second operand is treated as a 32-bit signed binary integer. For COMPARE AND TRAP (CGRT), the second operand is treated as a 64-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

- Data
- Operation (if the general-instructions-extension facility is not installed)

Programming Notes:

1. Possible uses of COMPARE AND TRAP and COMPARE IMMEDIATE AND TRAP include checking for pointers containing zero (null pointer), or array range checking.

2. On most models, when a data exception is not recognized, the performance of COMPARE AND TRAP and COMPARE IMMEDIATE AND TRAP is typically similar to other COMPARE instructions. When a data exception is recognized, the instruction may be significantly slower than an equivalent COMPARE instruction followed by a BRANCH ON CONDITION instruction.

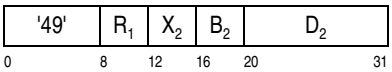
3. When bit positions 0-2 of the M_3 field contain zeros, the instruction acts as a NOP, however this is not the preferred instruction with which to create a NOP. When bit positions 0-2 of the M_3 field contain 111 binary, a data exception is always recognized.

4. When a data exception is recognized, the compare-and-trap data-exception code (DXC FF hex) is stored, as described in “Data Exception” on page 6-25.

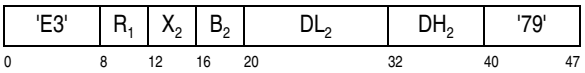
5. The discussion of extended mnemonics in programming note 3 for COMPARE AND BRANCH on page 7-137 also applies to the compare-and-trap instructions.

COMPARE HALFWORD

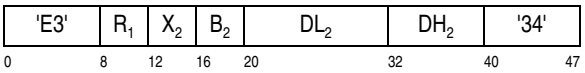
CH $R_1,D_2(X_2,B_2)$ [RX-a]



CHY $R_1,D_2(X_2,B_2)$ [RXY-a]



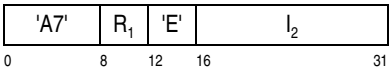
CGH $R_1,D_2(X_2,B_2)$ [RXY-a]



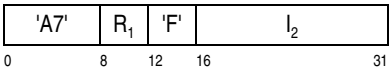
COMPARE HALFWORD IMMEDIATE

Register-and-immediate formats:

CHI R_1,I_2 [RI-a]

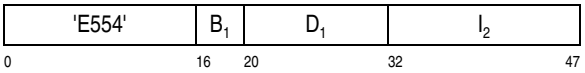


CGHI R_1,I_2 [RI-a]

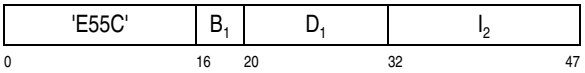


Storage-and-immediate formats:

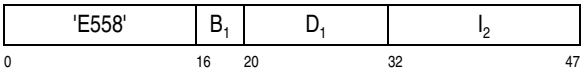
CHHSI $D_1(B_1),I_2$ [SIL]



CHSI $D_1(B_1),I_2$ [SIL]

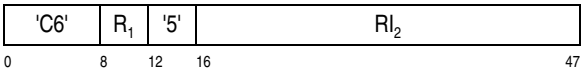


CGHSI $D_1(B_1),I_2$ [SIL]

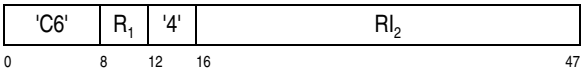


COMPARE HALFWORD RELATIVE LONG

CHRL R_1,RI_2 [RIL-b]



CGHRL R_1,RI_2 [RIL-b]



The first operand is compared with the second operand, and the result is indicated in the condition code. The second operand is two bytes in length and is treated as a 16-bit signed binary integer.

For COMPARE HALFWORD IMMEDIATE (CHHSI), the first operand is treated as a 16-bit signed binary integer. For COMPARE HALFWORD (CH, CHY), COMPARE HALFWORD IMMEDIATE (CHI and CHSI), and COMPARE HALFWORD RELATIVE LONG (CHRL), the first operand is treated as a 32-bit signed binary integer. For COMPARE HALFWORD (CGH), COMPARE HALFWORD IMMEDIATE (CGHI and CGHSI), and COMPARE HALFWORD RELATIVE LONG (CGHRL), the first operand is treated as a 64-bit signed binary integer.

The displacement for COMPARE HALFWORD (CH) and for COMPARE HALFWORD IMMEDIATE (CGHSI, CHHSI, CHSI) is treated as a 12-bit unsigned binary integer. The displacement for COMPARE HALFWORD (CHY, CGH) is treated as a 20-bit signed binary integer.

For COMPARE HALFWORD RELATIVE LONG, the contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the address of the second operand in storage. When DAT is on, the second operand is accessed using the same addressing-space mode as that used to access the instruction. When DAT is off, the second operand is accessed using a real address.

Resulting Condition Code:

0 Operands equal
1 First operand low
2 First operand high
3 --

Program Exceptions:

- Access (fetch, operand 1 of CGHSI, CHHSI, and CHSI only; fetch, operand 2 of CGH, CGHRL, CH, CHRL, CHY only)
- Operation (CHY, if the long-displacement facility is not installed; CGH, CHRL, CGHRL, CGHSI, CHHSI, and CHSI, if the general-instructions-extension facility is not installed)

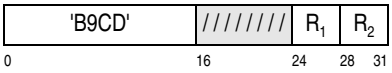
Programming Notes:

1. An example of the use of the COMPARE HALFWORD instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. For COMPARE HALFWORD RELATIVE LONG, the second operand is necessarily aligned on an integral boundary corresponding to the operand’s size.
3. When COMPARE HALFWORD RELATIVE LONG is the target of an execute-type instruction, the second-operand address is relative to the target address.

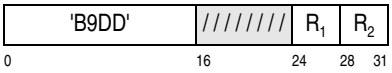
COMPARE HIGH

Register-and-register formats:

CHHR R_1,R_2 [RRE]

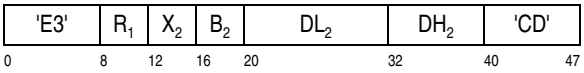


CHLR R_1,R_2 [RRE]



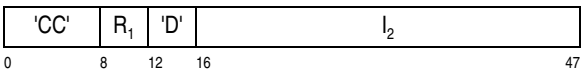
Register-and-storage format:

CHF $R_1,D_2(X_2,B_2)$ [RXY-a]



COMPARE IMMEDIATE HIGH

CIH R_1,I_2 [RIL-a]



The first operand is compared with the second operand, and the result is indicated in the condition code.

The operands are treated as 32-bit signed binary integers. The first operand is in bit positions 0-31 of general register R_1; bit positions 32-63 of the register are ignored. For COMPARE HIGH (CHHR), the second operand is in bit positions 0-31 of general register R_2; bit positions 32-63 of the register are ignored.

For COMPARE HIGH (CHLR), the second operand is in bit positions 32-63 of general register R_2; bit positions 0-31 of the register are ignored.

The displacement for CHF is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Operands equal
1 First operand low
2 First operand high
3 --

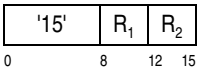
Program Exceptions:

- Access (operand 2 of CHF only)
- Operation (if the high-word facility is not installed)

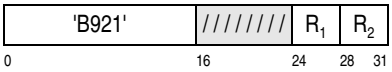
COMPARE LOGICAL

Register-and-register formats:

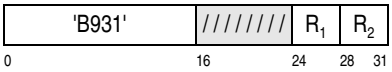
CLR R_1,R_2 [RR]



CLGR R_1,R_2 [RRE]

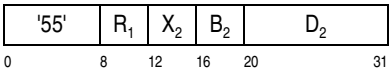


CLGFR R_1,R_2 [RRE]

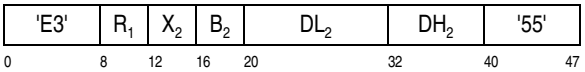


Register-and-storage formats:

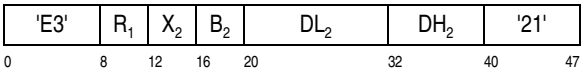
CL $R_1,D_2(X_2,B_2)$ [RX-a]



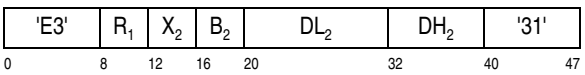
CLY $R_1,D_2(X_2,B_2)$ [RXY-a]



CLG $R_1,D_2(X_2,B_2)$ [RXY-a]

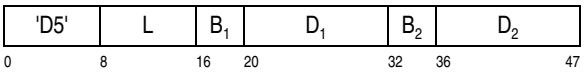


CLGF $R_1,D_2(X_2,B_2)$ [RXY-a]



Storage-and-storage format:

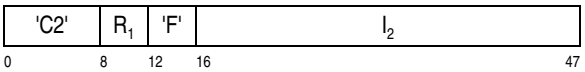
CLC $D_1(L,B_1),D_2(B_2)$ [SS-a]



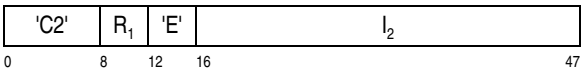
COMPARE LOGICAL IMMEDIATE

Register-and-immediate formats:

CLFI R_1,I_2 [RIL-a]

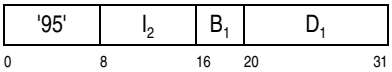


CLGFI R_1,I_2 [RIL-a]

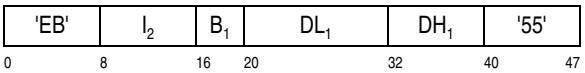


Storage-and-immediate formats:

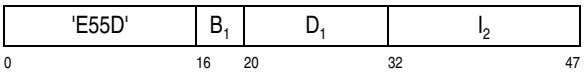
CLI $D_1(B_1),I_2$ [SI]



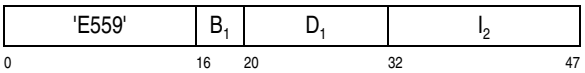
CLIY $D_1(B_1),I_2$ [SIY]



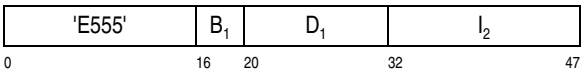
CLFHSI $D_1(B_1),I_2$ [SIL]



CLGHSI $D_1(B_1),I_2$ [SIL]

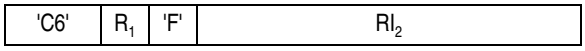


CLHHSI $D_1(B_1),I_2$ [SIL]

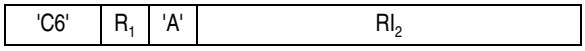


COMPARE LOGICAL RELATIVE LONG

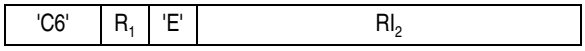
CLRL R_1,RI_2 [RIL-b]



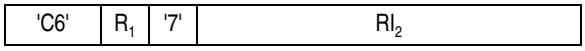
CLGRL R_1,RI_2 [RIL-b]



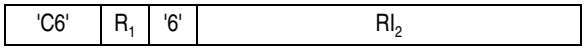
CLGFRL R_1,RI_2 [RIL-b]



CLHRL R_1,RI_2 [RIL-b]



CLGHRL R_1,RI_2 [RIL-b]



The first operand is compared with the second operand, and the result is indicated in the condition code.

The lengths of the operands are as follows:

- For COMPARE LOGICAL (CLR, CL, CLY), COMPARE LOGICAL IMMEDIATE (CLFI), and COMPARE LOGICAL RELATIVE LONG (CLRL), the operands are treated as 32 bits.
- For COMPARE LOGICAL (CLGR, CLG) and COMPARE LOGICAL RELATIVE LONG (CLGRL), the operands are treated as 64 bits.
- For COMPARE LOGICAL (CLGFR, CLGF), COMPARE LOGICAL IMMEDIATE (CLGFI), and COMPARE LOGICAL RELATIVE LONG (CLGFRL), the first operand is treated as 64 bits, and the second operand is treated as 32 bits with 32 zeros appended on the left.
- For COMPARE LOGICAL IMMEDIATE (CLHHSI), the operands are treated as 16 bits.
- For COMPARE LOGICAL IMMEDIATE (CLFHSI) and COMPARE LOGICAL RELATIVE LONG (CLHRL), the first operand is treated as 32 bits, and the second operand is treated as 16 bits with 16 zeros appended on the left.
- For COMPARE LOGICAL IMMEDIATE (CLGHSI) and COMPARE LOGICAL RELATIVE LONG (CLGHRL), the first operand is treated as 64 bits, and the second operand is treated as 16 bits with 48 zeros appended on the left.
- For COMPARE LOGICAL IMMEDIATE (CLI, CLIY), the operands are treated as 8 bits.
- For COMPARE LOGICAL (CLC), the first and second operands have the same length, in the range of one to 256 bytes.

The comparison proceeds left to right, byte by byte, and ends as soon as an inequality is found or the end of the fields is reached. For COMPARE LOGICAL (CL, CLY, CLG, CLGF, CLC), COMPARE LOGICAL IMMEDIATE (CLFHSI, CLGHSI, CLHHSI) and COMPARE LOGICAL RELATIVE LONG, access exceptions may or may not be recognized for the portion of a storage operand to the right of the first unequal byte.

The displacements for CL, CLI, CLFHSI, CLGHSI, CLHHSI, and both operands of CLC are treated as 12-bit unsigned binary integers. The displacement for CLY, CLG, CLGF, and CLIY is treated as a 20-bit signed binary integer.

For COMPARE LOGICAL RELATIVE LONG, the contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the address of the second operand in storage. When DAT is on, the second operand is accessed using the same addressing-space mode as that used to access the instruction. When DAT is off, the second operand is accessed using a real address.

For COMPARE LOGICAL RELATIVE LONG (CLRL, CLGFRL), the second operand must be aligned on a word boundary, and for COMPARE LOGICAL RELATIVE LONG (CLGRL), the second operand must be aligned on a doubleword boundary; otherwise, a specification exception is recognized.

Resulting Condition Code:

0 Operands equal
1 First operand low
2 First operand high
3 --

Program Exceptions:

- Access (fetch, operand 2, CL, CLC, CLG, CLGF, CLGFRL, CLGHRL, CLGRL, CLHRL, CLRL, and CLY; fetch, operand 1, CLC, CLFHSI, CLGHSI, CLHHSI, CLI, and CLIY)
- Operation (CLY and CLIY, if the long-displacement facility is not installed; CLFI and CLGFI, if the extended-immediate facility is not installed; CLFHSI, CLGFRL, CLGHSI, CLGHRL, CLGRL, CLHHSI, CLHRL, and CLRL, if the general-instructions-extension facility is not installed)
- Specification (CLRL, CLGFRL, CLGRL)
- Transaction constraint (CLC)

Programming Notes:

1. Examples of the use of the COMPARE LOGICAL instruction are given in Appendix A, "Number Representation and Instruction-Use Examples."
2. COMPARE LOGICAL treats all bits of each operand alike as part of a field of unstructured logical data. For COMPARE LOGICAL (CLC), the comparison may extend to field lengths of 256 bytes.
3. For COMPARE LOGICAL RELATIVE LONG, the second operand must be aligned on an integral boundary corresponding to the operand's size.
4. When COMPARE LOGICAL RELATIVE LONG is the target of an execute-type instruction, the second-operand address is relative to the target address.

COMPARE LOGICAL AND BRANCH

CLRB $R_1,R_2,M_3,D_4(B_4)$ [RRS]

'EC'	R_1	R_2	B_4	D_4	M_3	////	'F7'
0	8	12	16	20	32	36	40 47

CLGRB $R_1,R_2,M_3,D_4(B_4)$ [RRS]

'EC'	R_1	R_2	B_4	D_4	M_3	////	'E5'
0	8	12	16	20	32	36	40 47

COMPARE LOGICAL AND BRANCH RELATIVE

CLRJ R_1,R_2,M_3,RI_4 [RIE-b]

'EC'	R_1	R_2	RI_4	M_3	////	'77'
0	8	12	16	32	36	40 47

CLGRJ R_1,R_2,M_3,RI_4 [RIE-b]

'EC'	R_1	R_2	RI_4	M_3	////	'65'
0	8	12	16	32	36	40 47

COMPARE LOGICAL IMMEDIATE AND BRANCH

CLIB $R_1,I_2,M_3,D_4(B_4)$ [RIS]

'EC'	R_1	M_3	B_4	D_4	I_2	'FF'
0	8	12	16	20	32	40 47

CLGIB $R_1,I_2,M_3,D_4(B_4)$ [RIS]

'EC'	R_1	M_3	B_4	D_4	I_2	'FD'
0	8	12	16	20	32	40 47

COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE

CLIJ R_1,I_2,M_3,RI_4 [RIE-c]

'EC'	R_1	M_3	RI_4	I_2	'7F'
0	8	12	16	32	40 47

CLGIJ R_1,I_2,M_3,RI_4 [RIE-c]

'EC'	R_1	M_3	RI_4	I_2	'7D'
0	8	12	16	32	40 47

The first operand is compared with the second operand. If the mask bit in the M_3 field corresponding to the comparison result is one, the instruction address in the current PSW is replaced by the branch address specified by the fourth operand; otherwise, normal instruction sequencing proceeds with the updated instruction address.

For COMPARE LOGICAL AND BRANCH (CLRB), COMPARE LOGICAL AND BRANCH RELATIVE (CLRJ), COMPARE LOGICAL IMMEDIATE AND BRANCH (CLIB), and COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (CLIJ), the first

operand is treated as a 32-bit unsigned binary integer. For COMPARE LOGICAL AND BRANCH (CLGRB), COMPARE LOGICAL AND BRANCH RELATIVE (CLGRJ), COMPARE LOGICAL IMMEDIATE AND BRANCH (CLGIB), and COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (CLGIJ), the first operand is treated as a 64-bit unsigned binary integer.

For COMPARE LOGICAL AND BRANCH (CLRB) and COMPARE LOGICAL AND BRANCH RELATIVE (CLRJ), the second operand is treated as a 32-bit unsigned binary integer. For COMPARE LOGICAL AND BRANCH (CLGRB) and COMPARE LOGICAL AND BRANCH RELATIVE (CLGRJ), the second operand is treated as a 64-bit unsigned binary integer. For COMPARE LOGICAL IMMEDIATE AND BRANCH and COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE, the second operand is treated as an 8-bit unsigned binary integer. When the size of the second operand is smaller than the size of the first operand, the second operand is extended on the left with zeros to match the size of the first operand.

The comparison results and corresponding M_3 bits are as follows:

Comparison Result	M_3 Bit
Equal	0
First operand low	1
First operand high	2

Bit 3 of the M_3 field is reserved and should be zero; otherwise, the program may not operate compatibly in the future.

For COMPARE LOGICAL AND BRANCH and COMPARE LOGICAL IMMEDIATE AND BRANCH, the fourth operand is used as the branch address. For COMPARE LOGICAL AND BRANCH RELATIVE and COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE, the contents of the RI_4 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the branch address.

Condition Code: The code remains unchanged.

Program Exceptions:

- Operation (if the general-instructions-extension facility is not installed).
- Transaction constraint

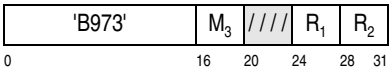
Programming Notes:

1. When bit positions 0-2 of the M_3 field contain zeros, the instruction acts as a NOP, however this is not the preferred instruction with which to create a NOP. When bit positions 0-2 of the M_3 field contain 111 binary, a branch always occurs.
2. When COMPARE LOGICAL AND BRANCH RELATIVE or COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE are the target of an execute-type instruction, the branch is relative to the target address. See “Branch-Address Generation” on page 5-12.
3. The discussion of extended mnemonics in programming note 3 for COMPARE AND BRANCH on page 7-137 also applies to the compare-logical-and-branch instructions.

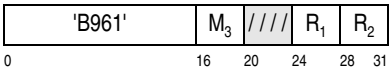
COMPARE LOGICAL AND TRAP

Register-and-register formats:

CLRT R_1,R_2,M_3 [RRF-c]

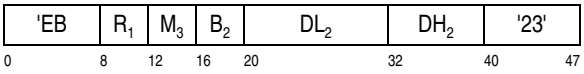


CLGRT R_1,R_2,M_3 [RRF-c]

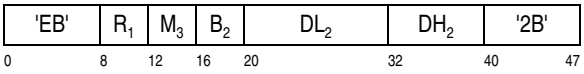


Register-and-storage formats:

CLT $R_1,M_3,D_2(B_2)$ [RSY-b]

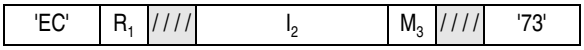


CLGT $R_1,M_3,D_2(B_2)$ [RSY-b]

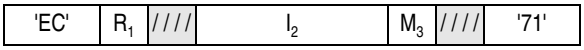


COMPARE LOGICAL IMMEDIATE AND TRAP

CLFIT R_1,I_2,M_3 [RIE-a]



CLGIT R_1,I_2,M_3 [RIE-a]



The first operand is compared with the second operand. If the mask bit in the M_3 field corresponding to the comparison result is one, a compare-and-trap-instruction data exception is recognized, and the operation is completed; otherwise, normal instruction sequencing proceeds with the updated instruction address.

The comparison results and corresponding M_3 bits are as follows:

Comparison Result	M_3 Bit
Equal	0
First operand low	1
First operand high	2

Bit 3 of the M_3 field is reserved and should be zero; otherwise, the program may not operate compatibly in the future.

For COMPARE LOGICAL AND TRAP (CLRT, CLT) and COMPARE LOGICAL IMMEDIATE AND TRAP (CLFIT), the first operand is treated as a 32-bit unsigned binary integer. For COMPARE LOGICAL AND TRAP (CLGRT, CLGT) and COMPARE LOGICAL IMMEDIATE AND TRAP (CLGIT), the first operand is treated as a 64-bit unsigned binary integer.

For COMPARE LOGICAL IMMEDIATE AND TRAP, the second operand is treated as a 16-bit unsigned binary integer. For COMPARE LOGICAL AND TRAP (CLRT, CLT), the second operand is treated as a 32-bit unsigned binary integer. For COMPARE LOGICAL AND TRAP (CLGRT, CLGT), the second operand is treated as a 64-bit unsigned binary integer. When the size of the second operand is smaller than the size of the first operand, the second operand is extended on the left with zeros to match the size of the first operand.

For COMPARE LOGICAL AND TRAP (CLT, CLGT), when bit positions 0-2 of the M_3 field contain either 000 or 111 binary, it is model dependent whether an access exception is recognized for the second-operand location; if accessible, it is model dependent whether a PER zero-address-detection event is recognized for the location.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2, CLT, CLGT)
- Data
- Operation (CLRT, CLGRT, CLFIT, CLGIT, if the general-instructions-extension facility is not installed; CLT, CLGT if the miscellaneous-instruction-extensions facility 1 is not installed)

Programming Notes:

1. Possible uses of COMPARE LOGICAL AND TRAP and COMPARE LOGICAL IMMEDIATE AND TRAP include checking for pointers containing zero (null pointer), or array range checking.

2. On most models, when a data exception is not recognized, the performance of COMPARE LOGICAL AND TRAP and COMPARE LOGICAL IMMEDIATE AND TRAP is typically similar to other COMPARE LOGICAL instructions. When a data exception is recognized, the instruction may be significantly slower than an equivalent COMPARE LOGICAL instruction followed by a BRANCH ON CONDITION instruction.

3. For CLFIT, CLGIT, CLGRT, and CLRT, when bit positions 0-2 of the M_3 field contain zeros, the instruction acts as a NOP, however this is not the preferred instruction with which to create a NOP. This NOP characteristic may not apply to CLGT and CLT.

   When bit positions 0-2 of the M_3 field contain 111 binary, a data exception is always recognized.

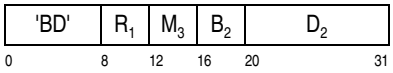
4. When a data exception is recognized, the compare-and-trap data-exception code (DXC FF hex) is stored, as described in “Data Exception” on page 6-25.

5. The discussion of extended mnemonics in programming note 3 for COMPARE AND BRANCH

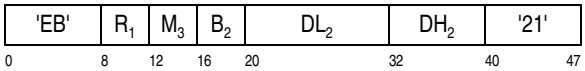
on page 7-137 also applies to the compare-logical-and-trap instructions.

COMPARE LOGICAL CHARACTERS UNDER MASK

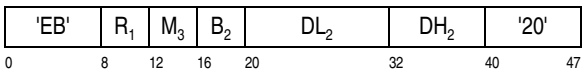
CLM $R_1,M_3,D_2(B_2)$ [RS-b]



CLMY $R_1,M_3,D_2(B_2)$ [RSY-b]



CLMH $R_1,M_3,D_2(B_2)$ [RSY-b]



The first operand is compared with the second operand under control of a mask, and the result is indicated in the condition code.

The contents of the M_3 field are used as a mask. These four bits, left to right, correspond one for one with four bytes, left to right, of general register R_1. For COMPARE LOGICAL CHARACTERS UNDER MASK (CLM, CLMY), the four bytes to which the mask bits correspond are in bit positions 32-63 of general register R_1. For COMPARE LOGICAL CHARACTERS UNDER MASK (CLMH), the four bytes are in the high-order half, bit positions 0-31, of the register. The byte positions corresponding to ones in the mask are considered as a contiguous field and are compared with the second operand. The second operand is a contiguous field in storage, starting at the second-operand address and equal in length to the number of ones in the mask. The bytes in the general register corresponding to zeros in the mask do not participate in the operation.

The comparison proceeds left to right, byte by byte, and ends as soon as an inequality is found or the end of the fields is reached.

When the mask is not zero, exceptions associated with storage-operand access are recognized for no more than the number of bytes specified by the mask. Access exceptions may or may not be recognized for the portion of a storage operand to the right of the first unequal byte. When the mask is zero, access exceptions are recognized for one byte at the second-operand address.

The displacement for CLM is treated as a 12-bit unsigned binary integer. The displacement for CLMY and CLMH is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Operands equal, or mask bits all zeros
1 First operand low
2 First operand high
3 --

Program Exceptions:

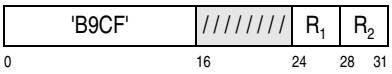
- Access (fetch, operand 2)
- Operation (CLMY, if the long-displacement facility is not installed)

Programming Note: An example of the use of the COMPARE LOGICAL CHARACTERS UNDER MASK instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

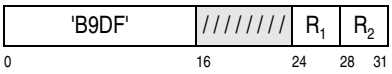
COMPARE LOGICAL HIGH

Register-and-register formats:

CLHHR R_1,R_2 [RRE]

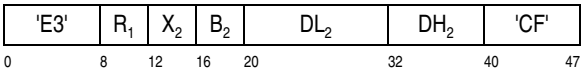


CLHLR R_1,R_2 [RRE]



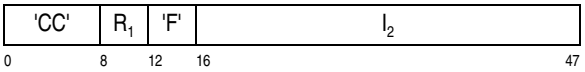
Register-and-storage format:

CLHF $R_1,D_2(X_2,B_2)$ [RXY-a]



COMPARE LOGICAL IMMEDIATE HIGH

CLIH R_1,I_2 [RIL-a]



The first operand is compared with the second operand, and the result is indicated in the condition code.

The operands are treated as 32-bit unsigned binary integers. The first operand is in bit positions 0-31 of general register R_1; bit positions 32-63 of the register are ignored. For COMPARE LOGICAL HIGH (CLHHR), the second operand is in bit positions 0-31 of general register R_2; bit positions 32-63 of the register are ignored. For COMPARE LOGICAL HIGH (CLHLR), the second operand is in bit positions 32-63 of general register R_2; bit positions 0-31 of the register are ignored.

The displacement for CLHF is treated as a 20-bit signed binary integer.

Resulting Condition Code:

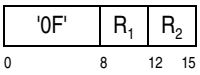
0 Operands equal
1 First operand low
2 First operand high
3 --

Program Exceptions:

- Access (operand 2 of CLHF only)
- Operation (if the high-word facility is not installed)

COMPARE LOGICAL LONG

CLCL R_1,R_2 [RR]



The first operand is compared with the second operand, and the result is indicated in the condition code. The shorter operand is considered to be extended on the right with padding bytes.

The R_1 and R_2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers R_1 and R_2, respectively. The number of bytes in the first-operand and second-operand locations is specified by unsigned binary integers in bit positions 40-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively. Bit positions 32-39 of general register $R_2 + 1$ contain the padding byte. The contents of bit positions 0-39 of general register $R_1 + 1$ and of bit positions 0-31 of general register $R_2 + 1$ are ignored.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-196.

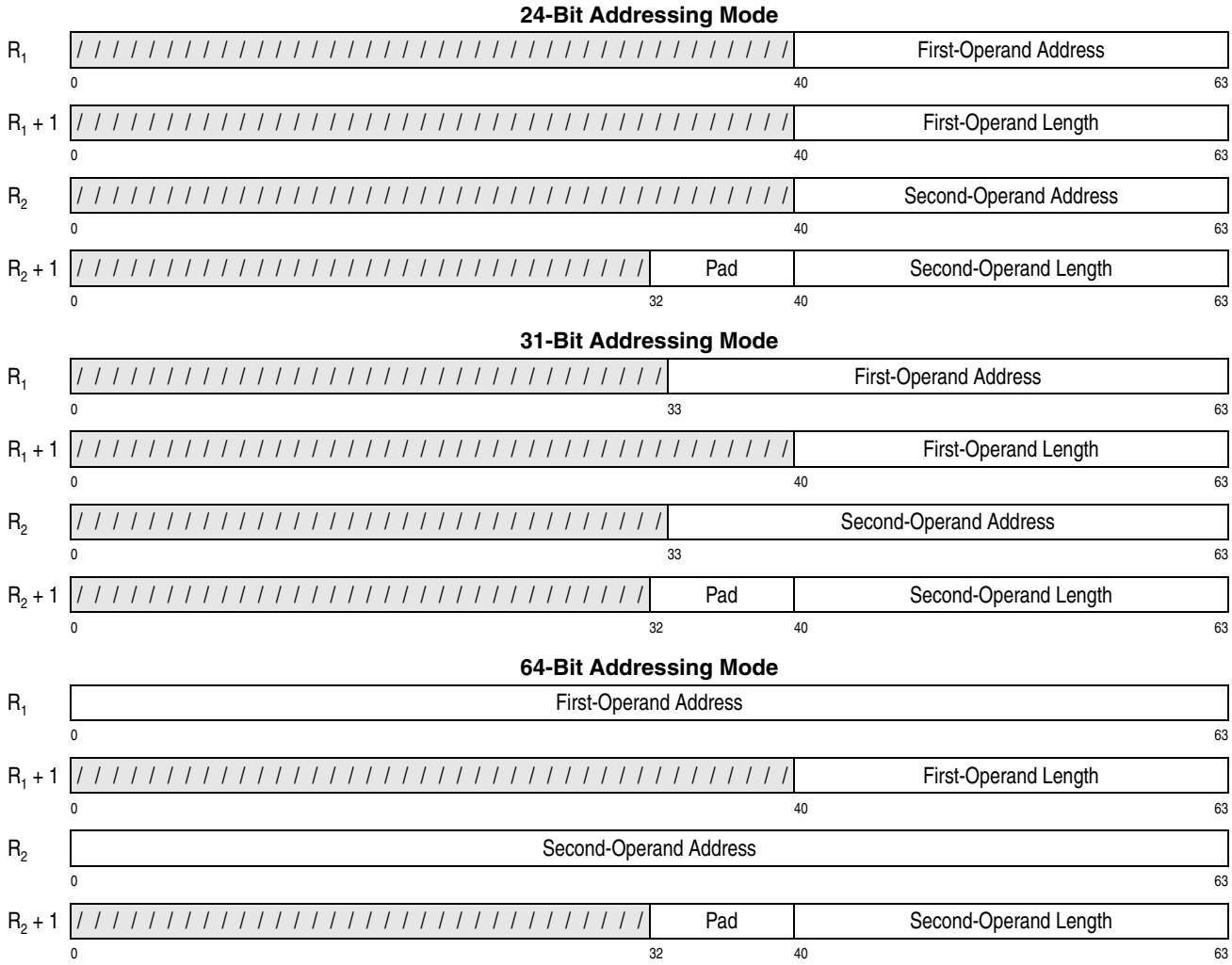


Figure 7-196. Register Contents for COMPARE LOGICAL LONG

The comparison proceeds left to right, byte by byte, and ends as soon as an inequality is found or the end of the longer operand is reached. If the operands are not of the same length, the shorter operand is considered to be extended on the right with the appropriate number of padding bytes.

If both operands are of zero length, the operands are considered to be equal.

The execution of the instruction is interruptible. When an interruption occurs, other than one that follows termination, the lengths in general registers $R_1 + 1$ and $R_2 + 1$ are decremented by the number of bytes compared, and the addresses in general registers R_1 and R_2 are incremented by the same number, so that the instruction, when reexecuted, resumes at the point of interruption. In the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R_1 and R_2 are set to zeros, and the contents of bit positions 0-31 remain unchanged. In any addressing mode, the contents of bit positions 0-39 of general registers $R_1 + 1$ and $R_2 + 1$ remain unchanged, and the condition code is unpredictable. If the operation is interrupted after the shorter operand has been exhausted, the length field pertaining to the shorter operand is zero, and its address is updated accordingly.

If the operation ends because of an inequality, the address fields in general registers R_1 and R_2 at com-

pletion identify the first unequal byte in each operand. The lengths in bit positions 40-63 of general registers $R_1 + 1$ and $R_2 + 1$ are decremented by the number of bytes that were equal, unless the inequality occurred with the padding byte, in which case the length field for the shorter operand is set to zero. The addresses in general registers R_1 and R_2 are incremented by the amounts by which the corresponding length fields were reduced.

If the two operands, including the padding byte, if necessary, are equal, both length fields are made zero at completion, and the addresses are incremented by the corresponding operand-length values.

At the completion of the operation, in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R_1 and R_2 are set to zeros, even when one or both of the initial length values are zero. In any addressing mode, the contents of bit positions 0-39 of general registers $R_1 + 1$ and $R_2 + 1$ remain unchanged.

Access exceptions for the portion of a storage operand to the right of the first unequal byte may or may not be recognized. For operands longer than 2K bytes, access exceptions are not recognized more than 2K bytes beyond the byte being processed. Access exceptions are not indicated for locations more than 2K bytes beyond the first unequal byte.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

Resulting Condition Code:

0 Operands equal, or both zero length
1 First operand low
2 First operand high
3 --

Program Exceptions:

- Access (fetch, operands 1 and 2)
- Specification
- Transaction constraint

Programming Notes:

1. An example of the use of the COMPARE LOGICAL LONG instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

2. When the R_1 and R_2 fields are the same, the operation proceeds in the same way as when two distinct pairs of registers having the same contents are specified, except that the contents of the designated registers are incremented or decremented only by the number of bytes compared, not by twice the number of bytes compared. In the absence of dynamic modification of the operand area by another CPU or by a channel program, condition code 0 is set. However, it is unpredictable whether access exceptions are recognized for the operand since the operation can be completed without storage being accessed.

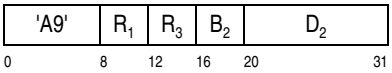
3. Special precautions should be taken when COMPARE LOGICAL LONG is made the target of an execute-type instruction. See the programming note concerning interruptible instructions under EXECUTE.

4. Other programming notes concerning interruptible instructions are included in "Interruptible Instructions" in Chapter 5, "Program Execution."

5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

COMPARE LOGICAL LONG EXTENDED

CLCLE $R_1,R_3,D_2(B_2)$ [RS-a]



The first operand is compared with the third operand until unequal bytes are compared, the end of the longer operand is reached, or a CPU-determined number of bytes have been compared, whichever occurs first. The shorter operand is considered to be extended on the right with padding bytes. The result is indicated in the condition code.

The R_1 and R_3 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and third operand is designated by the contents of general registers R_1 and R_3, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_3 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the entire contents of general registers $R_1 + 1$ and $R_3 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R_1 and R_3 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_3 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The second-operand address is not used to address data; instead, the rightmost eight bits of the second-operand address, bits 56-63, are the padding byte. Bits 0-55 of the second-operand address are ignored.

The contents of the registers and address just described are shown in Figure 7-197.

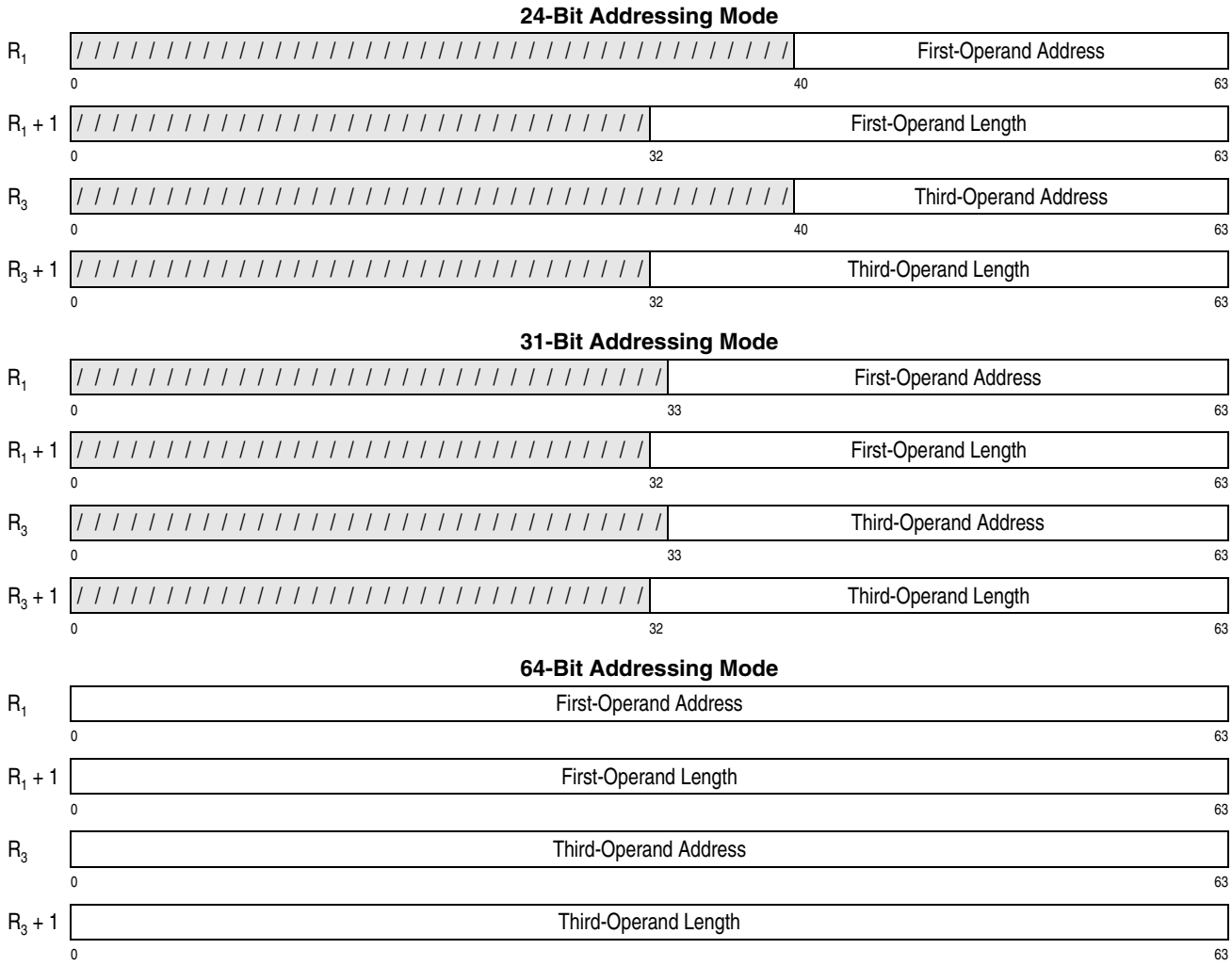


Figure 7-197. Register Contents and Second-Operand Address for COMPARE LOGICAL LONG EXTENDED (Part 1 of 2)

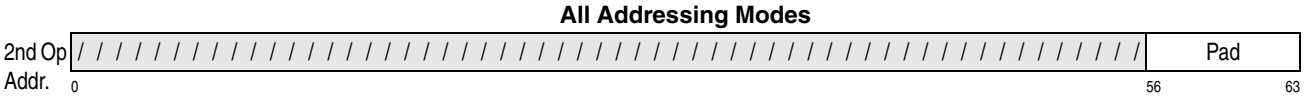


Figure 7-197. Register Contents and Second-Operand Address for COMPARE LOGICAL LONG EXTENDED (Part 2 of 2)

The comparison proceeds left to right, byte by byte, and ends as soon as an inequality is found, the end of the longer operand is reached, or a CPU-determined number of bytes have been compared, whichever occurs first. If the operands are not of the same length, the shorter operand is considered to be extended on the right with the appropriate number of padding bytes.

If both operands are of zero length, the operands are considered to be equal.

If the operation ends because of an inequality, the address fields in general registers R_1 and R_3 at completion identify the first unequal byte in each operand. The lengths in bit positions 32-63, in the 24-bit or 31-bit addressing mode, or in bit positions 0-63, in the 64-bit addressing mode, of general registers $R_1 + 1$ and $R_3 + 1$ are decremented by the number of bytes that were equal, unless the inequality occurred with the padding byte, in which case the length field for the shorter operand is set to zero. The addresses in general registers R_1 and R_3 are incremented by the amounts by which the corresponding length fields were decremented. Condition code 1 is set if the first operand is low, or condition code 2 is set if the first operand is high.

If the two operands, including the padding byte, if necessary, are equal, both length fields are made zero at completion, and the addresses are incremented by the corresponding operand-length values. Condition code 0 is set.

If the operation is completed because a CPU-determined number of bytes have been compared without finding an inequality or reaching the end of the longer operand, the lengths in general registers $R_1 + 1$ and $R_3 + 1$ are decremented by the number of bytes compared, and the addresses in general registers R_1 and R_3 are incremented by the same number, so that the instruction, when reexecuted, resumes at the next bytes to be compared. If the operation is completed after the shorter operand has been exhausted, the length field pertaining to the shorter operand is zero, and the operand address is updated accordingly. Condition code 3 is set.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_3, and $R_3 + 1$, always remain unchanged.

The padding byte may be formed from $D_2(B_2)$ multiple times during the execution of the instruction, and the registers designated by R_1 and R_3 may be updated multiple times. Therefore, if B_2 equals R_1, $R_1 + 1$, R_3, or $R_3 + 1$ and is subject to change during the execution of the instruction, the results are unpredictable.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed. The maximum amount is approximately 4K bytes of either operand.

At the completion of the operation in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R_1 and R_3 may be set to zeros or may remain unchanged from their original values, even when one or both of the initial length values are zero.

Access exceptions for the portion of a storage operand to the right of the first unequal byte may or may not be recognized. For operands longer than 4K bytes, access exceptions are not recognized more than 4K bytes beyond the byte being processed. Access exceptions are not indicated for locations more than 4K bytes beyond the first unequal byte.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

Resulting Condition Code:

0 All bytes compared; operands equal, or both zero length
1 All bytes compared, first operand low
2 All bytes compared, first operand high
3 CPU-determined number of bytes compared without finding an inequality

Program Exceptions:

- Access (fetch, operands 1 and 3)
- Specification
- Transaction constraint

Programming Notes:

1. COMPARE LOGICAL LONG EXTENDED is intended for use in place of COMPARE LOGICAL LONG when the operand lengths are specified as 32-bit or 64-bit binary integers. COMPARE LOGICAL LONG EXTENDED sets condition code 3 in cases in which COMPARE LOGICAL LONG would be interrupted.

2. When condition code 3 is set, the program can simply branch back to the instruction to continue the comparison. The program need not determine the number of bytes that were compared.

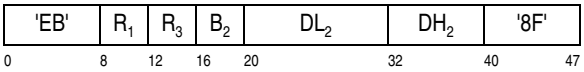
3. The function of not processing more than approximately 4K bytes of either operand is intended to permit software polling of a flag that may be set by a program on another CPU during long operations.

4. When the R_1 and R_3 fields are the same, the operation proceeds in the same way as when two distinct pairs of registers having the same contents are specified, except that the contents of the designated registers are incremented or decremented only by the number of bytes compared, not by twice the number of bytes compared. In the absence of dynamic modification of the operand area by another CPU or by a channel program, the condition code is finally set to 0 after possible settings to 3. However, it is unpredictable whether access exceptions are recognized for the operand since the operation can be completed without storage being accessed. If storage is not accessed, condition code 3 may or may not be set regardless of the operand length.

5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

COMPARE LOGICAL LONG UNICODE

CLCLU $R_1,R_3,D_2(B_2)$ [RSY-a]



The first operand is compared with the third operand until unequal two-byte Unicode characters are compared, the end of the longer operand is reached, or a CPU-determined number of characters have been compared, whichever occurs first. The shorter operand is considered to be extended on the right with two-byte padding characters. The result is indicated in the condition code.

The R_1 and R_3 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost character of the first operand and third operand is designated by the contents of general registers R_1 and R_3, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_3 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 0-63 of general registers $R_1 + 1$ and $R_3 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The contents of general registers $R_1 + 1$ and $R_3 + 1$ must specify an even number of bytes; otherwise, a specification exception is recognized.

The handling of the addresses in general registers R_1 and R_3 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_3 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of the registers constitute the address.

The second-operand address is not used to address data; instead, the rightmost 16 bits of the second-operand address, bits 48-63, are the two-byte padding character. Bits 0-47 of the second-operand address are ignored.

The contents of the registers and address just described are shown in Figure 7-198.

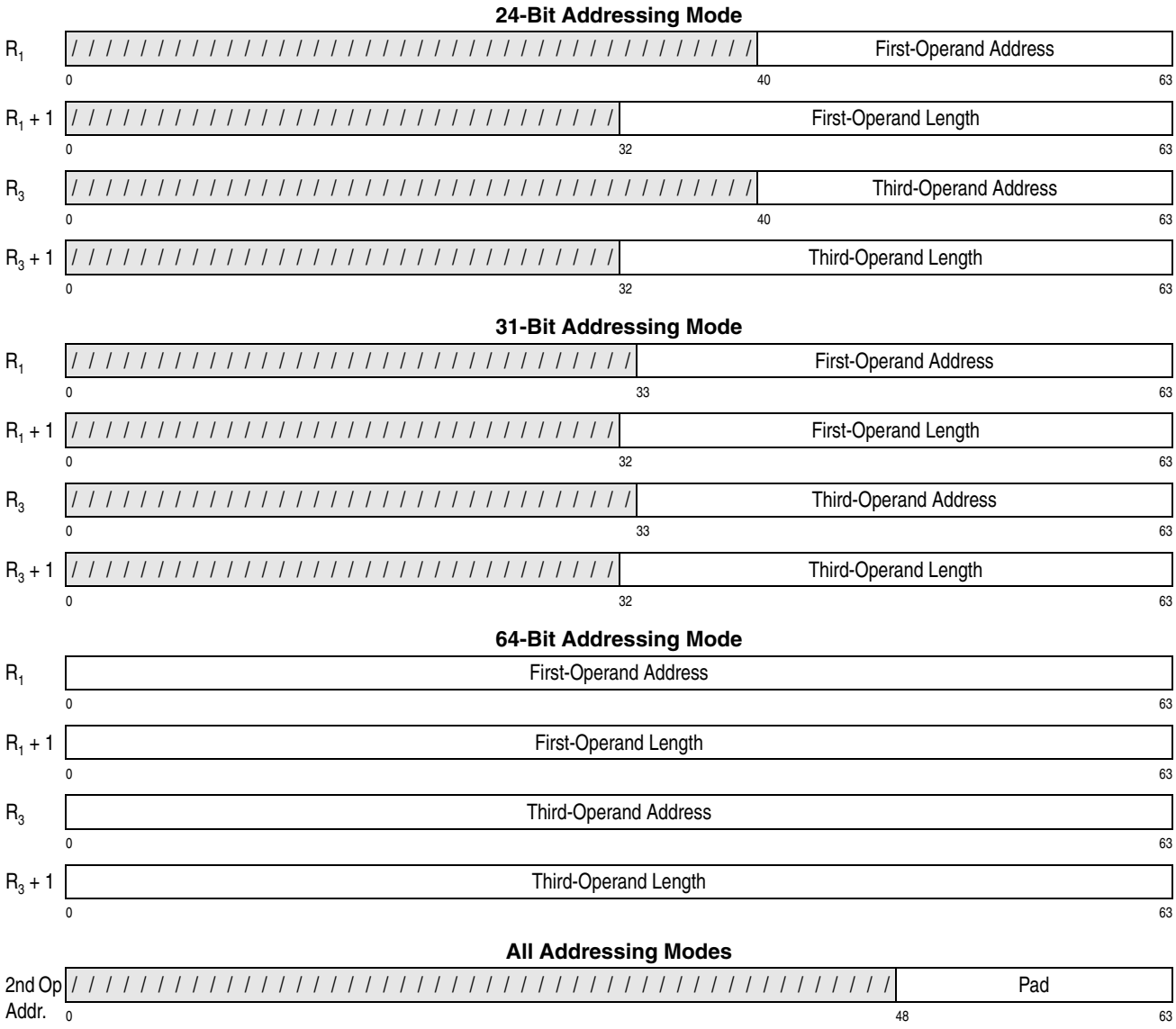


Figure 7-198. Register Contents and Second-Operand Address for COMPARE LOGICAL LONG UNICODE

The comparison proceeds left to right, character by character, and ends as soon as an inequality is found, the end of the longer operand is reached, or a CPU-determined number of characters have been compared, whichever occurs first. If the operands are not of the same length, the shorter operand is considered to be extended on the right with the appropriate number of two-byte padding characters.

If both operands are of zero length, the operands are considered to be equal.

If the operation ends because of an inequality, the address fields in general registers R_1 and R_3 at completion identify the first unequal two-byte character in each operand. The lengths in bit positions 32-63, in the 24-bit or 31-bit addressing mode, or in bit positions 0-63, in the 64-bit addressing mode, of general

registers $R_1 + 1$ and $R_3 + 1$ are decremented by 2 times the number of characters that were equal, unless the inequality occurred with the two-byte padding character, in which case the length field for the shorter operand is set to zero. The addresses in general registers R_1 and R_3 are incremented by the amounts by which the corresponding length fields were decremented. Condition code 1 is set if the first operand is low, or condition code 2 is set if the first operand is high.

If the two operands, including the two-byte padding character, if necessary, are equal, both length fields are made zero at completion, and the addresses are incremented by the corresponding operand-length values. Condition code 0 is set.

If the operation is completed because a CPU-determined number of characters have been compared without finding an inequality or reaching the end of the longer operand, the lengths in general registers $R_1 + 1$ and $R_3 + 1$ are decremented by 2 times the number of characters compared, and the addresses in general registers R_1 and R_3 are incremented by the same number, so that the instruction, when reexecuted, resumes at the next characters to be compared. If the operation is completed after the shorter operand has been exhausted, the length field pertaining to the shorter operand is zero, and the operand address is updated accordingly. Condition code 3 is set.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_3, and $R_3 + 1$, always remain unchanged.

The two-byte padding character may be formed from $D_2(B_2)$ multiple times during the execution of the instruction, and the registers designated by R_1 and R_3 may be updated multiple times. Therefore, if B_2 equals R_1, $R_1 + 1$, R_3, or $R_3 + 1$ and is subject to change during the execution of the instruction, the results are unpredictable.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

At the completion of the operation in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R_1 and R_3 may be set to zeros or may remain unchanged from their original values, including the case when one or both of the initial length values are zero.

Access exceptions for the portion of a storage operand to the right of the first unequal character may or may not be recognized. For operands longer than 4K bytes, access exceptions are not recognized more than 4K bytes beyond the character being processed. Access exceptions are not indicated for locations more than 4K bytes beyond the first unequal character.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field or length associated with that operand is odd.

Resulting Condition Code:

0 All characters compared; operands equal, or both zero length
1 First operand low
2 First operand high
3 CPU-determined number of characters compared without finding an inequality

Program Exceptions:

- Access (fetch, operands 1 and 3)
- Operation (if the extended-translation facility 2 is not installed)
- Specification
- Transaction constraint

Programming Notes:

1. COMPARE LOGICAL LONG UNICODE is intended for use in place of COMPARE LOGICAL LONG or COMPARE LOGICAL LONG EXTENDED when two-byte characters are to be compared. The characters may be Unicode characters or any other double-byte characters. COMPARE LOGICAL LONG UNICODE sets condition code 3 in cases in which COMPARE LOGICAL LONG would be interrupted.

2. When condition code 3 is set, the program can simply branch back to the instruction to continue the comparison. The program need not determine the number of characters that were compared.

3. When the R_1 and R_3 fields are the same, the operation proceeds in the same way as when two distinct pairs of registers having the same contents are specified, except that the contents of the designated registers are incremented or decremented only by 2 times the number of characters compared, not by 4 times the number of characters compared. In the absence of dynamic modification of the operand area by another CPU or by a channel program, the condition code is finally set to 0 after possible settings to 3. However, it is unpredictable whether access exceptions are recognized for the operand since the operation can be completed without storage being accessed. If storage is not accessed, condition code 3 may or may not be set regardless of the operand length.

4. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

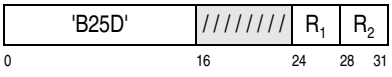
5. The padding character may be represented in the displacement field of the instruction. The following example illustrates padding with a Unicode space character.

   ```
   CLCLU   6,8,X'020'
   ```

   When the B_2 field of the instruction designates general register 0, and the long-displacement facility is not installed, the padding character is limited to a character whose representation is less than or equal to FFF hex.

COMPARE LOGICAL STRING

CLST R_1,R_2 [RRE]



The first operand is compared with the second operand until unequal bytes are compared, the end of either operand is reached, or a CPU-determined number of bytes have been compared, whichever occurs first. The CPU-determined number is at least 256. The result is indicated in the condition code.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers R_1 and R_2, respectively.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The first and second operands may be of the same or different lengths. The end of an operand is indicated by an ending character in the last byte position of the operand. The ending character to be used to determine the end of an operand is specified in bit positions 56-63 of general register 0. Bit positions 32-55 of general register 0 are reserved for possible future extensions and must contain all zeros; otherwise, a specification exception is recognized.

The operation proceeds left to right, byte by byte, and ends as soon as the ending character is encountered in either or both operands, unequal bytes which do not include an ending character are compared, or a CPU-determined number of bytes have been compared, whichever occurs first. The CPU-determined number is at least 256. When the ending character is encountered simultaneously in both operands, including when it is in the first byte position of the operands, the operands are of the same length and are considered to be equal, and condition code 0 is set. When the ending character is encountered in only one operand, that operand, which is the shorter operand, is considered to be low, and condition code 1 or 2 is set. Condition code 1 is set if the first operand is low, or condition code 2 is set if the second operand is low. Similarly, when unequal bytes which do not include an ending character are compared, condition code 1 is set if the lower byte is in the first operand, or condition code 2 is set if the lower byte is in the second operand. When a CPU-determined number of bytes have been compared, condition code 3 is set.

When condition code 1 or 2 is set, the address of the last byte processed in the first and second operands is placed in general registers R_1 and R_2, respectively. That is, when condition code 1 is set, the address of the ending character or first unequal byte in the first operand, whichever was encountered, is placed in general register R_1, and the address of the second-operand byte corresponding in position to the first-operand byte is placed in general register R_2. When

condition code 2 is set, the address of the ending character or first unequal byte in the second operand, whichever was encountered, is placed in general register R_2, and the address of the first-operand byte corresponding in position to the second-operand byte is placed in general register R_1. When condition code 3 is set, the address of the next byte to be processed in the first and second operands is placed in general registers R_1 and R_2, respectively. Whenever an address is placed in a general register, bits 32-39 of the register, in the 24-bit addressing mode, or bit 32 in the 31-bit addressing mode, are set to zeros. Bits 0-31 of the R_1 and R_2 registers always remain unchanged in the 24-bit or 31-bit mode.

When condition code 0 is set, the contents of general registers R_1 and R_2 remain unchanged.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

Access exceptions for the first and second operands are recognized only for that portion of the operand which is necessarily examined in the operation.

Resulting Condition Code:

0 Entire operands equal; general registers R_1 and R_2 unchanged
1 First operand low; general registers R_1 and R_2 updated with addresses of last bytes processed
2 First operand high; general registers R_1 and R_2 updated with addresses of last bytes processed
3 CPU-determined number of bytes equal; general registers R_1 and R_2 updated with addresses of next bytes

Program Exceptions:

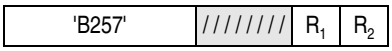
- Access (fetch, operands 1 and 2)
- Specification
- Transaction constraint

Programming Notes:

1. Several examples of the use of the COMPARE LOGICAL STRING instruction are given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. When condition code 0 is set, no indication is given of the position of either ending character.
3. When condition code 3 is set, the program can simply branch back to the instruction to continue the comparison. The program need not determine the number of bytes that were compared.
4. R_1 or R_2 may be zero, in which case general register 0 is treated as containing an address and also the ending character.
5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

COMPARE UNTIL SUBSTRING EQUAL

CUSE R_1,R_2 [RRE]



The first operand is compared with the second operand until equal substrings (sequences of bytes) of a specified length are found, the end of the longer operand is reached, or a CPU-determined number of unequal bytes have been compared, whichever occurs first. The shorter operand is considered to be extended on the right with padding bytes. The CPU-determined number is at least 256. The result is indicated in the condition code.

The R_1 and R_2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and second operand is specified by the contents of the R_1 and R_2 general registers, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the 32-bit signed binary integer in bit positions 32-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively. In the 64-bit addressing mode, the number of bytes is specified by the 64-bit signed binary integer in bit positions 0-63 of those registers. When an operand length is negative, it is treated as zero, and it remains unchanged upon completion of the instruction.

Bits 56-63 of general register 0 specify the unsigned substring length, a value of 0-255, in bytes. Bits 56-63 of general register 1 are the padding byte. Bits 0-55 of general registers 0 and 1 are ignored.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-199.

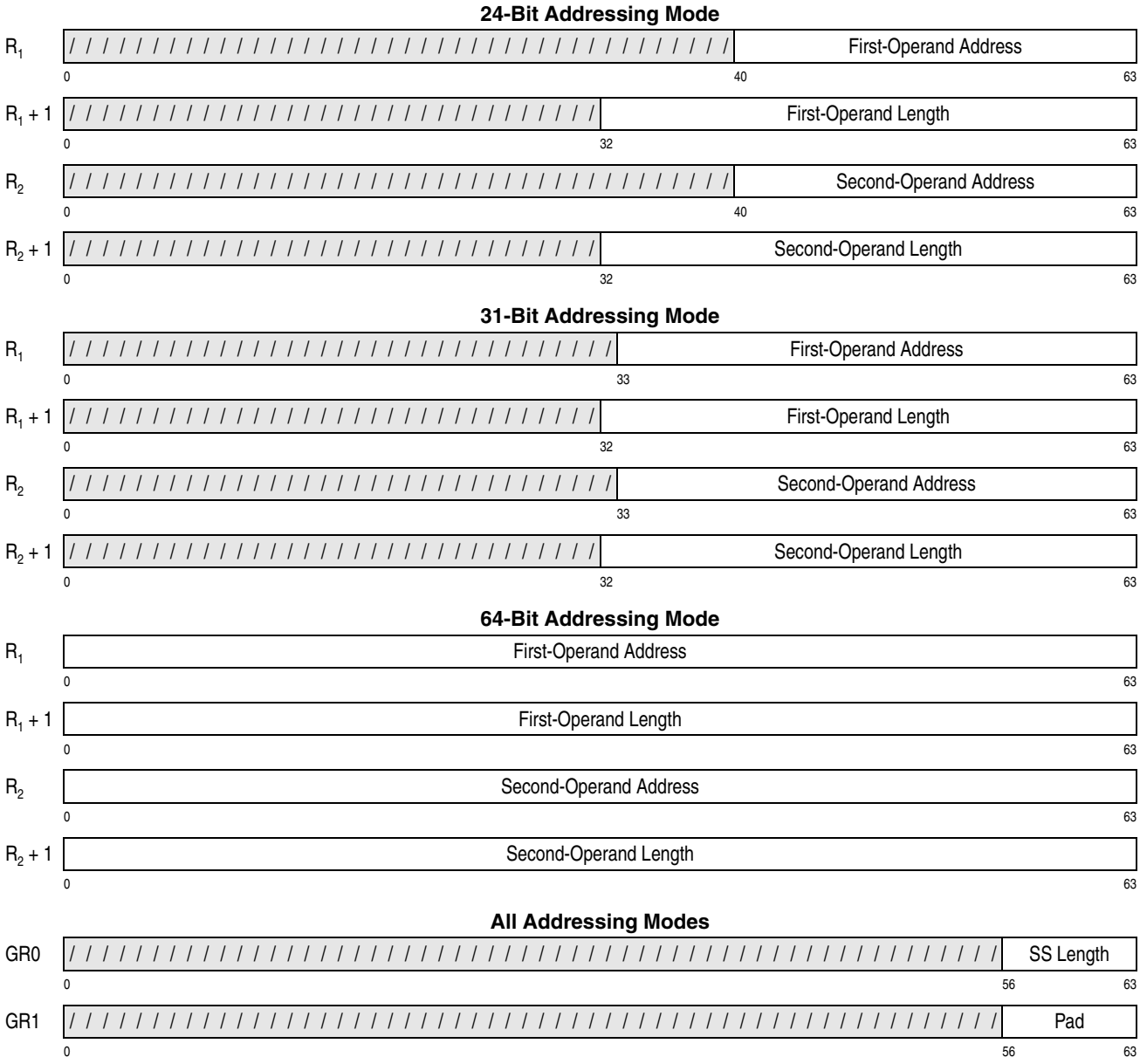


Figure 7-199. Register Contents for COMPARE UNTIL SUBSTRING EQUAL

The result is obtained as if the operands were processed from left to right. However, multiple accesses may be made to all or some of the bytes of each operand.

The comparison proceeds left to right, byte by byte, and ends as soon as (1) equal substrings of the specified length are found, (2) the end of the longer operand is reached without finding equal substrings of the specified length, or (3) the last bytes compared are unequal, and a CPU-determined number of bytes have been compared. The CPU-determined number is at least 256. If the operands are not of the same length, the shorter operand is considered to be extended on the right with the appropriate number of padding bytes.

If the operation ends because equal substrings of the specified length were found, the condition code is set to 0. If the operation ends because the end of the longer operand was reached without finding equal substrings of the specified length, the condition code is set to 1 if equal bytes were the last bytes compared, or it is set to 2 if unequal bytes were the last bytes compared. If the operation ends because unequal bytes were compared when a CPU-determined number of bytes had been compared, the condition code is set to 3.

If the specified substring length is zero, it is considered that equal substrings of the specified length were found, and condition code 0 is set.

If both operands are of zero length but the specified substring length is not zero, it is considered that the end of the longer operand was reached when unequal bytes were the last bytes compared, and condition code 2 is set.

If equal bytes have been compared but then unequal bytes are compared, it is considered that all bytes so far compared are unequal.

At the completion of the operation, the operand-length fields in the $R_1 + 1$ and $R_2 + 1$ registers are decremented by the number of unequal bytes compared (including equal bytes before unequal bytes compared), and the addresses in the R_1 and R_2 registers are incremented by the same number. However, in the case when a byte of the longer operand is compared against the padding byte, the length field for the shorter operand is not decremented below zero, and the corresponding address is not incremented above the address of the first byte after the shorter operand. In the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the addresses in bit positions 32-63 of registers R_1 and R_2 are set to zeros, even if the substring length is zero or both operand lengths are initially zero.

Thus, when condition code 0 or 1 is set, the resulting addresses in the R_1 and R_2 registers designate the first bytes of equal substrings in the two operands, and the lengths in the $R_1 + 1$ and $R_2 + 1$ registers have been decremented by the number of bytes preceding the equal substrings, except when the equal substring in the shorter operand begins with the padding byte, in which case the length field for the shorter operand is zero, and the corresponding address field has been incremented by the operand length. When condition code 2 is set, each address field designates the first byte after the corresponding operand, and both length fields are zero. When condition code 3 is set, each address field designates the first byte after the last compared byte of the corresponding operand, and both length fields have been decremented by the number of bytes compared, except that a length field is not decremented below zero.

When the contents of the R_1 and R_2 fields are the same, the first and second operands may be compared, or the condition code may be set to 0 or 1 without comparing the operands.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_2, and $R_2 + 1$, always remain unchanged.

The substring length and padding byte may be fetched from general registers 0 and 1 multiple times during the execution of the instruction, and the registers designated by R_1 and R_2 may be updated multiple times. Therefore, if R_1 or R_2 is zero, the results are unpredictable.

When condition code 3 is set, the general registers used by the instruction have been set so that the remainder of the operands can be processed by simply branching back and reexecuting the instruction.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

The execution of the instruction is interruptible when the last bytes compared are unequal; it is not interruptible when the last bytes compared are equal. When an interruption occurs, other than one that follows termination, the contents of the registers designated by the R_1 and R_2 fields are updated the same as upon normal completion of the instruction, so that

the instruction, when reexecuted, resumes at the point of interruption. The condition code is unpredictable.

Access exceptions for the portion of a storage operand to the right of the last byte processed may or may not be recognized. For operands longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd. Although the operand address and length fields remain unchanged when a zero substring length is specified, the recognition of access exceptions is not necessarily prevented.

Resulting Condition Code:

0 Equal substrings of specified length found
1 End of longer operand reached when last bytes compared are equal
2 End of longer operand reached when last bytes compared are unequal
3 Last bytes compared are unequal, and CPU-determined number of bytes compared

Program Exceptions:

- Access (fetch, operands 1 and 2)
- Specification
- Transaction constraint

Programming Notes:

1. When the R_1 and R_2 fields are the same, the operation proceeds in the same way as when two distinct pairs of registers having the same contents are specified, and, in the absence of dynamic modification of the operand area by another CPU or by a channel program, condition code 0, 1, or 2 is set (as explained in the next note). However, it is unpredictable whether access exceptions are recognized for the operand since the operation can be completed without storage being accessed.

2. If the contents of the R_1 and R_2 fields are the same and the operand length is nonzero, and provided that another CPU or a channel program is not changing an operand, condition code 0 is set if the operand length is equal to or greater than the specified substring length, or condition code 1 is set if the operand length is less than the specified substring length. Whether or not R_1 equals R_2, if both operand lengths are zero, condition code 0 is set if the specified substring length is zero, or condition code 2 is set if the specified substring length is nonzero. In all of these cases, the addresses in the R_1 and R_2 registers and the lengths in the $R_1 + 1$ and $R_2 + 1$ registers remain unchanged.

3. Special precautions should be taken when COMPARE UNTIL SUBSTRING EQUAL is made the target of an execute-type instruction. See the programming note concerning interruptible instructions under EXECUTE.

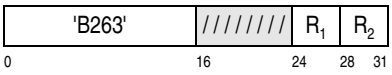
4. Other programming notes concerning interruptible instructions are included in “Interruptible Instructions” on page 5-24.

5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

6. The storage-operand references of COMPARE UNTIL SUBSTRING EQUAL may be multiple-access references. (See “Storage-Operand Consistency” on page 5-125.)

COMPRESSION CALL

CMPSC R_1,R_2 [RRE]



This definition assumes knowledge of the introductory information and information about dictionary formats in Appendix D.

The second operand is compressed or expanded, depending on a specification in general register 0, and the results are placed at the first-operand location. The compressed-data operand consists of either index symbols corresponding to entries in a dictionary designated by an address in general register 1, or codewords which are converted to index symbols depending on the specification of entropy encoding. This dictionary is a compression dictionary during a compression operation or an expansion dictionary during an expansion operation.

During compression when format-1 sibling descriptors are specified in general register 0, an expansion dictionary immediately follows the compression dictionary. During compression when the symbol-translation option is specified in general register 0, the index symbols resulting from compression are translated to interchange symbols by means of a symbol-translation table designated by the address and an offset in general register 1, and it is the interchange symbols that are placed at the first-operand location. During compression when the order preservation option is specified in general register 0, a modified form of symbol translation occurs. During compression when the entropy-encoding option is specified in general register 0, the index symbols are translated into codewords via a modified version of symbol translation whose parameters are specified by the address and offset in general register 1. The number of bits in an index or interchange symbol in the compressed-data operand or in an index symbol used in entropy encoding is specified in general register 0. The operation proceeds until the end of either operand is reached or a CPU-determined amount of data has been processed, whichever occurs first. The results are indicated in the condition code.

The R_1 and R_2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte containing any bit of the first operand and second operand is designated by an address in general registers R_1 and R_2, respectively. The number of bytes containing any bits of the first operand and second operand is specified by bits 32-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively, in the 24-bit or 31-bit addressing mode or by bits 0-63 of the registers in the 64-bit addressing mode. The contents of general registers $R_1 + 1$ and $R_2 + 1$ are treated as 32-bit unsigned binary integers in the 24-bit or 31-bit addressing mode or as 64-bit unsigned binary integers in the 64-bit addressing mode.

The location of the leftmost byte of the compression dictionary during compression, or of the expansion dictionary during expansion, is designated on a 4 K-byte boundary by an address in general register 1. Optionally, general register 1 also contains an offset to the first byte of an entropy descriptor or symbol translation table.

24-Bit Addressing Mode

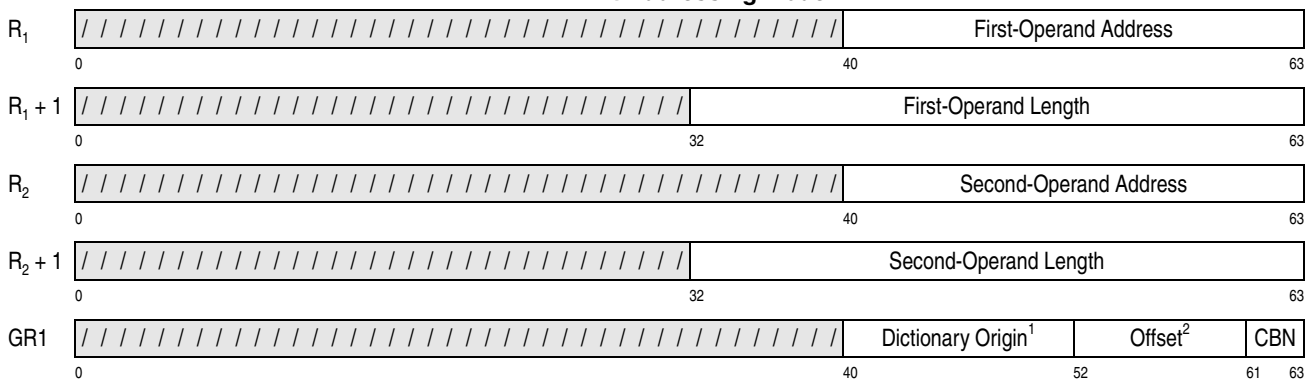
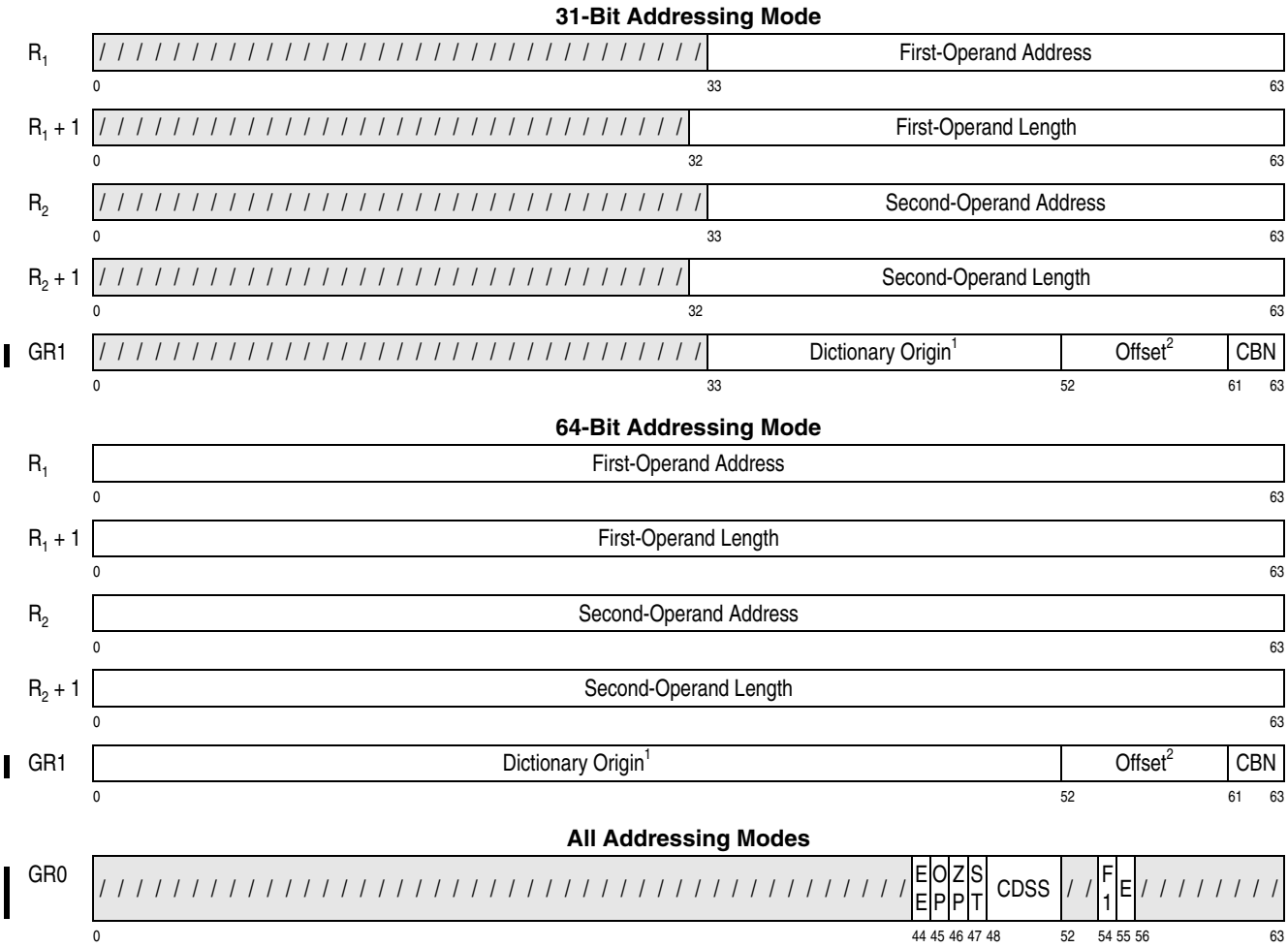


Figure 7-200. Register Contents for COMPRESSION CALL (Part 1 of 2)



Explanation:

1 Compression dictionary during compression, or expansion dictionary during expansion

2 The Offset field is only meaningful during compression when any of the ST, EE, and OP bits are one. When the ST, EE, and OP bits are zero during compression, or the EE bit is zero during expansion, the Offset field is ignored.

CBN Compressed-data bit number

CDSS (binary)	Symbol Size	Dictionary Size
0000	Causes a specification exception to be recognized	
0001	9 bits	512 entries, 4K bytes
0010	10 bits	1K entries, 8K bytes
0011	11 bits	2K entries, 16K bytes
0100	12 bits	4K entries, 32K bytes
0101	13 bits	8K entries, 64K bytes
0110-1111	Causes a specification exception to be recognized	

E Expansion operation

EE Entropy-encoding option (when the entropy-encoding compression facility is installed).

F1 Format-1 sibling descriptors (ignored during expansion)

Offset Offset to symbol translation table or entropy descriptor

OP Order-preservation option (ignored during expansion, or when the order-preserving-compression facility is not installed)

ST Symbol-translation option (ignored during expansion, implied one for entropy encoding)

ZP Zero padding control (when the CMPSC-enhancement facility is installed).

Figure 7-200. Register Contents for COMPRESSION CALL (Part 2 of 2)

The handling of the addresses in general registers R_1, R_2, and 1 is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of registers R_1 and R_2 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of registers R_1 and R_2 constitute the address. In the 24-bit addressing mode, the contents of bit positions 40-51 of register 1, with 12 rightmost zeros appended, constitute the address, and the contents of bit positions 0-39 and 52-63 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-51 of register 1, with 12 rightmost zeros appended, constitute the address, and the contents of bit positions 0-32 and 52-63 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-51 of register 1, with 12 rightmost zeros appended, constitute the address, and the contents of bit positions 52-63 are ignored.

Although the contents of bit positions 52-63 of general register 1 are ignored in the formation of a dictionary address, those contents are used as follows. The contents of bit positions 61-63 of the register are the compressed-data bit number (CBN). At the beginning of the operation, the CBN designates the leftmost bit within the leftmost byte of the compressed-data operand. The compressed-data operand is the first operand during compression, or it is the second operand during expansion. When the symbol-translation or order-preservation option is specified during compression and the entropy-encoding option is not specified, the contents of bit positions 52-60 of the register, with seven rightmost zeros appended, are the byte offset from the beginning of the compression dictionary to the leftmost byte of the symbol-translation table. When the entropy-encoding option is specified the contents of bit positions 52-60 of the register, with seven rightmost zeros appended, are the byte offset from the beginning of the compression dictionary to the leftmost byte of the entropy descriptor. When the entropy encoding option is specified during compression, the leftmost byte of the entropy encoding symbol translation table is located at the byte offset plus 32. Symbol translation and order-preservation cannot be specified during expansion, and the contents of bit positions 52-60 are ignored during expansion.

The contents of the registers just described and also of general register 0 are shown in Figure 7-200 on page 7-171.

Bit 55 (E) of general register 0 specifies the compression operation if zero or the expansion operation if one.

When the CMPSC-enhancement facility is installed, bit 46 of general register 0 is the zero-padding (ZP) control. When the ZP control is zero, zero padding of the first operand is not performed. When the ZP control is one, zero padding of the first operand may be performed, as described below. When the CMPSC-enhancement facility is not installed, bit 46 of general register 0 is ignored.

Bit 44 (EE) of general register 0 is the entropy-encoding-option bit; this bit is only meaningful when the entropy-encoding-compression facility is installed. Bit 45 (OP) of general register 0 is the order-preservation-option bit; this bit is only meaningful when the order-preserving-compression facility is installed. Bit 47 (ST) of general register 0 is the symbol-translation-option bit. When bit 44 (EE) is one, bit 45 (OP) must be zero; otherwise results are unpredictable.

During compression when bits 45 and 47 are both zero, the operation produces indexes, called index symbols, to compression-dictionary entries that represent character strings, and the operation then places the index symbols in the compressed-data operand. During compression when bit 45 is zero and bit 47 is one, the operation still produces index symbols but then translates the index symbols to other symbols, called interchange symbols, that it then places in the compressed-data operand. This symbol translation is done by using the symbol-translation table specified by the address and offset in general register 1. During compression when bit 45 is one, bit 47 is ignored, and a modified form of symbol translation is done by using the symbol-translation table. Bits 45 and 47 and the offset in general register 1 are ignored during expansion. During expansion, the compressed-data operand always contains index symbols that designate entries in the expansion dictionary.

When bit 44 of general register 0 is one during compression the operation produces indexes called index symbols which are translated to codewords which are then placed in the compressed-data operand. The translation is done by using the symbol-translation table specified by the address and offset + 32

bytes. During expansion the compressed-data operand always contains codewords that are converted to index symbols by the process shown in Figure 7-205, "Entropy Encoded Expansion" on page 7-187 which are then used to designate entries in the expansion dictionary.

When bit 44 of general register 0 is zero, bits 48-51 (CDSS) of the register specify the number of bits in the index symbols or interchange symbols in the compressed-data operand, as shown in the figure. Bits 48-51 must not have any of the values 0000 or 0110-1111 binary; otherwise, a specification exception is recognized. When bit 44 (EE) of general register 0 is one, bits 48-51 (CDSS) are ignored. When order preservation, entropy encoding, and symbol-translation are not specified, bits 48-51 also specify, as shown in the figure, the number of eight-byte entries in each of the compression and expansion dictionaries, and, thus, they specify the size in bytes of each of the dictionaries. When order preservation, entropy encoding or symbol translation is specified, the compression dictionary is considered to extend to the beginning of the symbol-translation table, that is, the size in bytes of the compression dictionary is the offset in bit positions 52-60 of general register 1, with seven rightmost zeros appended. The size in bytes of the symbol-translation table is considered to be one fourth that of the compression dictionary. However, the offset in general register 1 must be at least as large as the size of the compression dictionary would be if symbol translation were not specified and the CDSS were one less than it actually is, and, when the CDSS is 0001 binary, the offset must be at least 2K bytes; otherwise, the results are unpredictable. For example, if the CDSS is 0101, the offset must be at least 32 K-bytes. When entropy encoding is specified, the compression dictionary must be at least 2K bytes and at most 64 K-bytes; otherwise, the results are unpredictable.

Bit 54 (F1) of general register 0 specifies that the compression dictionary contains format-0 sibling descriptors if the bit is zero or format-1 sibling descriptors if the bit is one. Sibling descriptors are used during the compression operation. A format-0 sibling descriptor is eight bytes at an index position in the compression dictionary. A format-1 sibling descriptor is 16 bytes, with the first eight bytes at an index position in the compression dictionary and the second eight bytes at the same index position in the expansion dictionary. During compression when bit 54 is one, an expansion dictionary is considered to immediately follow the compression dictionary specified by the address in general register 1. Bit 54 is ignored during expansion. During compression when bit 54 is one, bits 44, 45 and 47 of general register 0 must be zeros; otherwise, the results are unpredictable.

Bits 47 and 54 of general register 0 must not both be ones; otherwise, the results are unpredictable.

The unused bit positions in general register 0 are reserved for possible future extensions and should contain zeros; otherwise, the program may not operate compatibly in the future.

In the access-register mode, the contents of access register R_1 are used for accessing the first operand, and the contents of access register R_2 are used for accessing the second operand, the dictionaries, the entropy descriptor, and the symbol-translation table.

When the entropy-encoding option is not specified and the first or second operand lengths are initially zero, the dictionary, first, and second operands are not accessed and condition code 0 or 1 is set.

The operation starts at the left end of both operands and proceeds to the right. The operation is ended when the end of either operand is reached or when a CPU-determined amount of data has been processed, whichever occurs first.

During a compression operation, the end of the first operand is considered to be reached when the number of unused bit positions remaining in the first-operand location is not sufficient to contain additional compressed data. When the entropy-encoding option is specified and the first operand length is non-zero, this check can only be performed after compression is attempted and the codeword size is determined.

During an expansion operation, the end of the first-operand location is considered to be reached when any of the following conditions are met:

1. The number of unused byte positions remaining in the first-operand location is not sufficient to contain all the characters that would result from expansion of the next index symbol or codeword.
2. Immediately when the number of unused byte positions is zero, that is, immediately when the expansion of an index symbol or codeword completely fills the first-operand location.

During an expansion operation, the end of the second-operand location is considered to be reached when the next index symbol or codeword does not reside entirely within the second-operand location or if the entropy-encoding option is specified and a codeword of all zeros is encountered. The second-operand location ends at the beginning of the byte designated by the sum of the address in general register R_2 and the length in general register $R_2 + 1$, regardless of the compressed-data bit number in bit positions 61-63 of general register 1.

If the operation is ended because the end of the second operand is reached, condition code 0 is set. If the operation is ended because the end of the first operand is reached, condition code 1 is set, except that condition code 0 is set provided that the end of the second operand is also reached. If the operation is ended because a CPU-determined amount of data has been processed, condition code 3 is set.

At the completion of the operation, the length in general register $R_1 + 1$ is decremented by the number of complete bytes stored at the first-operand location, and the address in general register R_1 is incremented by the same amount. During compression, a complete byte is considered to be stored only if all of its bit positions contain bits of compressed data. During compression when the first bit of compressed data stored is not in bit position 0 of a byte, the bits in the byte to the left of the first bit of compressed data remain unchanged. During compression if the last byte stored does not completely contain compressed data, the bits in the byte to the right of the rightmost bit of compressed data in the byte either are unchanged or are set to zeros.

The length in general register $R_2 + 1$ is decremented by the number of complete bytes processed at the second-operand location, and the address in general register R_2 is incremented by the same amount. During expansion, a complete byte is considered to be processed only if all of its bits have been used to produce expanded data.

The leftmost bits which are not part of the address in general registers R_1 and R_2 may be set to zeros or may remain unchanged. However, in the 24-bit or 31-bit addressing mode, bits 0-31 of these registers and also of general registers $R_1 + 1$ and $R_2 + 1$ always remain unchanged.

When all of the following conditions are met, zero padding may be performed, as described below.

- The CMPSC-enhancement facility is installed.
- The zero-padding (ZP) control, bit 46 of general register 0, is one.
- Any of the following is true:
  - The end of the first operand is reached (CC1).
  - The end of the second operand is reached (CC0).
  - A CPU-determined amount of data has been processed (CC3).
  - A codeword consisting of all zero bits is found in the middle of the compressed operand (CC0).
- The last byte of compressed or expanded data does not coincide with the rightmost byte of a model-dependent integral storage boundary.
- The remaining length in general register $R_1 + 1$ is sufficient to store up to the model-dependent integral storage boundary.

It is model dependent whether zero padding is performed. When none of the conditions listed above are met, zero padding is not performed.

Zero padding consists of storing zeros to the right of the last compressed or expanded byte, up to a model-dependent integral storage boundary. The address in general register R_1 and the length in general register $R_1 + 1$ are not updated to account for the zero padding. PER storage-alteration events are recognized for stores that occur as a result of zero padding. The model-dependent integral storage boundary is no larger than 4,096.

When the R_1 and R_2 fields do not designate general register 0, the handling of general register 1 is as follows. The bit number of the bit following the last bit of compressed data processed is placed in bit positions 61-63 of general register 1, and bits 52-60 of the register and the leftmost bits which are not part of the address in the register may be set to zeros or may remain unchanged, except that when one or both of the original length values are so small that no compressed data can be processed, all bits in the register may remain unchanged. However, when bits 44, 45 or 47 of general register 0 are one, bits 52-60 of general register 1 always remain unchanged. Also, in

the 24-bit or 31-bit addressing mode, bits 0-31 of the register always remain unchanged.

If (a) the operands overlap one another, (b) the first operand overlaps the dictionaries, entropy descriptor, or the symbol-translation table in storage in any way, or (c) either the R_1 or the R_2 field designates general register 0, the results are unpredictable.

The compression dictionary is a tree of entries in which a parent entry may have child entries. Each entry represents one or more characters called extension characters, and it also represents a character symbol consisting of the extension characters represented by the entry preceded by the extension characters represented by all of the entry's ancestors.

When order preservation is not specified, the compression process uses an unordered comparison algorithm: the children of a parent are not considered to be in any particular order, and the next characters from the string being compressed are compared against the extension characters represented by the children in the left-to-right order of the children until either a match is found or all of the children have been processed. (However, see Appendix D, “Restriction on Identical Child and Sibling Characters” for a restriction on identical first extension characters of children of the same parent.) The result of the compression process is the index symbol that designates the last matched entry, which is the parent entry if no match was found on any child or there are no children.

The order-preservation option causes the compression process to use an ordered comparison algorithm: the children of a parent are considered to be ordered such that the string of one or more extension characters represented by any child is always earlier in the collating sequence than the string of one or more extension characters represented by the next child of the same parent, that is, the children are considered to be in collating-sequence order. The next characters from the string being compressed are compared against the extension characters represented by the children in the left-to-right order of the children until any of the following is true: (1) a match is found on an entry without children, or a match is found on an entry with children but there is not another character in the string; (2) the next characters from the string have a collating-sequence value less than that of the extension characters represented by a child of the last matched parent; or (3) the next characters from the string have a collating-sequence value greater than that of the extension characters represented by the last child of the last matched parent. The result of the compression process in each of the three cases is as follows:

Case	Resulting Index Symbol
1	If the process ends because of a match on an entry without children or because of a match on an entry when there is not another character in the string, the result is the index symbol that designates the entry.
2	If the process ends because the next characters of the string have a value less than that of the characters represented by a child, the result is the index symbol that designates the child.
3	If the process ends because the next characters of the string have a value greater than that of the characters represented by the last child, the result is the index symbol that designates the last child.

Note that Case 2 includes the case when the next character of the string match the leftmost extension characters represented by a child but there are not as many next characters remaining in the string as there are additional extension characters represented in the child.

When symbol translation is specified, the symbol-translation table consists of two-byte entries, and an entry contains an interchange symbol in the rightmost bit positions of the entry. The length of the interchange symbol is specified by bits 48-51 of general register 0. The left-hand bits that are not part of the interchange symbol in a symbol-translation-table entry must be zeros; otherwise, the results are unpredictable.

To translate an index symbol to an interchange symbol, the index symbol is multiplied by 2 and then added to the address of the beginning of the symbol-translation table to locate an entry in the table, and then the interchange symbol is obtained from the entry.

When order preservation is specified, the index symbol produced by the compression process is used to locate a symbol-translation-table entry containing an interchange symbol just as when the symbol-translation option is specified. The resulting interchange symbol that is placed in the compressed-data operand depends on which of the above three cases the

index symbol was produced. The resulting interchange symbol is as follows:

Case	Resulting Interchange Symbol
1	The interchange symbol in the entry.
2	The interchange symbol in the entry, minus one; that is, one is subtracted from the value of the interchange symbol that is in the entry to form the interchange symbol that is stored.
3	The interchange symbol in the entry, plus one; that is, one is added to the value of the interchange symbol that is in the entry to form the interchange symbol that is stored.

In the above, the result of subtracting one from an interchange symbol consisting of all zero bits is an interchange symbol consisting of all one bits, and the result of adding one to an interchange symbol consisting of all one bits is an interchange symbol consisting of all zero bits.

When entropy encoding is specified, the index symbol produced by the compression process is used to locate a two-byte symbol-translation-table entry containing a codeword. The size of the codeword is determined and that number of leftmost bits of the symbol-translation-table entry are placed in the compressed data operand. The size of the codeword can be determined by the process shown in Figure 7-206, “Entropy Encoded Compression” on page 7-188.

The execution of the instruction is interruptible. When an interruption occurs, other than one that follows termination, the contents of the registers designated by the R_1 and R_2 fields and of general register 1 are updated the same as upon normal completion of the instruction, so that the instruction, when reexecuted, resumes at the point of interruption. The condition code is unpredictable.

For operands longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the current location being processed. Access exceptions may be recognized for all locations in the dictionaries and symbol-translation table if those areas are specified to be used and even if the locations would not be used during the operation. Access exceptions are not recognized for an operand, the dictionaries, or the symbol-translation table if the R field associated with that operand is odd. Also, when the R_1 field is odd, PER storage-alteration events are not recognized, and no change bits are set.

If an access exception is due to be recognized for either of the operands or for a dictionary, entropy descriptor, or the symbol-translation table, the result is that either the exception is recognized or condition code 3 is set. If condition code 3 is set, the exception will be recognized when the instruction is executed again to continue processing the same operands, assuming that the exception condition still exists.

During compression, regardless of whether the exception is recognized or condition code 3 is set, a nullifying access-exception condition or a suppressing DAT-protection exception conditionis handled so that an index symbol is generated only if it is the one that would result if there were no access-exception condition.

During compression or expansion, regardless of whether the exception is recognized or condition code 3 is set, a nullifying or suppressing access-exception condition may result in data having been stored at the first-operand location at or to the right of the location designated by the final address in general register R_1, which result is not true nullification or suppression. The amount of data stored depends on the reason for the access-exception condition. If the condition is due to a reference to a dictionary or the symbol-translation table, up to 4K bytes of data may have been stored at or to the right of the location designated by the final address. If the condition is due to a reference to the first or second operand, part of one index or interchange symbol, during compression, or part of one character symbol, during expansion, may have been stored at or to the right of the location designated by the final address. In all cases, the storing will be repeated when the instruction is executed again to continue processing the same operands.

If the end of the first operand is reached and an access exception is due to be recognized for the second operand, it is unpredictable whether condition code 1 is set or the access exception is recognized.

During expansion when the expansion dictionary is not logically correct, unusual storing may occur as described in Appendix D, “Expansion Process”. The results of an access exception in this case may not be true nullification or suppression.

Special Conditions

A specification exception is recognized for any of the following conditions:

1. Either the R_1 or R_2 fields contain an odd-numbered register.
2. Bits 48-51 of general register 0 contain any value not in the range of 0001-0101 binary and bit 44 of general register 0 is zero.

During compression of each character symbol, either the characters in the symbol or the dictionary character entries (not sibling descriptors) representing characters of the symbol are counted, and a general-operand data exception is recognized if this count becomes too large. The count can reach at least 260 without the exception being recognized.

During compression, the number of child characters or sibling characters processed during the processing of each parent entry are counted, and a general-operand data exception is recognized if this count becomes too large. The count can reach at least 260 without the exception being recognized. That is, a parent must not have more than 260 children; otherwise, a general-operand data exception may be recognized. Extension characters are not counted toward the maximum number of child and sibling characters until a child becomes a parent. When the examine child bit is zero, and a match is encountered, further matches are stopped and the child does not become a parent.

During expansion of each character symbol, either the characters in the symbol or the dictionary entries representing characters of the symbol are counted, and a general-operand data exception is recognized if this count becomes too large. If the characters in the symbol are counted, the count can reach at least 260 without the exception being recognized. If the dictionary entries representing characters of the symbol are counted, the count can reach at least 127 without the exception being recognized.

Certain error conditions in the dictionaries cause a general-operand data exception to be recognized and the operation to be either suppressed or terminated. Some of these error conditions are described in the sections “Expansion Process”, “Results of Dictionary Errors”, “Compression Dictionary”, and “Expansion Dictionary” in Appendix D.

When entropy encoding is specified, the entropy descriptor must satisfy the following properties otherwise a general-operand data exception is recognized.

1. The sum of the sixteen entries must add up to the number of dictionary entries plus one.
2. There is no arithmetic underflow or overflow when performing bounds calculations see Figure 7-204, “Entropy Descriptor Validation” on page 7-187.

During compression, when entropy encoding is specified, a symbol-translation-table entry with a zero value or an entry not padded on the right with zeros will cause a general-operand data exception to be recognized.

Resulting Condition Code:

0 End of second operand reached
1 End of first operand reached and end of second operand not reached
2 --
3 CPU-determined amount of data processed

Program Exceptions:

- Access (fetch, operand 2, dictionaries, and symbol-translation table; store, operand 1)
- Data with DXC 0, general operand
- Specification
- Transaction constraint

Programming Notes:

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the operation. The program need not determine the amount of data processed.
2. During compression when a nullifying access exception is due to be recognized, an index symbol is generated only if it is the one that would result if there were no access-exception condition. The result of this is that compression of the same expanded data by means of one or more executions of the instruction and by using the same dictionary always results in the same compressed data. That is, (1) the best possible matches in the dictionary are always found for the characters in the second operand, or else the execution is ended by either setting CC3 or rec-

ognizing the exception, and (2) the results of compression are repeatable (although possibly by means of a different number of executions of the instruction) and predictable.

For example, if the next characters of the string being compressed are ABC, and dictionary entry A has a child B, which has a child C, the normal operation is to compress ABC as a single index symbol, but if the C of the string is in the first byte of an invalid page (a page-translation exception is due to be recognized), no index symbol is generated. More specifically, an index symbol corresponding to the character symbol AB is not generated because this is not the index symbol that would be generated if the access-exception condition did not exist.

In the above, "best possible match" refers only to the characters in the second operand. In the example above, if the next two characters of the second operand (the string) are AB, and these are the last two characters of the second operand, the best possible match is on AB, even though there could be a match on ABC if the second operand included one more byte containing C.

Expansion is normally always repeatable. An index symbol is always expanded to exactly the character symbol it represents unless an exception that causes termination is recognized.

3. During expansion, if at least one unused byte position remains in the first operand location, COMPRESSION CALL may completely process the next index symbol in the second operand before it determines that the first-operand location does not have sufficient unused byte positions to contain the expanded data that would result from the next index symbol. If that next index symbol causes encountering of bad dictionary entries, the result can be either a data exception or condition code 1.

   COMPRESSION CALL immediately sets condition code 1 when processing of an index symbol exactly fills the first-operand location, except that it sets condition code 0 if the end of the second-operand location also has been reached. Immediately setting condition code 1 has the advantage that data can be compressed using one dictionary and then followed immediately, possibly on a bit boundary, by a different type of data compressed using another dictionary. The compressed data can be successfully expanded if, during the expansion of the data compressed using the first dictionary, the length of the first-operand location is specified to be exactly the length of the expanded data that will be produced. Condition code 1 will then be set when the first-operand location is full, at which time the specification of the dictionary can be changed in order to expand the remainder of the compressed data using the second dictionary. If the definition allowed condition code 1 not to be set, it might be attempted to expand the next index symbol, which resulted from use of the second dictionary, by means of the first dictionary, and this might cause recognition of a general-operand data exception. For example, the next index symbol, which properly designates a character entry in the second dictionary, might designate the second half of a format-1 sibling descriptor in the first dictionary, and that second half might begin with a character, such as 0 (F0 hex), that would appear to be an invalid partial symbol length in a character entry.

4. A nullifying access-exception condition due to a reference to a dictionary or the symbol-translation table may result in the storing of data at or to the right of the location designated by the final address in general register R_1. This storing and the processing needed to produce the data stored will be repeated when COMPRESSION CALL is executed again to continue processing the same operands. The repeated processing will reduce the performance of the instruction execution, and it should be avoided by ensuring that the environment in which the program is executed is one in which page-translation-exception conditions for the dictionaries and symbol-translation table are infrequent.

5. Following is an example of how the compressed-data bit number (CBN) is used and set. In this example:

   - The operation is an expansion operation.
   - The CDSS in general register 0 is 0010 binary. Therefore, there are 1K entries in the expansion dictionary, and the length of an index symbol is 10 bits.
   - The second operand (compressed-data operand) begins at location 6000 hex and has a length of five bytes. The initial CBN is

7. Therefore, there are three index symbols to be expanded, and the final CBN will be 5.

- The compressed data beginning at location 6000 hex is 0081FF9FF8 hex. Therefore, the three index symbols are 103, 3FC, and 3FF hex.
- The first operand (expanded-data operand) begins at location 5000 hex and has a length of 64 bytes. The three index symbols are expanded to a total of 14 bytes of expanded data.

The following figure shows the initial and final contents of general registers R_1, $R_1 + 1$, R_2, and $R_2 + 1$, the contents of locations 6000-6004 hex in binary, and the way a cursor corresponding to the CBN is advanced during the expansion operation.

Register	Initial Contents in Hex	Final Contents in Hex
R_1	5000	500E
R_1+1	40	32
R_2	6000	6004
R_2+1	5	1

Contents of Locations 6000-6004 Hex in Binary

```
00000000 10000001 11111111 10011111 11111000
       ↑               ↑        ↑        ↑
   Initial                                Final
   CBN                                    CBN
   (7)                                    (5)
```

6. The reason for allowing a parent to have no more than 260 children is as follows. The parent can contain five identical child characters. Then, 255 different sibling characters are possible — all of these must be different from the child characters and each other, or else they may be wasted (never matched against), depending on the implementation. Thus, every possible child is permitted.

7. Symbol translation is for use by VTAM. VTAM will begin by doing compression by means of software and an adaptive dictionary. When the adaptive dictionary has matured such that the degree of compression becomes sufficiently good (crosses some threshold), VTAM will "freeze" (stop adapting) its dictionary, inform the other end of the session to freeze also, transform its adaptive dictionary to the dictionary form used by COMPRESSION CALL, and then use COMPRESSION CALL to continue on with the compression. The other end of the session can continue to use its frozen adaptive dictionary.

   Following is clarification about the STT offset. Assume VTAM uses a 4K-entry adaptive dictionary, which is the largest size VTAM uses. All of the entries in this dictionary correspond to character symbols because there are no sibling descriptors in the VTAM dictionary. The VTAM dictionary cannot map one-to-one to a COMPRESSION CALL dictionary because the latter requires that some of the entries be sibling descriptors. Therefore, VTAM must have an 8K-entry dictionary for use in the basic compression operation. Only the first hundred or so entries in the second 4K of the 8K need to be used, and these entries compensate for (take the place of) the entries in the first 4K that must be sibling descriptors. The STT can and should, to save space, begin immediately after those hundred or so entries in the second 4K. In this example, the index symbols will be 13 bits but will be transformed to 12-bit interchange symbols.

8. A program may place the dictionaries in pages that are managed by means of chaining fields at their beginnings. In this case, either the parts of a dictionary have to be moved to be compacted into contiguous locations or there have to be holes in the dictionaries. The definition of COMPRESSION CALL contains nothing explicitly to support holes. However, assuming there is at least one character that never appears in the expanded data, that character can be used as a child character in a parent entry or as a sibling character in a sibling descriptor to specify a child or children that will never be referenced, thus creating a hole.

9. The references to the operands, dictionaries, and symbol-translation table for COMPRESSION CALL may be multiple-access references. (See "Storage-Operand Consistency" on page 5-125.)

10. Setting the ZP control to one may provide improved performance on models in which the CMPSC-enhancement facility is installed.

11. Zero padding is not necessarily performed in every case where it is permissible.

12. Figure 7-201 on page 7-182 and Figure 7-203 on page 7-186 show possible forms (not the only possible forms) of the compression process

(without order preservation), the order preservation compression process and expansion processes. The figures do not show testing for or the results of dictionary errors.

13. As observed by this CPU, other CPUs, and the I/O subsystem, inconsistent results may be briefly stored in the first-operand location. See the section “Effects of CPU Retry” on page 11-3 for further details.

14. When the entropy-encoding option is specified the minimum dictionary size is 128-bytes or 16 entries. The dictionary must also be a multiple of 16 entries.

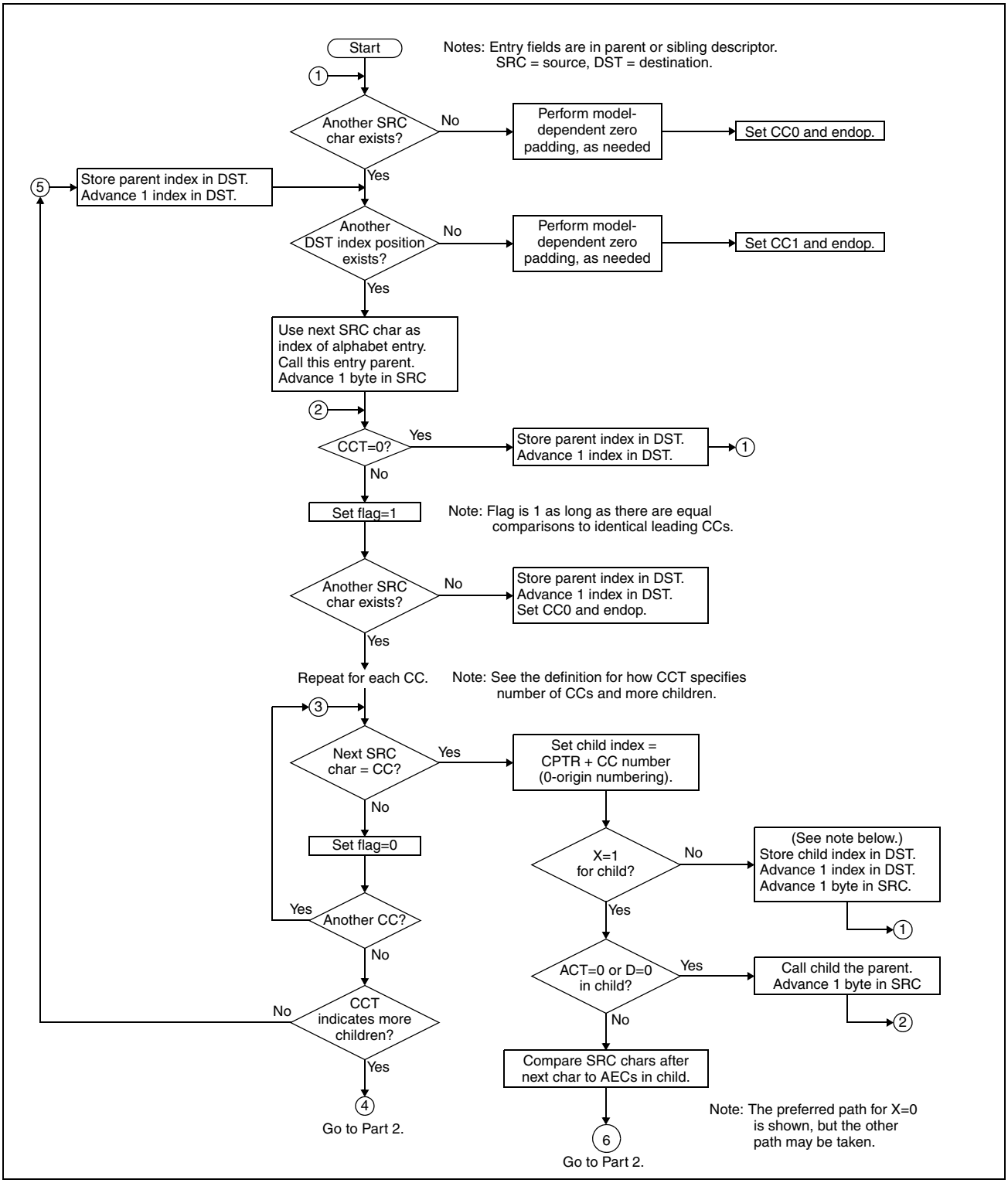


Figure 7-201. Compression Process (Part 1 of 2)

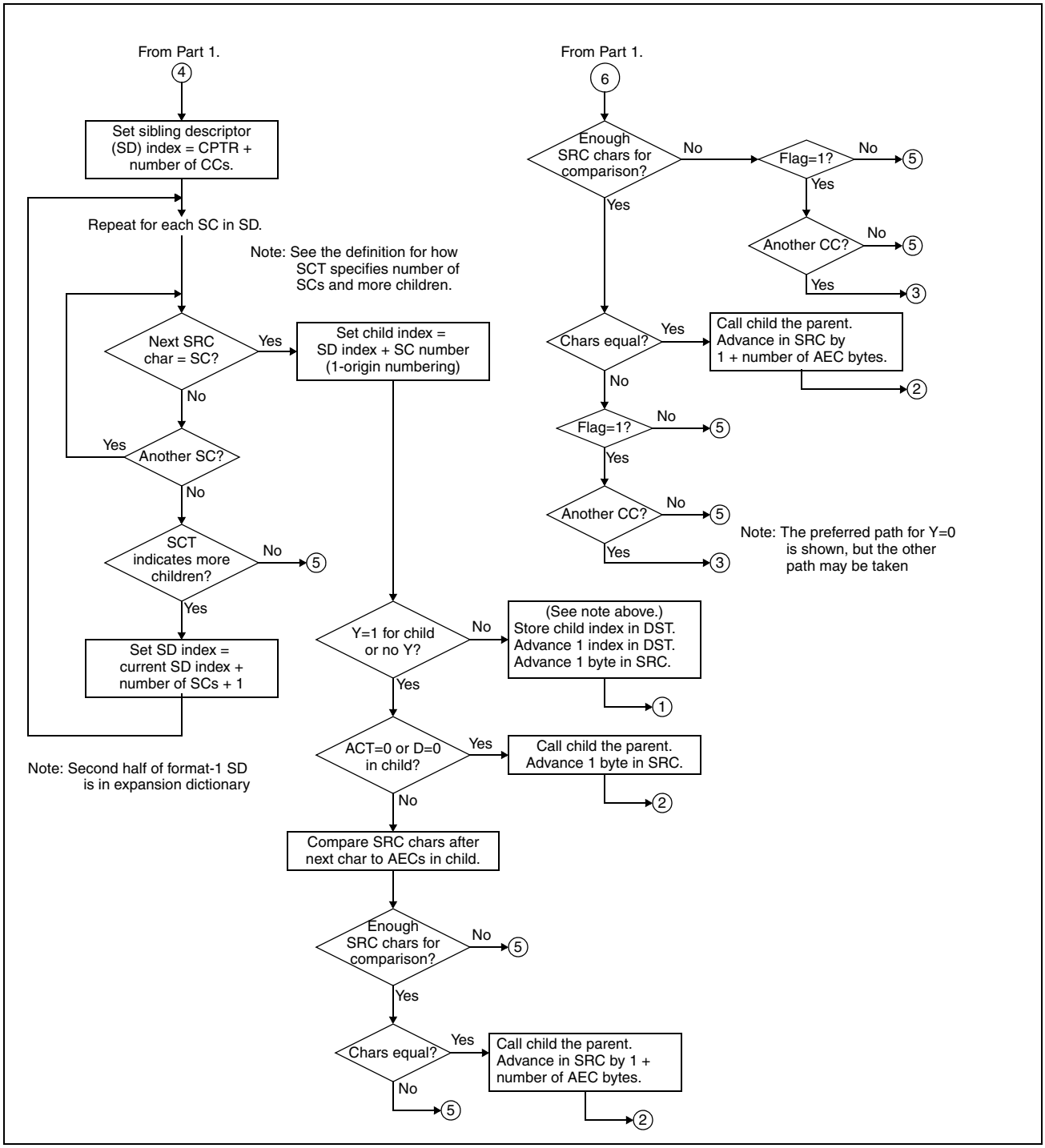


Figure 7-201. Compression Process (Part 2 of 2)

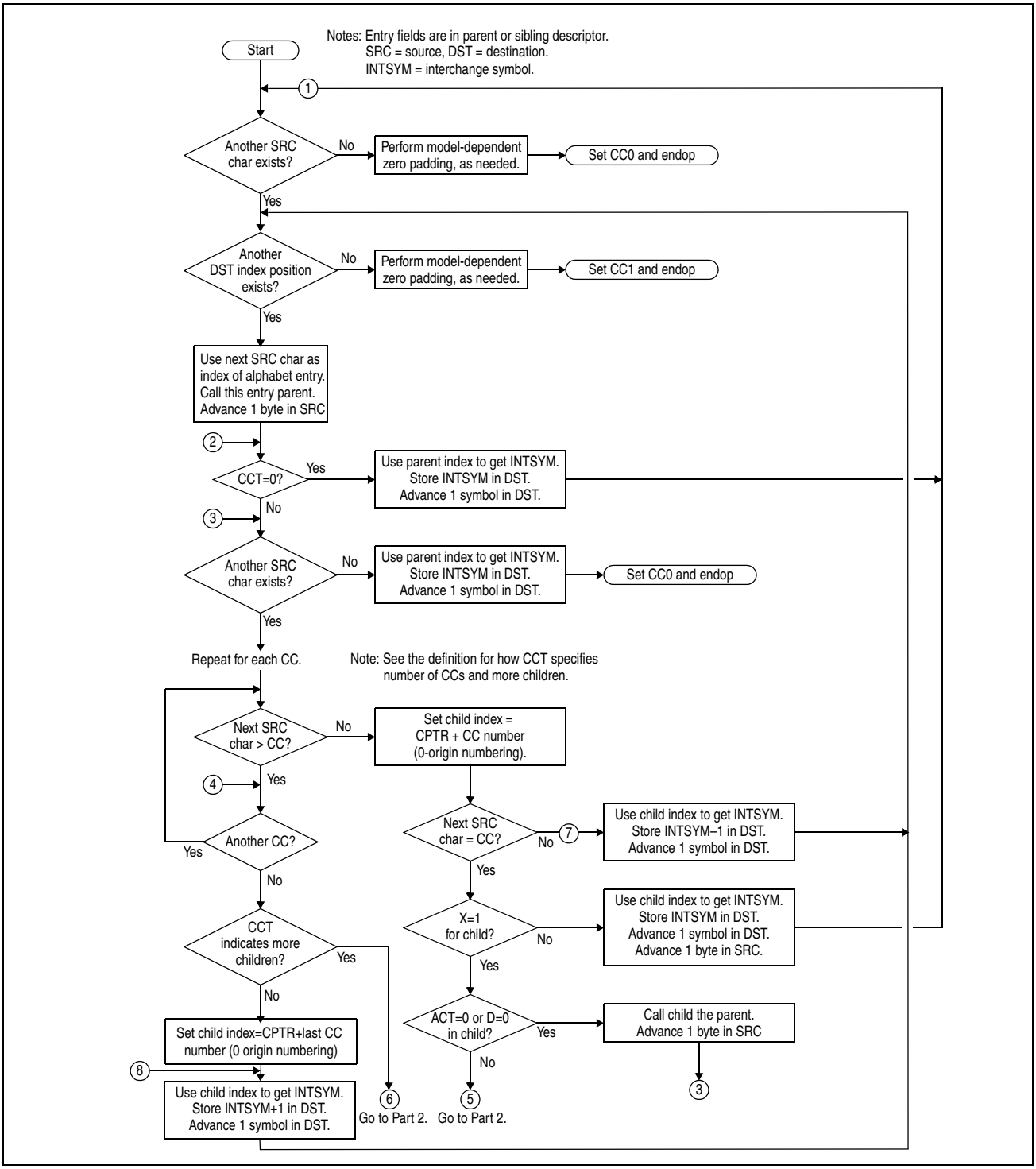


Figure 7-202. Order-Preservation Compression Process (Part 1 of 2)

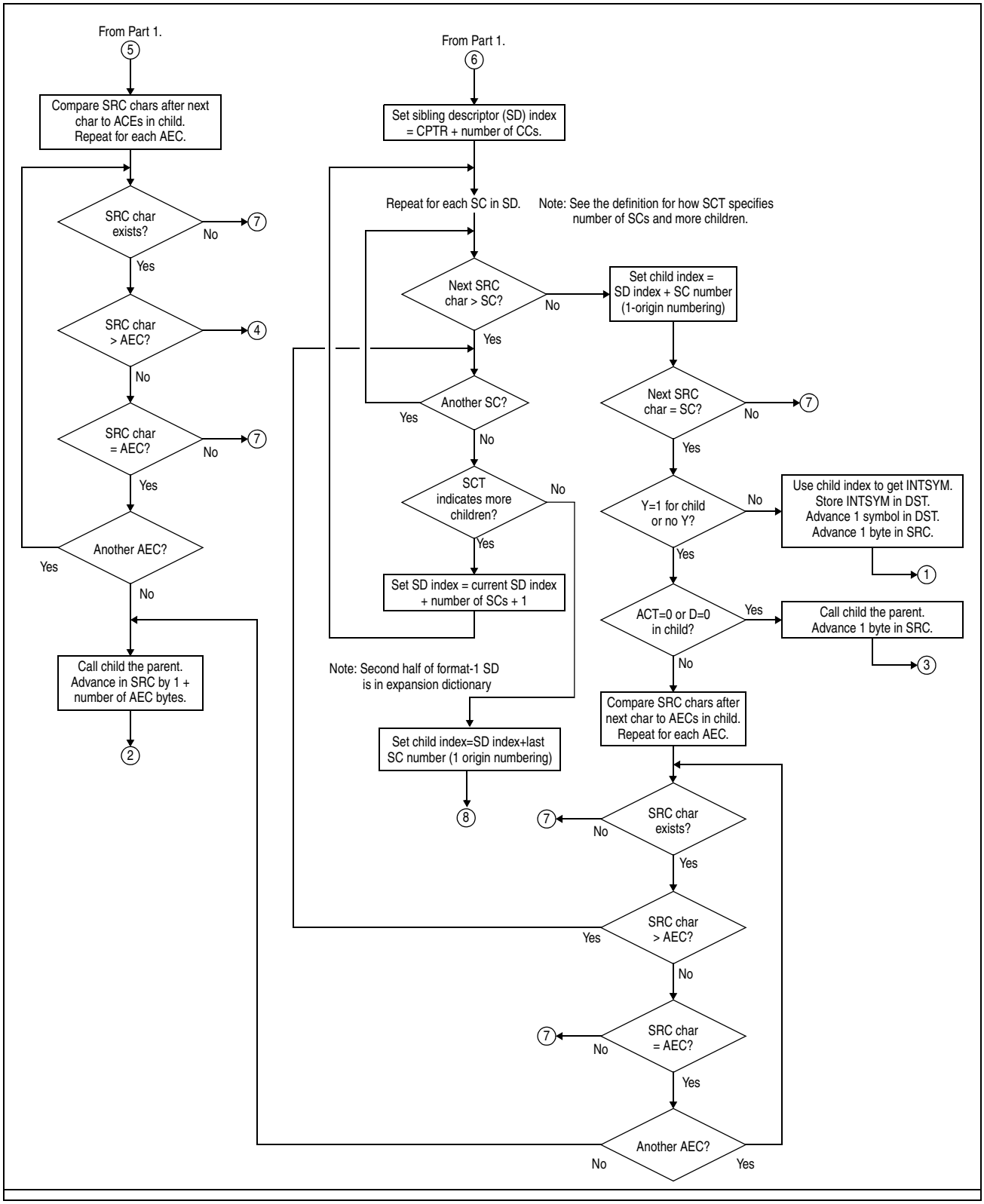


Figure 7-202. Order-Preservation Compression Process (Part 2 of 2)

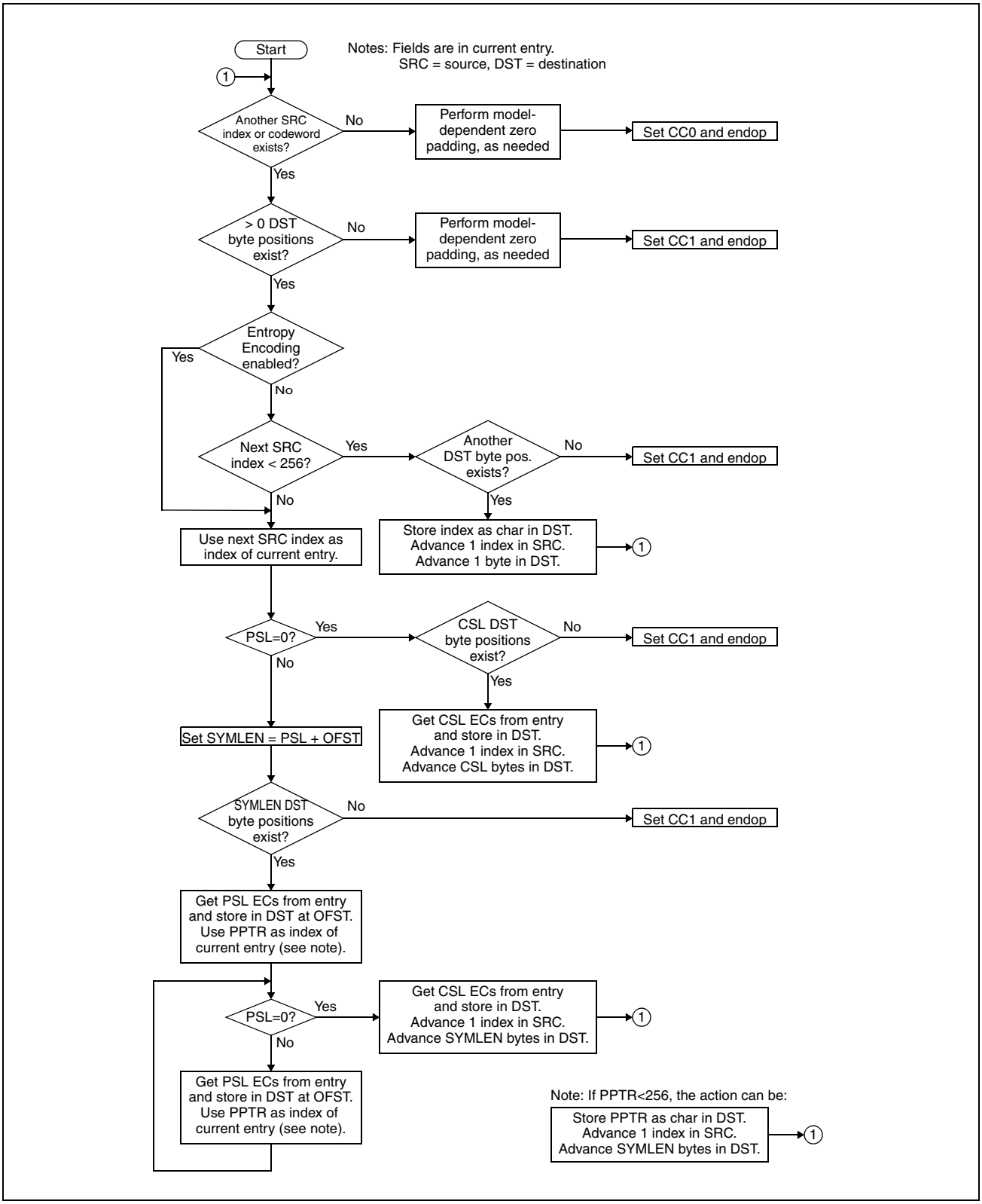


Figure 7-203. Expansion Process

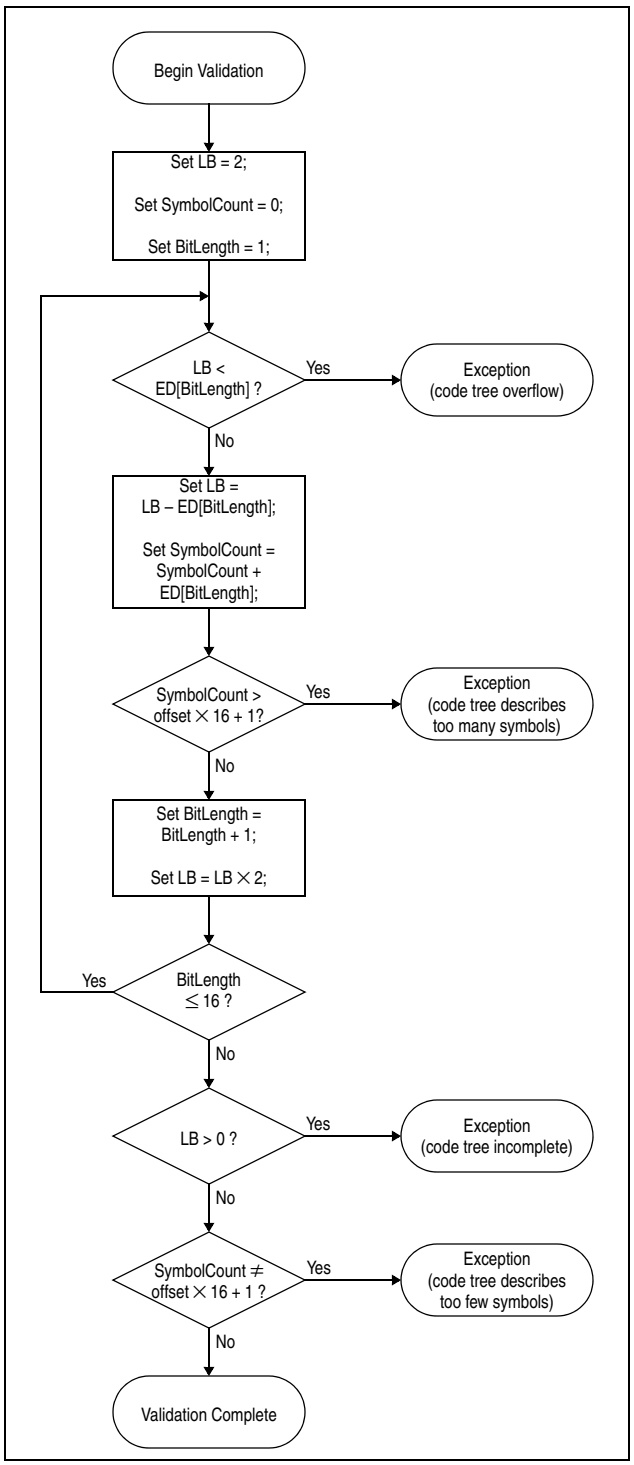


Figure 7-204. Entropy Descriptor Validation

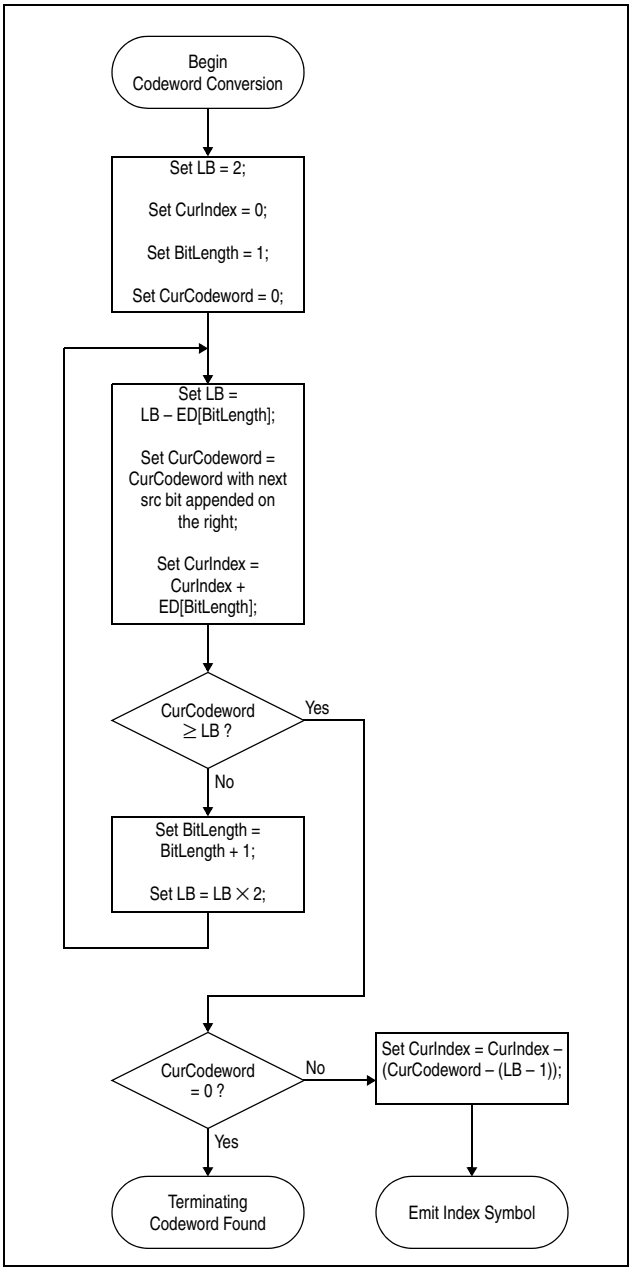


Figure 7-205. Entropy Encoded Expansion

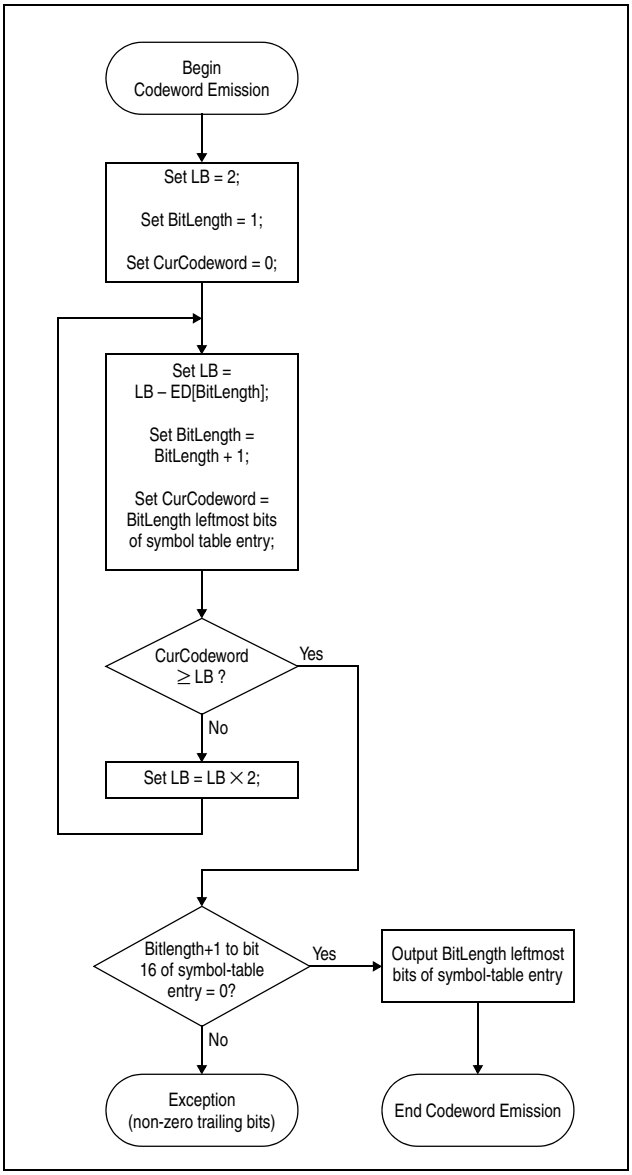
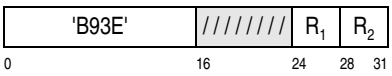


Figure 7-206. Entropy Encoded Compression

COMPUTE INTERMEDIATE MESSAGE DIGEST

KIMD R_1,R_2 [RRE]



A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction and the R_1 field are reserved and should contain zeros; otherwise, the program may not operate compatibly in the future.

Bit positions 57-63 of general register 0 contain the function code. Figure 7-207 show the assigned function codes for COMPUTE INTERMEDIATE MESSAGE DIGEST and COMPUTE LAST MESSAGE DIGEST, respectively. All other function codes are unassigned. Bit 56 of general register 0 must be zero; otherwise, a specification exception is recognized. All other bits of general register 0 are ignored.

The assigned function codes are enumerated in Figure 7-207. All other function codes are unassigned.

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
0	KIMD-Query *	16	—
1	KIMD-SHA-1 *	20	64
2	KIMD-SHA-256 *	32	64
3	KIMD-SHA-512 *	64	128
32	KIMD-SHA3-224	200	144
33	KIMD-SHA3-256	200	136
34	KIMD-SHA3-384	200	104
35	KIMD-SHA3-512	200	72
36	KIMD-SHAKE-128	200	168
37	KIMD-SHAKE-256	200	136
65	KIMD-GHASH	32	16

Explanation:

— Not applicable

* Function is also defined in the ESA/390 architectural mode and the ESA/390-compatibility mode. It is unpredictable whether other function codes are available in the ESA/390-compatibility mode.

Figure 7-207. Function Codes for COMPUTE INTERMEDIATE MESSAGE DIGEST

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register 1 constitute the address.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R_2 and $R_2 + 1$ are ignored for the query function.

For all other functions, the second operand is processed as specified by the function code using an initial chaining value in the parameter block, and the result replaces the chaining value. For COMPUTE LAST MESSAGE DIGEST, the operation also uses a message bit length in the parameter block. The operation proceeds until the end of the second-operand location is reached or a CPU-determined number of bytes have been processed, whichever occurs first. The result is indicated in the condition code.

The R_2 field designates an even-odd pair of general registers and must designate an even-numbered register other than general register 0; otherwise, a specification exception is recognized.

The location of the leftmost byte of the second operand is specified by the contents of the R_2 general register. The number of bytes in the second-operand location is specified in general register $R_2 + 1$. For all functions except KIMD-query, the number of bytes in general register $R_2 + 1$ must be a multiple of the data block size for the function; otherwise, a specification exception is recognized.

As part of the operation, the address in general register R_2 is incremented by the number of bytes processed from the second operand, and the length in general register $R_2 + 1$ is decremented by the same number. The formation and updating of the address and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general register R_2 constitute the address of the second operand, and the contents of bit positions 0-39 are ignored; bits 40-63 of the updated address replace the corresponding bits in general register R_2, carries out of bit position 40 of the updated address are ignored, and the contents of bit positions 32-39 of general register R_2 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register R_2 constitute the address of the second operand, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated address replace the corresponding bits in general register R_2, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general register R_2 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register R_2 constitute the address of the second operand; bits 0-63 of the updated address replace the contents of general register R_2 and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_2 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of bit positions 32-63 of general register $R_2 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of general register $R_2 + 1$.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_2 and $R_2 + 1$, always remain unchanged.

Figure 7-208 shows the contents of the general registers just described.

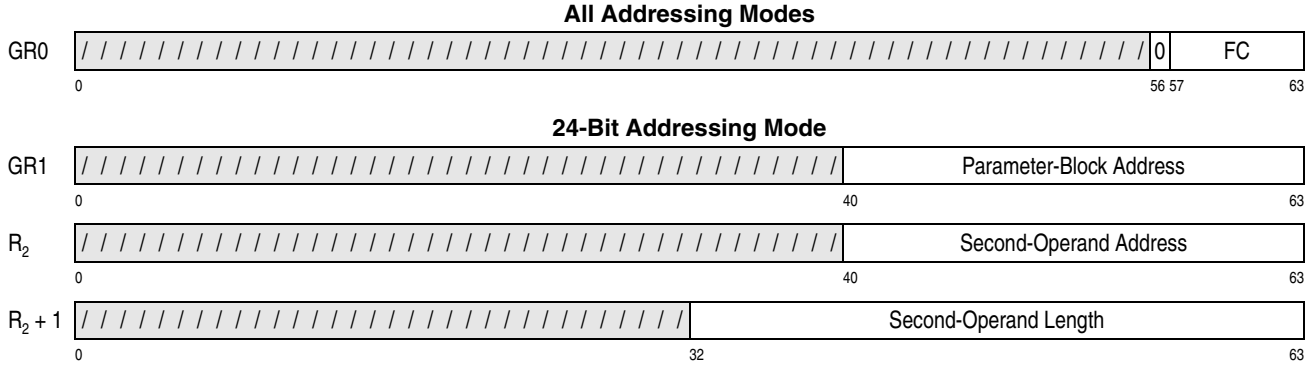


Figure 7-208. General Register Assignment for KIMD and KLMD (Part 1 of 2)

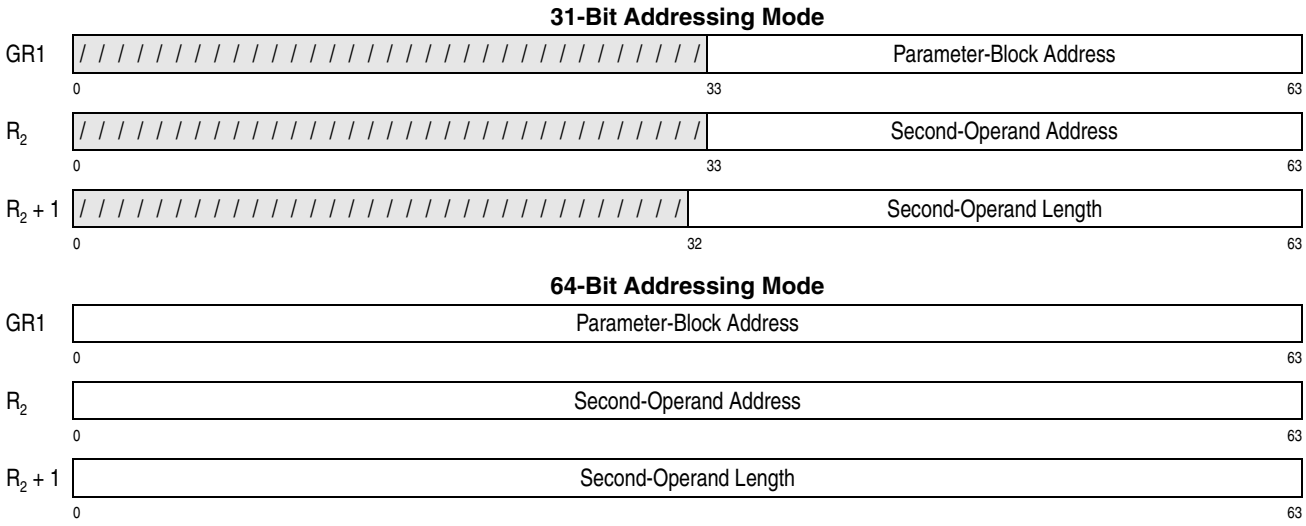


Figure 7-208. General Register Assignment for KIMD and KLMD (Part 2 of 2)

In the access-register mode, access registers 1 and R_2 specify the address spaces containing the parameter block and second operand, respectively.

The result is obtained as if processing starts at the left end of the second operand and proceeds to the right, block by block. The operation is ended when all source bytes in the second operand have been processed (called normal completion), or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

When the chaining-value field overlaps any portion of the second operand, the result in the chaining-value field is unpredictable.

Normal completion occurs when the number of bytes in the second operand as specified in general register $R_2 + 1$ have been processed.

When the operation ends due to normal completion, condition code 0 is set, the second-operand address in general register R_2 is updated, and the second-operand length in $R_2 + 1$ is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in $R_2 + 1$ is nonzero.

When the second-operand length is initially zero, the second operand is not accessed, the second-operand address and second-operand length in general registers R_2 and $R_2 + 1$, respectively, are not changed, and condition code 0 is set. The parameter block is not accessed.

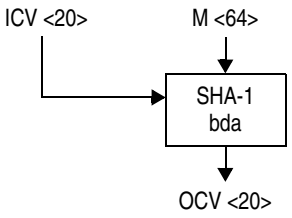
A PER storage-alteration event is recognized, when applicable, for the portion of the parameter block that is stored. A PER zero-address-detection event is recognized, when applicable, for the second-operand location and for the parameter block. When PER events are detected for more than one location, it is unpredictable which location is identified in the PER access identification (PAID) and PER ASCE ID (AI).

As observed by this CPU, other CPUs, and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

Access exceptions may be reported for a larger portion of the second operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of the second operand nor for locations more than 4K bytes beyond the current location being processed.

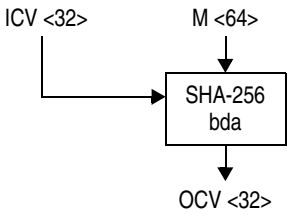
Symbols Used in Function Descriptions

The following symbols are used in the subsequent description of the COMPUTE INTERMEDIATE MESSAGE DIGEST and COMPUTE LAST MESSAGE DIGEST functions. Further description of the secure hash algorithm may be found in Reference [15.] on page xxx. Further description of the GCM (Galois/counter mode) multiplication over GF(2^{128}) may be found in Reference [17.] on page xxx. Further description of the SHA-3 and SHAKE hash algorithms and the KECCAK family of sponge functions may be found in Reference [21.] on page xxx.



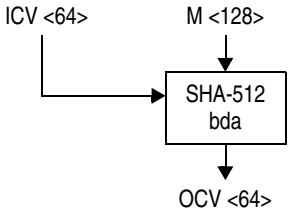
Symbol	Explanation
<n>	Length of item in bytes
bda	Block digest algorithm
ICV	Initial chaining value
M	Message block
OCV	Output chaining value

Figure 7-209. Symbol for SHA-1 Block Digest Algorithm



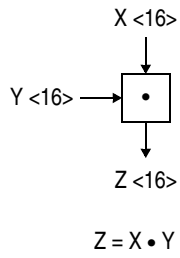
Symbol	Explanation
<n>	Length of item in bytes
bda	Block digest algorithm
ICV	Initial chaining value
M	Message block
OCV	Output chaining value

Figure 7-210. Symbol for SHA-256 Block Digest Algorithm



Symbol	Explanation
<n>	Length of item in bytes
bda	Block digest algorithm
ICV	Initial chaining value
M	Message block
OCV	Output chaining value

Figure 7-211. Symbol for SHA-512 Block Digest Algorithm



Symbol	Explanation
•	GCM multiplication operation over GF(2^{128})

Figure 7-212. Symbol For GCM Multiplication Operation Over GF(2^{128})

KIMD-Query (KIMD Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

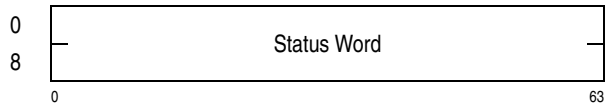


Figure 7-213. Parameter Block for KIMD-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the COMPUTE INTERMEDIATE MESSAGE DIGEST instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KIMD-Query function completes; condition code 3 is not applicable to this function.

KIMD-SHA-1 (KIMD Function Code 1)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

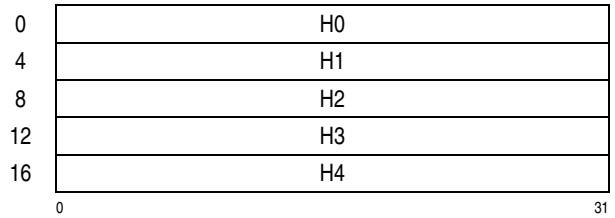


Figure 7-214. Parameter Block for KIMD-SHA-1

A 20-byte intermediate message digest is generated for the 64-byte message blocks in operand 2 using the SHA-1 block digest algorithm with the 20-byte chaining value (called H fields) in the parameter block. The generated intermediate message digest, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. The operation is shown in Figure 7-215.

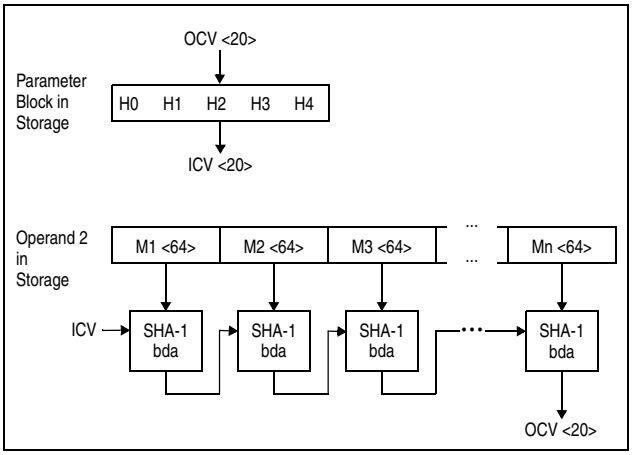


Figure 7-215. KIMD-SHA-1

KIMD-SHA-256 (KIMD Function Code 2)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

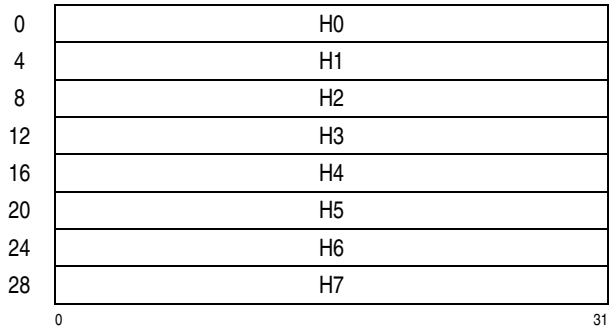


Figure 7-216. Parameter Block for KIMD-SHA-256

A 32-byte intermediate message digest is generated for the 64-byte message blocks in operand 2 using the SHA-256 block digest algorithm with the 32-byte chaining value (called H fields) in the parameter block. The generated intermediate message digest, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. The operation is shown in Figure 7-217.

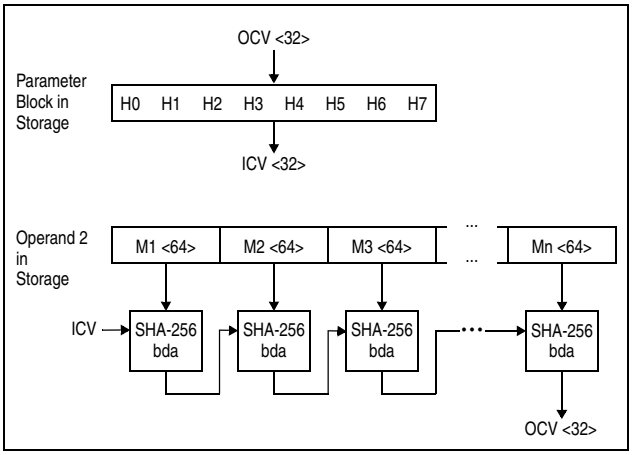


Figure 7-217. KIMD-SHA-256

KIMD-SHA-512 (KIMD Function Code 3)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

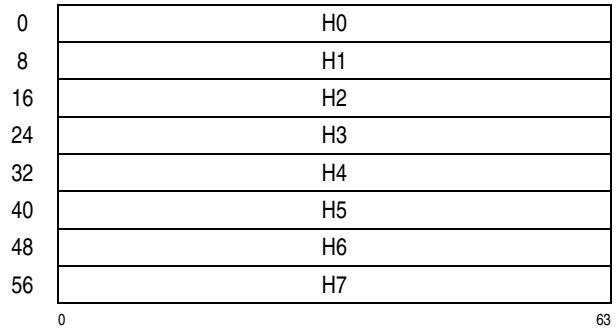


Figure 7-218. Parameter Block for KIMD-SHA-512

A 64-byte intermediate message digest is generated for the 128-byte message blocks in operand 2 using the SHA-512 block digest algorithm with the 64-byte chaining value (called H fields) in the parameter block. The generated intermediate message digest, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. The operation is shown in Figure 7-217.

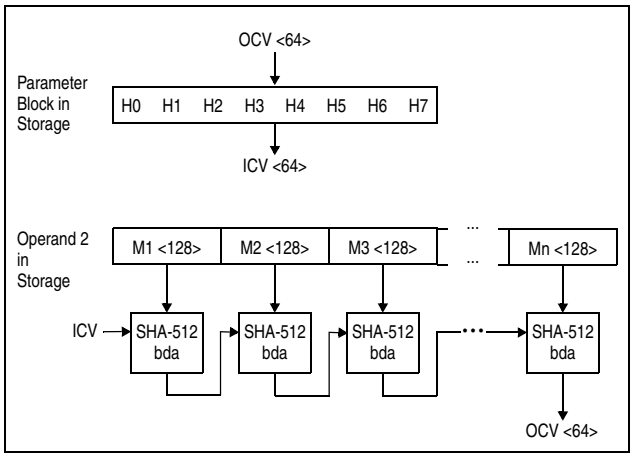


Figure 7-219. KIMD-SHA-512

KIMD-SHA3-224 (KIMD Function Code 32)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

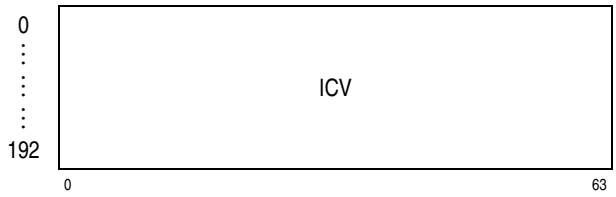
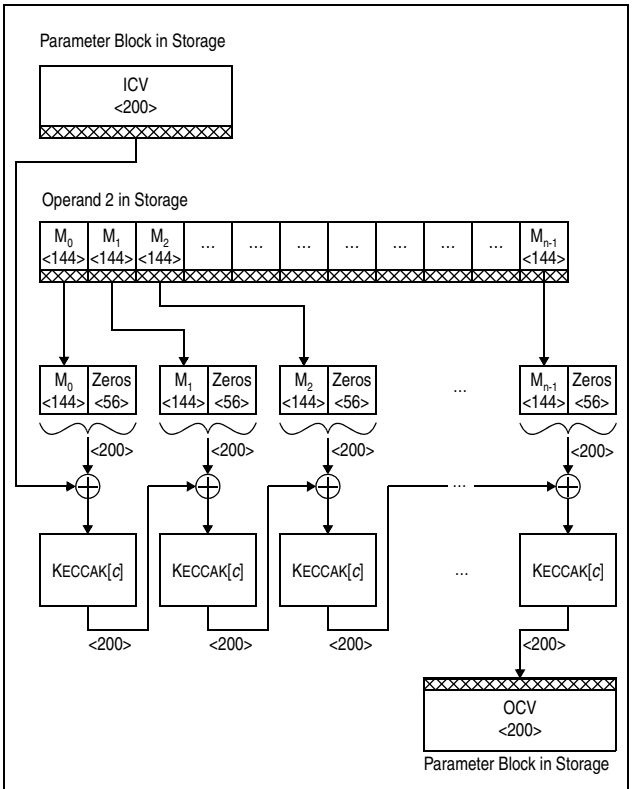


Figure 7-220. Parameter Block for KIMD-SHA3-224

The initial chaining value (ICV) represents the 1600-bit state array used by the KECCAK[*c*] functions which implement the SHA-3 algorithms described in Reference [21.] on page xxx.

A 200-byte intermediate message digest is generated for the 144-byte message blocks in operand 2 using the KECCAK[*c*] algorithm with the 200-byte initial chaining value in the parameter block. The generated intermediate message digest, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. See programming note 10 on page 7-200 regarding the numbering of the bits in the parameter block and second operand as compared with the numbering of the bits in the state array.

The operation is shown in Figure 7-221.



Explanation:

⊠	The bit positions of each byte of the parameter block and the message block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
<#>	Size of an item in bytes
ICV	Input chaining value (in the parameter block)
M	144-byte block of the second-operand message
n	Number of blocks in the second-operand (that is, the second operand length divided by the block size of 144)
OCV	Output chaining value (in the parameter block)

Figure 7-221. KIMD-SHA3-224 Processing

KIMD-SHA3-256 (KIMD Function Code 33)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

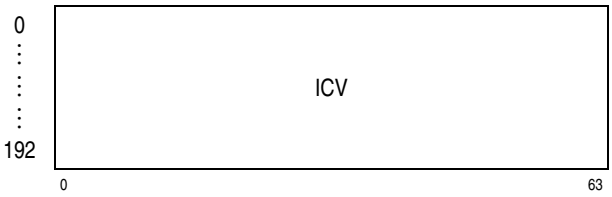
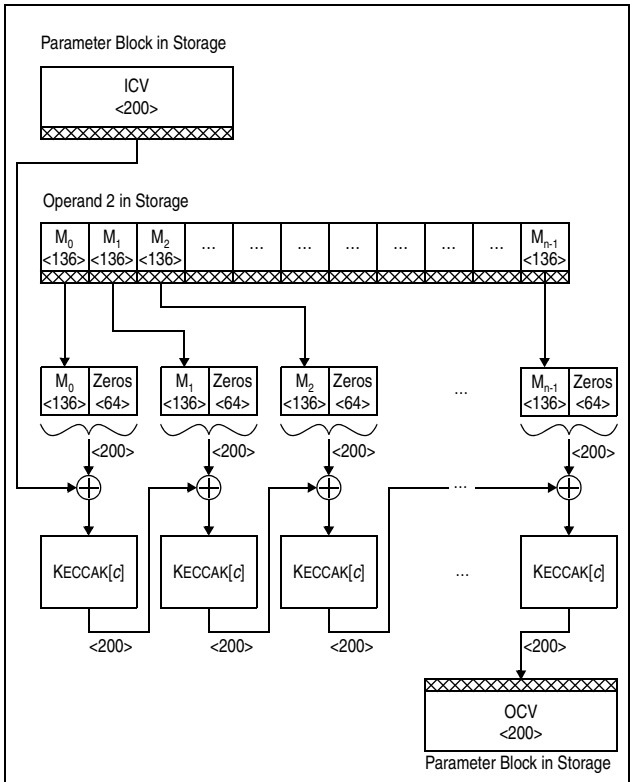


Figure 7-222. Parameter Block for KIMD-SHA3-256

The initial chaining value (ICV) represents the 1600-bit state array used by the KECCAK[*c*] functions which implement the SHA-3 algorithms described in Reference [21.] on page xxx.

A 200-byte intermediate message digest is generated for the 136-byte message blocks in operand 2 using the KECCAK[*c*] algorithm with the 200-byte initial chaining value in the parameter block. The generated intermediate message digest, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. See programming note 10 on page 7-200 regarding the numbering of the bits in the parameter block and second operand as compared with the numbering of the bits in the state array.

The operation is shown in Figure 7-223.



Explanation:

⊠	The bit positions of each byte of the parameter block and the message block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
⊕	Bitwise exclusive-OR operation
<#>	Size of an item in bytes
ICV	Input chaining value (in the parameter block)
M	136-byte block of the second-operand message
n	Number of blocks in the second-operand (that is, the second operand length divided by the block size of 136)
OCV	Output chaining value (in the parameter block)

Figure 7-223. KIMD-SHA3-256 Processing

KIMD-SHA3-384 (KIMD Function Code 34)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

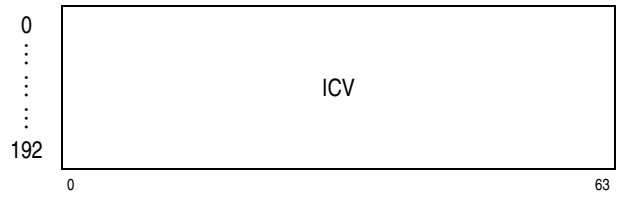
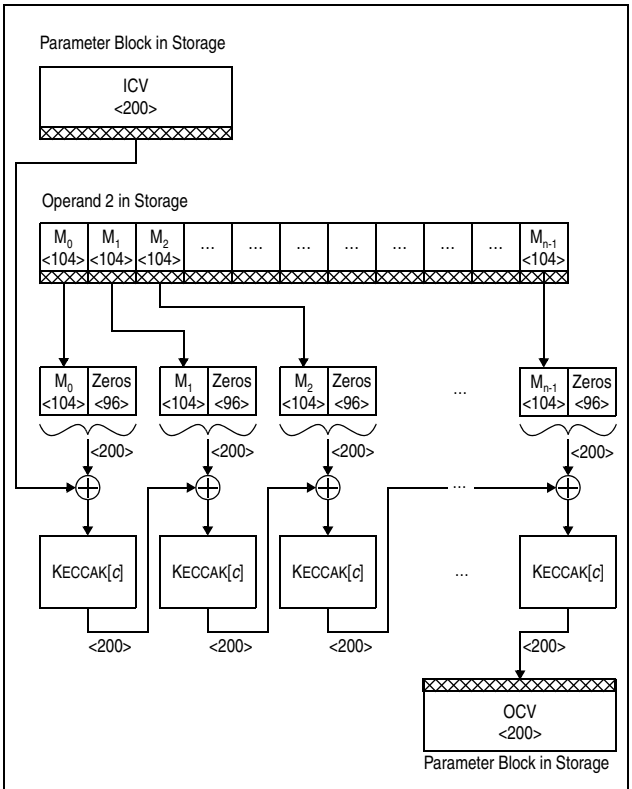


Figure 7-224. Parameter Block for KIMD-SHA3-384

The initial chaining value (ICV) represents the 1600-bit state array used by the KECCAK[*c*] functions which implement the SHA-3 algorithms described in Reference [21.] on page xxx.

A 200-byte intermediate message digest is generated for the 104-byte message blocks in operand 2 using the KECCAK[*c*] algorithm with the 200-byte initial chaining value in the parameter block. The generated intermediate message digest, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. See programming note 10 on page 7-200 regarding the numbering of the bits in the parameter block and second operand as compared with the numbering of the bits in the state array.

The operation is shown in Figure 7-225.



Explanation:

⊠	The bit positions of each byte of the parameter block and the message block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
⊕	Bitwise exclusive-OR operation
<#>	Size of an item in bytes
ICV	Input chaining value (in the parameter block)
M	104-byte block of the second-operand message
n	Number of blocks in the second-operand (that is, the second operand length divided by the block size of 104)
OCV	Output chaining value (in the parameter block)

Figure 7-225. KIMD-SHA3-384 Processing

KIMD-SHA3-512 (KIMD Function Code 35)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

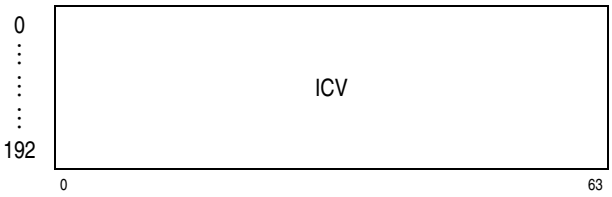
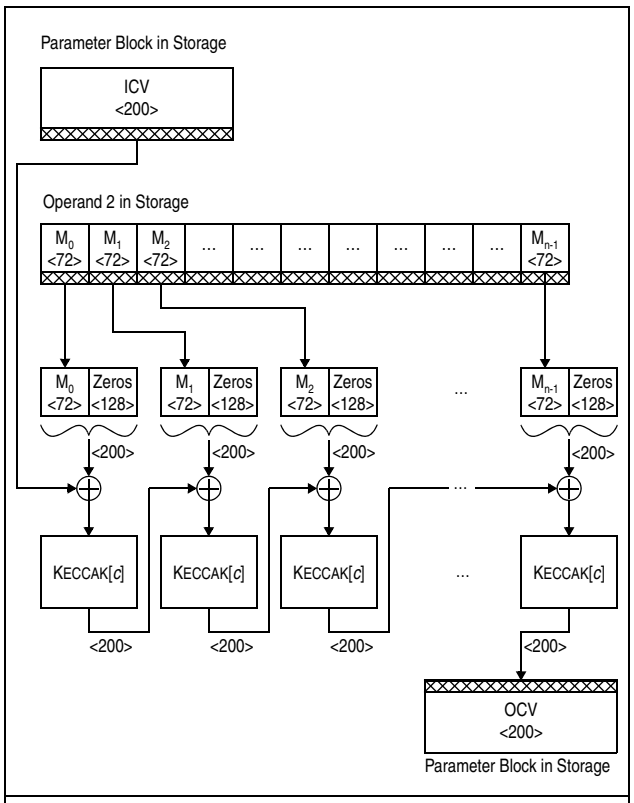


Figure 7-226. Parameter Block for KIMD-SHA3-512

The initial chaining value (ICV) represents the 1600-bit state array used by the KECCAK[*c*] functions which implement the SHA-3 algorithms described in Reference [21.] on page xxx.

A 200-byte intermediate message digest is generated for the 72-byte message blocks in operand 2 using the KECCAK[*c*] algorithm with the 200-byte initial chaining value in the parameter block. The generated intermediate message digest, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. See programming note 10 on page 7-200 regarding the numbering of the bits in the parameter block and second operand as compared with the numbering of the bits in the state array.

The operation is shown in Figure 7-227.



Explanation:

⊠	The bit positions of each byte of the parameter block and the message block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
⊕	Bitwise exclusive-OR operation
<#>	Size of an item in bytes
ICV	Input chaining value (in the parameter block)
M	72-byte block of the second-operand message
n	Number of blocks in the second-operand (that is, the second operand length divided by the block size of 72)
OCV	Output chaining value (in the parameter block)

Figure 7-227. KIMD-SHA3-512 Processing

KIMD-SHAKE-128 (KIMD Function Code 36)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

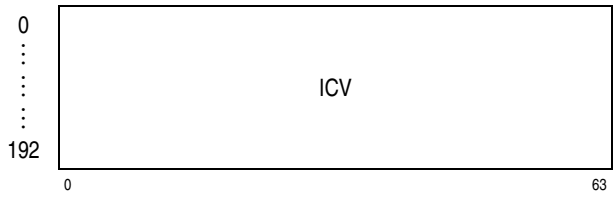
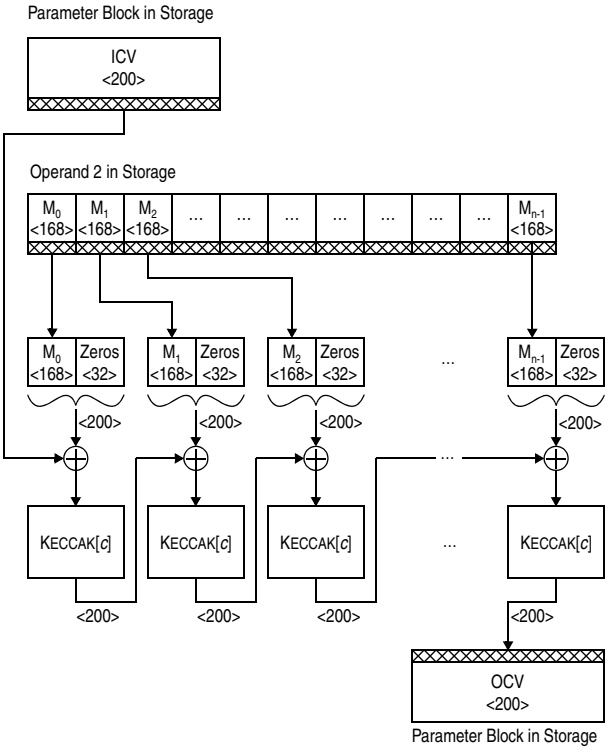


Figure 7-228. Parameter Block for KIMD-SHAKE-128

The initial chaining value (ICV) represents the 1600-bit state array used by the KECCAK[*c*] functions which implement the SHAKE algorithms described in Reference [21.] on page xxx.

A 200-byte intermediate message digest is generated for the 168-byte message blocks in operand 2 using the KECCAK[*c*] algorithm with the 200-byte initial chaining value in the parameter block. The generated intermediate message digest, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. See programming note 10 on page 7-200 regarding the numbering of the bits in the parameter block and second operand as compared with the numbering of the bits in the state array.

The operation is shown in Figure 7-229.



Explanation:

⊠	The bit positions of each byte of the parameter block and the message block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
⊕	Bitwise exclusive-OR operation
<#>	Size of an item in bytes
ICV	Input chaining value (in the parameter block)
M	168-byte block of the second-operand message
n	Number of blocks in the second-operand (that is, the second operand length divided by the block size of 168)
OCV	Output chaining value (in the parameter block)

Figure 7-229. KIMD-SHAKE-128 Processing

KIMD-SHAKE-256 (KIMD Function Code 37)

The operands, addresses, parameter-block format, and operation of the KIMD-SHAKE-256 function are identical to those of the KIMD-SHA3-256 function.

Note: A separate function code for KIMD-SHAKE-256 is defined to retain symmetry with the KLMD functions (where the operation of KLMD-SHAKE-256 differs from that of KLMD-SHA3-256).

KIMD-GHASH (Function Code 65)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the KIMD-GHASH function has the following format:

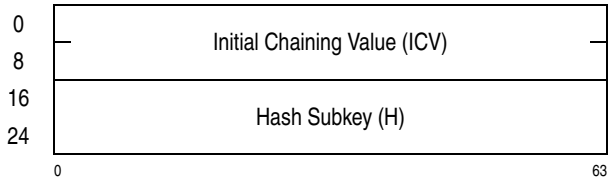


Figure 7-230. Parameter Block for KIMD-GHASH

The initial-chaining value is in byte offsets 0-15 of the parameter block, and the hash subkey is in byte offsets 16-31 of the parameter block.

The GHASH for the 16-byte message blocks (M1, M2, …, Mn) in operand 2 is computed using the hash subkey and the 128-bit chaining value.

The first message block is exclusive-ORed with the initial chaining value (ICV) in the parameter block. The result of the exclusive-OR operation and the hash subkey (H) are multiplied using a Galois/counter-mode (GCM) multiplication operation over a Galois field ($GF(2^{128})$) to produce the chaining value for the next message block. The process is repeated until all message blocks have been processed, or until a CPU-determined number of message blocks have been processed.

The GHASH, called the output chaining value (OCV), is stored into the Initial-chaining-value field of the parameter block. The operation is shown in Figure 7-231 on page 7-198.

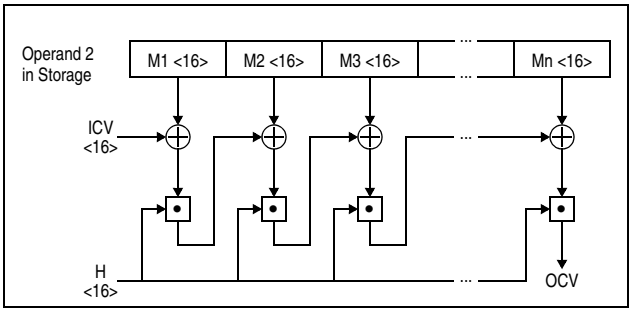


Figure 7-231. KIMD-GHASH

Special Conditions

A specification exception is recognized and no other action is taken if any of the following occurs:

1. Bit 56 of general register 0 is not zero.
2. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
3. The R_2 field designates an odd-numbered register or general register 0.
4. The second-operand length is not a multiple of the data block size of the designated function (see Figure 7-207 on page 7-188 for COMPUTE INTERMEDIATE MESSAGE DIGEST functions).

Resulting Condition Code:

0 Normal completion
1 --
2 --
3 Partial completion

Program Exceptions:

- Access (fetch, operand 2 and message bit length; fetch and store, chaining value)
- Operation (if the message-security assist is not installed)
- Specification
- Transaction constraint

1.-6.	Exceptions with the same priority as the priority of program-interruption conditions for the general case.
7.A	Access exceptions for second instruction halfword.
7.B	Operation exception.
7.C	Transaction constraint
8.	Specification exception due to invalid function code or invalid register number.
9.	Specification exception due to invalid operand length.
10.	Condition code 0 due to second-operand length originally zero.
11.	Access exceptions for an access to the parameter block or second operand.
12.	Condition code 0 due to normal completion (second-operand length originally nonzero, but stepped to zero).
13.	Condition code 3 due to partial completion (second-operand length still nonzero).

Figure 7-232. Priority of Execution: KIMD

Figure 7-232. Priority of Execution: KIMD

Programming Notes:

1. The performance of the query function (code 0) may be significantly slower than that of simply examining a location in storage. If the program needs to frequently test for the availability of a function, it should perform the query function once during initialization; subsequently it should examine the stored results of the query function in memory with an instruction such as TEST UNDER MASK.
2. Bit 56 of general register 0 is reserved for future extension and should be set to zero.
3. When condition code 3 is set, the second operand address and length in general registers R_2 and $R_2 + 1$, respectively, and the chaining-value in the parameter block are usually updated such that the program can simply branch back to the instruction to continue the operation.

   For unusual situations, the CPU protects against endless reoccurrence for the no-progress case. Thus, the program can safely branch back to the instruction whenever condition code 3 is set with no exposure to an endless loop.
4. If the length of the second operand is nonzero initially and condition code 0 is set, the registers are updated in the same manner as for condition code 3; the chaining value in this case is such that additional operands can be processed as if they were part of the same chain.
5. The instructions COMPUTE INTERMEDIATE MESSAGE DIGEST and COMPUTE LAST MESSAGE DIGEST are designed to be used by a security service application programming interface (API). These APIs provide the program with means to compute the digest of messages of almost unlimited size, including those too large to fit in storage all at once. This is accomplished by permitting the program to pass the message to the API in parts. The following programming notes are described in terms of these APIs.

6. Before processing the first part of a message, the program must set the initial values for the chaining-value field. For SHA-1, the initial hexadecimal chaining values are listed as follows:

   ```
   H0 = 6745 2301
   H1 = EFCD AB89
   H2 = 98BA DCFE
   H3 = 1032 5476
   H4 = C3D2 E1F0
   ```

   For SHA-256, the initial hexadecimal chaining values are listed as follows:

   ```
   H0 = 6A09 E667
   H1 = BB67 AE85
   H2 = 3C6E F372
   H3 = A54F F53A
   H4 = 510E 527F
   H5 = 9B05 688C
   H6 = 1F83 D9AB
   H7 = 5BE0 CD19
   ```

   For SHA-512, the initial hexadecimal chaining value are listed as follows:

   ```
   H0 = 6A09 E667 F3BC C908
   H1 = BB67 AE85 84CA A73B
   H2 = 3C6E F372 FE94 F82B
   H3 = A54F F53A 5F1D 36F1
   H4 = 510E 527F ADE6 82D1
   H5 = 9B05 688C 2B3E 6C1F
   H6 = 1F83 D9AB FB41 BD6B
   H7 = 5BE0 CD19 137E 2179
   ```

   For the SHA3 and SHAKE functions, the initial chaining value must be set to all zeros.

7. When computing a message digest, the program may not initially be aware of the total message-bit length; for example, for a message being read from an I/O device, the message-bit length may not be known until the final block is read. When computing a message digest for a message whose length is not known, or for a message where it is known that the last message block is not included in the calculation, COMPUTE INTERMEDIATE MESSAGE DIGEST may be used. When computing a message digest for a message that includes the last block, COMPUTE LAST MESSAGE DIGEST must be used.

8. The SHA-224 algorithm is the same as the SHA-256 algorithm, except that the initial chaining values and the final message digest lengths are different. The program may obtain the SHA-224 message digest using the SHA-256 functions with the following two actions:

   a. The following initial hexadecimal chaining values for SHA-224 are used:

      ```
      H0 = C105 9ED8
      H1 = 367C D507
      H2 = 3070 DD17
      H3 = F70E 5939
      H4 = FFC0 0B31
      H5 = 6858 1511
      H6 = 64F9 8FA7
      H7 = BEFA 4FA4
      ```

   b. The 224-bit message digest is obtained by truncating the final message digest to its leftmost 224 bits.

9. The SHA-384 algorithm is the same as the SHA-512 algorithm, except that the initial chaining values and the final message digest lengths are different. The program may obtain the SHA-384 message digest using the SHA-512 functions with the following two actions:

   a. The following initial hexadecimal chaining values for SHA-384 are used:

      ```
      H0 = CBBB 9D5D C105 9ED8
      H1 = 629A 292A 367C D507
      H2 = 9159 015A 3070 DD17
      H3 = 152F ECD8 F70E 5939
      H4 = 6733 2667 FFC0 0B31
      H5 = 8EB4 4A87 6858 1511
      H6 = DB0C 2E0D 64F9 8FA7
      H7 = 47B5 481D BEFA 4FA4
      ```

   b. The 384-bit message digest is obtained by truncating the final message digest to its leftmost 384 bits.

10. For the SHA3 and SHAKE functions (for both KIMD and KLMD), the KECCAK[*c*] algorithm uses a 1,600-bit state array in the parameter block. The state array comprises 25 *lanes* having 64 bits each. The bits of the state array are numbered from 0 to from 1,599, and the bits of each lane are numbered from 0 to 63.

    In the parameter block, the bytes of the state array appear in ascending order from left-to-right; but within each byte, the bits appear in descending order from left to right. That is, byte 0 of the parameter block contains (from left to right) bits 7, 6, 5, 4, 3, 2, 1, and 0 of the state array; byte 1

contains bits 15, 14, 13, 12, 11, 10, 9 and 8 of the state array; and so forth. When represented in memory, the bit positions of any individual lane of the state array correspond to powers of two, as shown below.

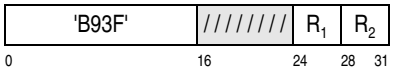
Byte	0								1									6								7							
Lane Bit	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	...	55	54	53	52	51	50	49	48	63	62	61	60	59	58	57	56

The bit ordering of the parameter block described above also applies to the message bytes of the second operand. For the KLMD-SHAKE functions, the results stored in the first operand are similar to that of the parameter block in that the left-to-right ordering of bits within each byte is 7, 6, 5, 4, 3, 2, 1, and 0.

11. For the GHASH function, the hash subkey, H, is the result of encrypting a 128-bit zero using the AES encryption algorithm with a 128-bit, 192-bit, or 256-bit cryptographic key.

COMPUTE LAST MESSAGE DIGEST

KLMD R_1,R_2 [RRE]



A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction are reserved, and for all functions except KLMD-SHAKE-128 and KLMD-SHAKE-256, the R_1 field is reserved. Reserved fields should contain zeros; otherwise, the program may not operate compatibly in the future.

Bit positions 57-63 of general register 0 contain the function code. Figure 7-233 show the assigned function codes. All other function codes are unassigned. Bit 56 of general register 0 must be zero; otherwise, a specification exception is recognized. For the KLMD-SHAKE functions, bit 55 of general register 0 is the padding state; bit 55 of the register is ignored for all other functions. All other bits of general register 0 are ignored.

The assigned function codes are enumerated in Figure 7-233. All other function codes are unassigned.

Code	Function	Parm. Block Size (bytes)	Data Block Size[1] (bytes)
0	KLMD-Query *	16	—
1	KLMD-SHA-1 *	28	64
2	KLMD-SHA-256 *	40	64
3	KLMD-SHA-512 *	80	128
32	KLMD-SHA3-224	200	144
33	KLMD-SHA3-256	200	136
34	KLMD-SHA3-384	200	104
35	KLMD-SHA3-512	200	72
36	KLMD-SHAKE-128	200	168
37	KLMD-SHAKE-256	200	136

Explanation:

— Not applicable

1 For all data blocks except the last block. The size of the last data block ranges from 0 to one less than the value shown.

* Function is also defined in the ESA/390 architectural mode and the ESA/390-compatibility mode. It is unpredictable whether other function codes are available in the ESA/390-compatibility mode.

Figure 7-233. Function Codes for COMPUTE LAST MESSAGE DIGEST

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register 1 constitute the address.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R_1, $R_1 + 1$, R_2 and $R_2 + 1$ are ignored for the query function.

For all other functions, the second operand is processed as specified by the function code using an ini-

tial chaining value in the parameter block, and the result replaces the chaining value. For the SHA-1, SHA-256, and SHA-512 functions, the operation also uses a message bit length in the parameter block. The operation proceeds until the end of the second-operand location is reached or a CPU-determined number of bytes have been processed, whichever occurs first.

For the KLMD-SHAKE functions, when the end of the second operand is reached, an extended-output-function (XOF) digest is stored at the first operand location. The operation then proceeds until either the end of the first-operand location is reached or a CPU-determined number of bytes have been stored, whichever occurs first.

The result is indicated in the condition code.

The R_2 field designates an even-odd pair of general registers and must designate an even-numbered register other than general register 0; otherwise, a specification exception is recognized.

The location of the leftmost byte of the second operand is specified by the contents of the R_2 general register. The number of bytes in the second-operand location is specified in general register $R_2 + 1$.

As part of the operation, the address in general register R_2 is incremented by the number of bytes processed from the second operand, and the length in general register $R_2 + 1$ is decremented by the same number. The formation and updating of the address and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general register R_2 constitute the address of the second operand, and the contents of bit positions 0-39 are ignored; bits 40-63 of the updated address replace the corresponding bits in general register R_2, carries out of bit position 40 of the updated address are ignored, and the contents of bit positions 32-39 of general register R_2 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register R_2 constitute the address of the second operand, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated address replace the corresponding bits in general register R_2, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general register R_2 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register R_2 constitute the address of the second operand; bits 0-63 of the updated address replace the contents of general register R_2 and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_2 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of bit positions 32-63 of general register $R_2 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of general register $R_2 + 1$.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_2 and $R_2 + 1$, always remain unchanged.

For the two KLMD-SHAKE functions, the following applies:

- Bit 55 of general register 0 is the padding state (PS). A value of zero indicates that padding of the second operand has not yet been performed. A value of one indicates that padding of the second operand has been performed.

  A specification exception is recognized, and the operation is suppressed when the padding state is one and the second-operand length in general register $R_2 + 1$ is nonzero at the beginning of the instruction.

  When the remaining second-operand length is zero, the CPU inspects the padding state to determine whether padding of the second operand is to be performed. The padding state is set to one by the CPU when padding of the second operand has been performed.

- The R_1 field designates an even-odd pair of general registers and must designate an even-numbered register other than general register 0 and other than general register R_2; otherwise a specification exception is recognized.

- The location of the leftmost byte of the first operand is specified by the contents of general register R_1. The number of bytes in the first-operand location is specified in general register $R_1 + 1$.

- As a part of the operation, the address in general register R_1 is incremented by the number of

bytes stored into the first operand, and the length in general register $R_1 + 1$ is decremented by the same number. The formation and updating of the address and length is dependent on the addressing mode.

- The addressing-mode characteristics for general registers R_1 and $R_1 + 1$ are identical to those for general registers R_2 and $R_2 + 1$, respectively, as described above.

For functions other than the two KLMD-SHAKE functions, bit 55 of general register 0 and the R_1 field of the instruction are ignored. In this case, a first operand is not present, and general registers R_1 and $R_1 + 1$ are not modified.

Figure 7-234 shows the contents of the general registers just described.

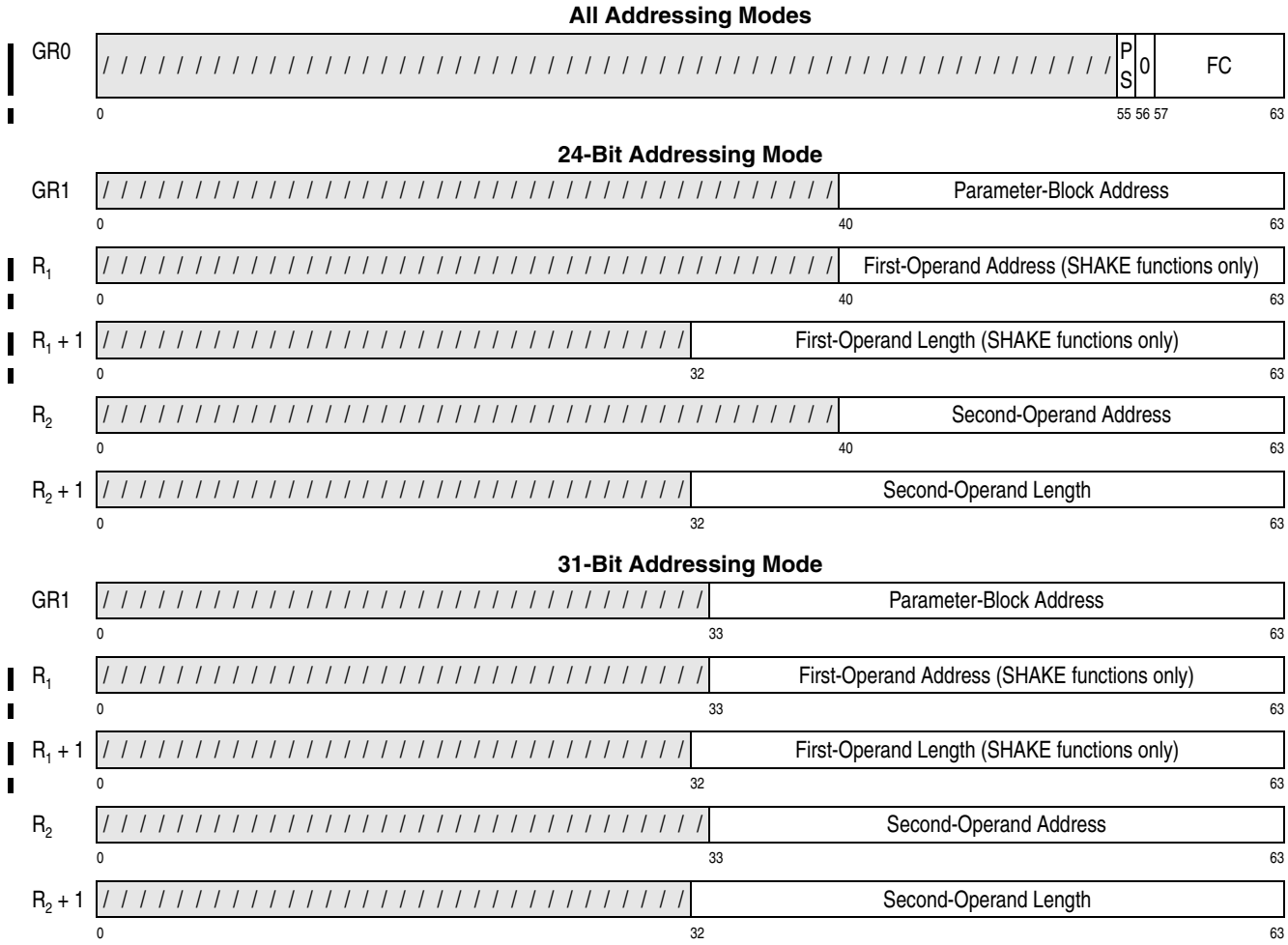


Figure 7-234. General Register Assignment for KLMD (Part 1 of 2)

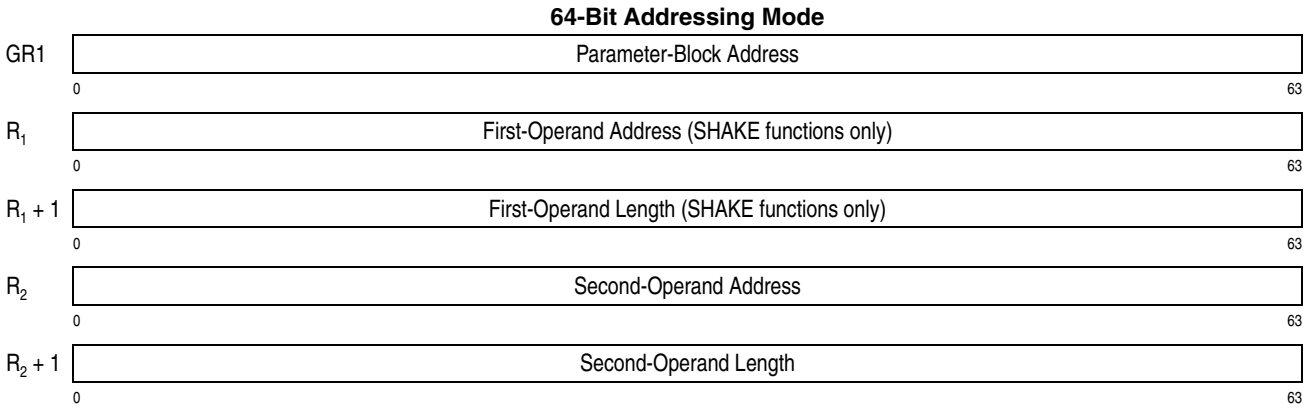


Figure 7-234. General Register Assignment for KLMD (Part 2 of 2)

In the access-register mode, access registers 1 and R_2 specify the address spaces containing the parameter block and second operand, respectively. For the KLMD-SHAKE functions, access register R_1 specifies the address space containing the first operand.

The result is obtained as if processing starts at the left end of the second operand and proceeds to the right, block by block. For all functions except the KLMD-SHAKE functions, the operation is ended when all source bytes in the second operand have been processed (called normal completion), or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

When the chaining-value field overlaps any portion of the first or second operand, the result in the chaining-value field is unpredictable.

For the KLMD-SHA-1, KLMD-SHA-256, and KLMD-SHA-512 functions, after all bytes in the second operand as specified in general register $R_2 + 1$ have been processed, the padding operation is performed, a final hashing operation is performed on the padded block, and then normal completion occurs. For the KLMD-SHA3 and KLMD-SHAKE functions, after all full blocks of the second operand have been processed, the padding operation is performed on either the remaining partial block or on a null block, a final hashing operation is performed on the padded block; normal completion then occurs for the KLMD-SHA3 functions.

For the KLMD-SHAKE functions, when padding has been performed, the padding state is set to one in general register 0, and an extended-output-function (XOF) message digest is stored into the first-operand location. XOF message-digest generation is ended when all of the first operand has been stored (called normal completion) or when a CPU-determined number of blocks that is less than the length of the first operand have been stored (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

When the operation ends due to normal completion, condition code 0 is set, the second-operand address in general register R_2 is updated, and the second-operand length in general register $R_2 + 1$ is zero; for the KLMD-SHAKE functions, the first-operand address in general register R_1 is updated, and the first-operand length in general register $R_1 + 1$ is zero.

When the operation ends due to partial completion, condition code 3 is set. For all functions except the KLMD-SHAKE functions, the resulting value in general register $R_2 + 1$ is nonzero. For the KLMD-SHAKE functions, if the second operand has not been completely processed, the resulting value in general register $R_2 + 1$ is nonzero, and if the first

operand has not been completely processed, the resulting value in general register R_1 is nonzero.

When the second-operand length is initially zero, the second operand is not accessed, the second-operand address and second-operand length in general registers R_2 and $R_2 + 1$, respectively, are not changed, and condition code 0 is set. Except for the KLMD-SHAKE functions when the padding-state (PS) control is one, the empty block padding operation is performed and the result is stored into the parameter block.

For the KLMD-SHAKE functions, when the first-operand length is initially zero, the first operand is not accessed, and the first-operand address and first-operand length in general registers R_1 and $R_1 + 1$, respectively, are not changed. When the first-operand length is initially zero and the PS control is one, it is model dependent whether the parameter block is updated.

A PER storage-alteration event is recognized, when applicable, for the portion of the parameter block and first-operand location that is stored. A PER zero-address-detection event is recognized, when applicable, for the first-operand location, for the second-operand location and for the parameter block. When PER events are detected for more than one location, it is unpredictable which location is identified in the PER access identification (PAID) and PER ASCE ID (AI).

As observed by this CPU, other CPUs, and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

Access exceptions may be reported for a larger portion of the first operand (when applicable) and second operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of the first or second operand nor for locations more than 4K bytes beyond the current location being processed.

Symbols Used in Function Descriptions

The subsequent description of COMPUTE LAST MESSAGE DIGEST uses the same symbols as those used in COMPUTE INTERMEDIATE MESSAGE DIGEST, plus the following additional symbols. Further description of the secure hash algorithm may be found in Reference [15.] on page xxx. Further description of the SHA-3 and SHAKE hash algorithms and the KECCAK family of sponge functions may be found in Reference [21.] on page xxx.

Symbol	Explanation for KLMD Function Figures
L	Byte length of operand 2 in storage.
p <n>	n padding bytes; leftmost byte is 80 hex; all other bytes are 00 hex.
z <n>	n padding bytes of zero.
mbl	an 8-byte or 16-byte value specifying the bit length of the total message.
q <64>	a padding block, consisting of 56 bytes of zero followed by an 8-byte mbl.
q <128>	a padding block, consisting of 112 bytes of zero followed by a 16-byte mbl.
sp<n>	n padding bytes used by SHA-3 functions. Bit positions 7, 6, and 5 of the leftmost byte of the pad contain binary 0, 1, and 1, respectively, and bit position 0 of the rightmost byte of the pad contains a binary one. All other bits are zeros. Padding for SHA-3 functions is performed even if the second-operand length is zero.
xp<n>	n padding bytes used by SHAKE functions. Bit positions 3 through 7 of the leftmost byte of the pad all contain binary ones, and bit position 0 of the rightmost pad byte contains a binary one. All other bits are zeros. Padding for SHAKE functions is performed when the padding state (PS, bit 55 of general register 0) is zero and the remaining second-operand length is less than the data-block size.

KLMD-Query (KLMD Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

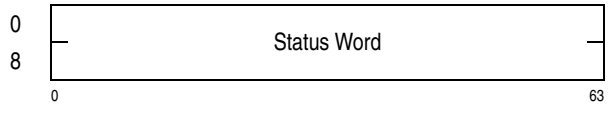


Figure 7-235. Parameter Block for KLMD-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the COMPUTE LAST MESSAGE DIGEST instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KLMD-Query function completes; condition code 3 is not applicable to this function.

KLMD-SHA-1 (KLMD Function Code 1)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

0	H0
4	H1
8	H2
12	H3
16	H4
20 24	Message Bit Length (mbl)
	0 31

Figure 7-236. Parameter Block for KLMD-SHA-1

The message digest for the message (M) in operand 2 is generated using the SHA-1 algorithm with the chaining value (called H fields) and message-bit-length information in the parameter block.

If the length of the message in operand 2 is equal to or greater than 64 bytes, an intermediate message digest is generated for each 64-byte message block using the SHA-1 block digest algorithm with the 20-byte chaining value in the parameter block, and the generated intermediate message digest, also called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. This operation is shown in Figure 7-237 and repeats until the remaining message is less than 64 bytes or until a CPU-determined number of blocks have been stored.

If the length of the message or the remaining message is zero bytes, then the operation in Figure 7-238 is performed. If the length of the message or the remaining message is between one byte and 55 bytes inclusive, then the operation in Figure 7-239 is performed; if the length is between 56 bytes and 63 bytes inclusive, then the operation in Figure 7-240 is performed. The message digest, also called the output chaining value (OCV), is stored into the chaining-value field of the parameter block.

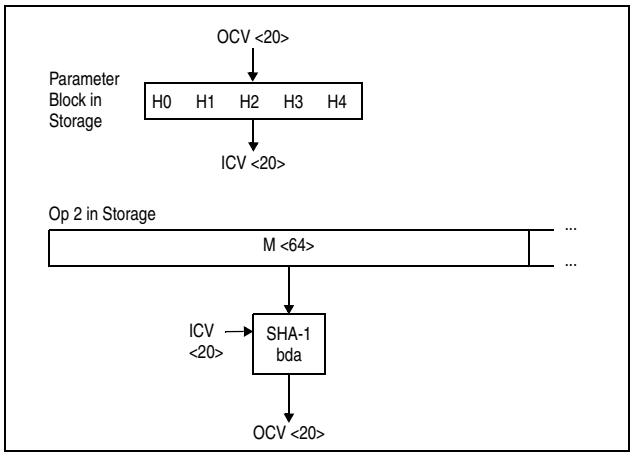


Figure 7-237. KLMD-SHA-1 Full Block (L ≥ 64)

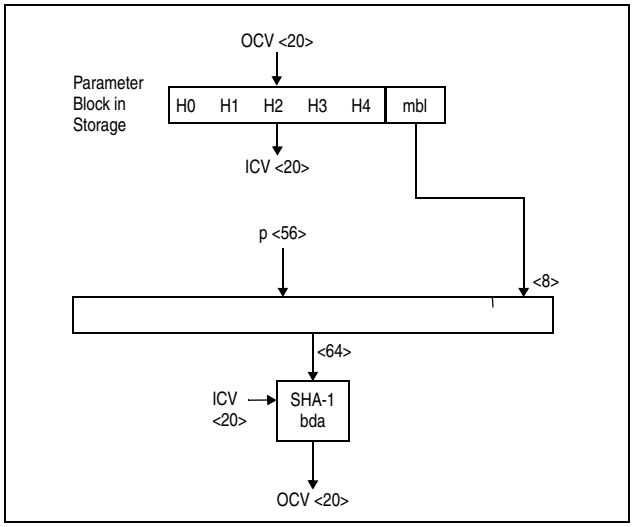


Figure 7-238. KLMD-SHA-1 Empty Block (L = 0)

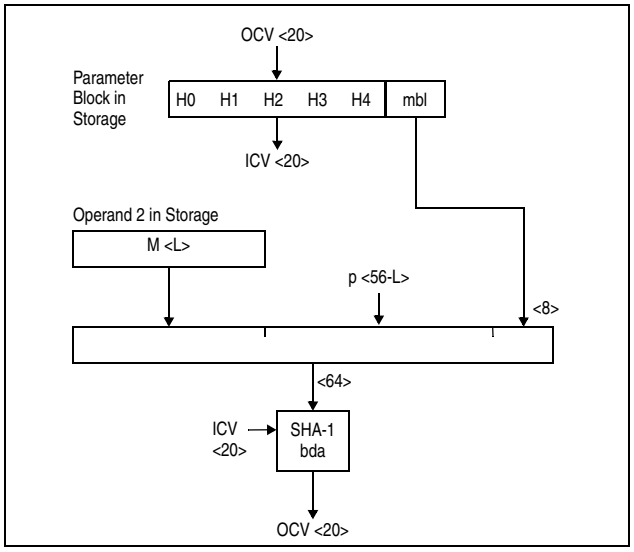


Figure 7-239. KLMD-SHA-1 Partial-Block Case 1 (1 ≤ L ≤ 55)

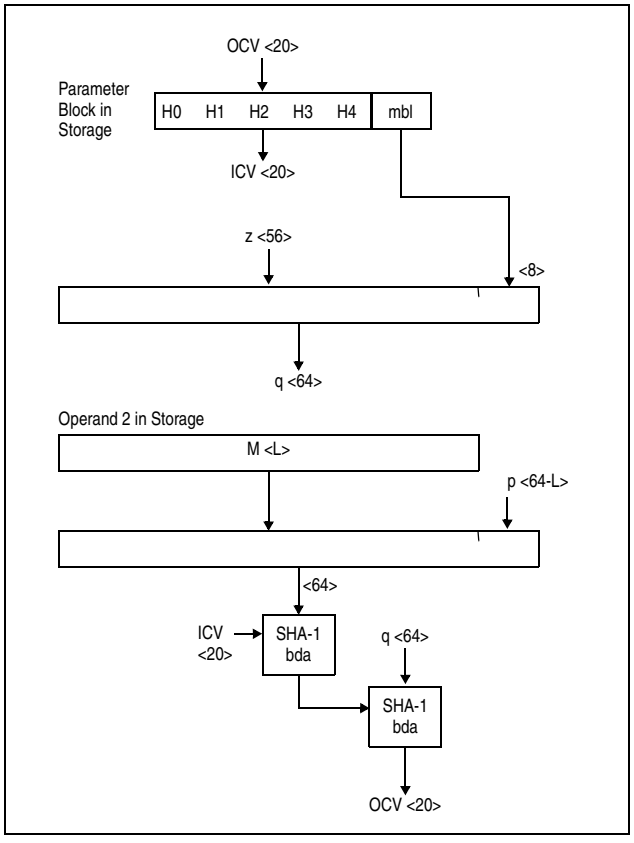


Figure 7-240. KLMD-SHA-1 Partial-Block Case 2 (56 ≤ L ≤ 63)

KLMD-SHA-256 (KLMD Function Code 2)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

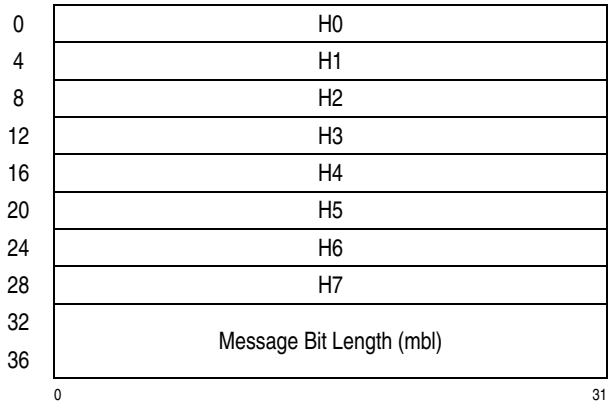


Figure 7-241. Parameter Block for KLMD-SHA-256

The message digest for the message (M) in operand 2 is generated using the SHA-256 algorithm with the chaining value (called H fields) and message-bit-length information in the parameter block.

If the message in operand 2 is equal to or greater than 64 bytes, an intermediate message digest is generated for each 64-byte message block using the SHA-256 block digest algorithm with the 32-byte chaining value in the parameter block, and the generated intermediate message digest, also called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. This operation is shown in Figure 7-242 and repeats until the remaining message is less than 64 bytes or until a CPU-determined number of blocks have been stored.

If the length of the message or the remaining message is zero bytes, then the operation in Figure 7-243 is performed. If the length of the message or the remaining message is between one byte and 55 bytes inclusive, then the operation in Figure 7-244 is performed; if the length is between 56 bytes and 63 bytes inclusive, then the operation in Figure 7-245 is performed. The message digest, also called the out-

put chaining value (OCV), is stored into the chaining-value field of the parameter block.

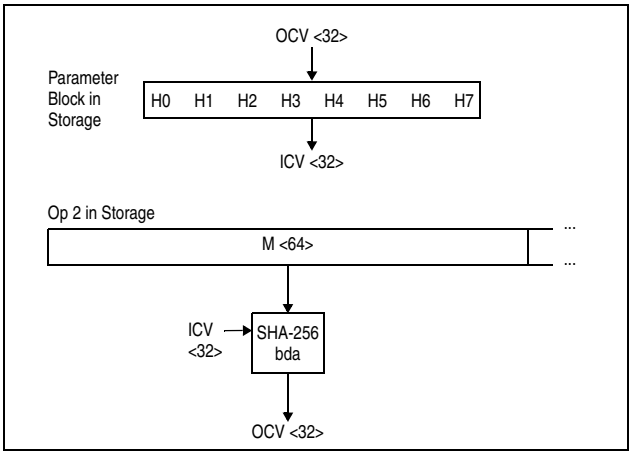


Figure 7-242. KLMD-SHA-256 Full Block (L ≥ 64)

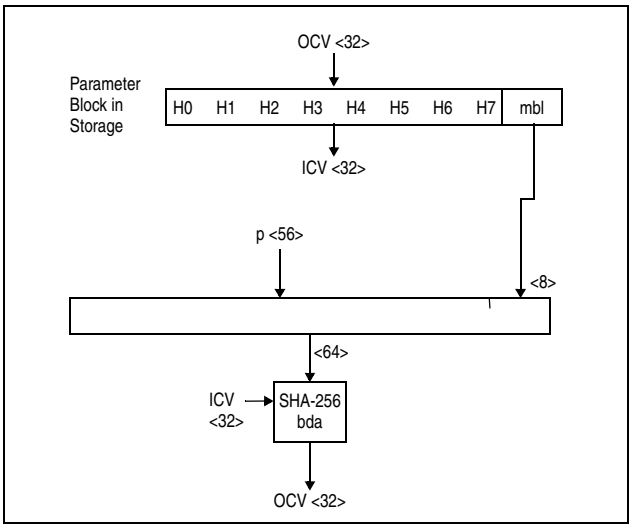


Figure 7-243. KLMD-SHA-256 Empty Block (L = 0)

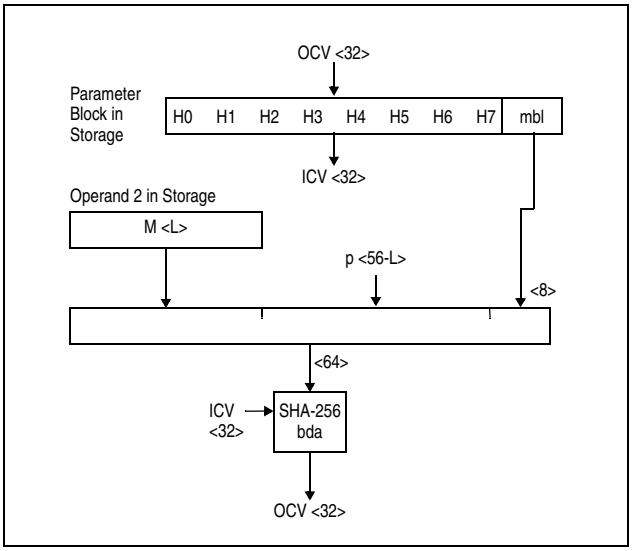


Figure 7-244. KLMD-SHA-256 Partial-Block Case 1 (1 ≤ L ≤ 55)

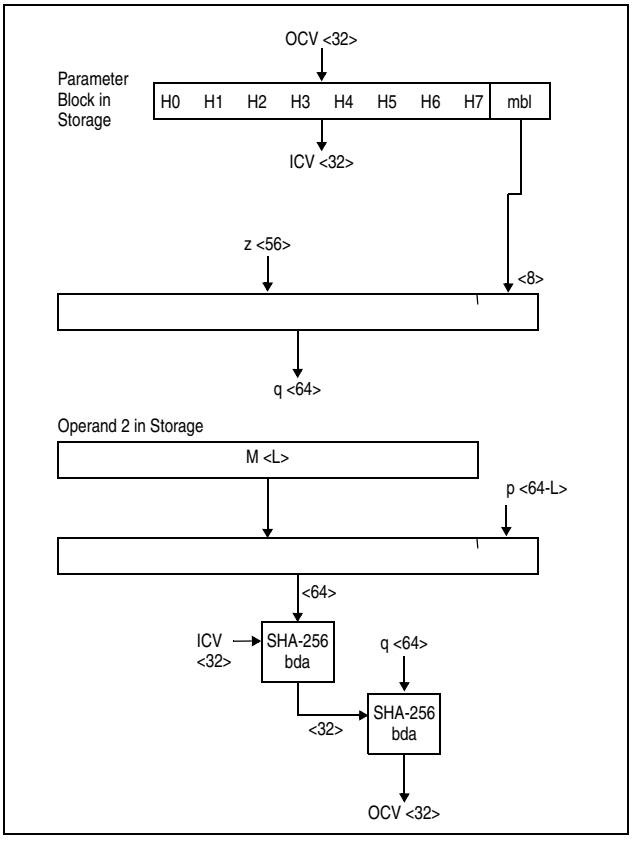


Figure 7-245. KLMD-SHA-256 Partial-Block Case 2 (56 ≤ L ≤ 63)

KLMD-SHA-512 (KLMD Function Code 3)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

0	H0
8	H1
16	H2
24	H3
32	H4
40	H5
48	H6
56	H7
64 72	Message Bit Length (mbl)

0 64

Figure 7-246. Parameter Block for KLMD-SHA-512

The message digest for the message (M) in operand 2 is generated using the SHA-512 algorithm with the chaining value (called H fields) and message-bit-length information in the parameter block.

If the message in operand 2 is equal to or greater than 128 bytes, an intermediate message digest is generated for each 128-byte message block using the SHA-512 block digest algorithm with the 64-byte chaining value in the parameter block, and the generated intermediate message digest, also called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. This operation is shown in Figure 7-247 and repeats until the remaining message is less than 128 bytes or until a CPU-determined number of blocks have been stored.

If the length of the message or the remaining message is zero bytes, then the operation in Figure 7-248 is performed. If the length of the message or the remaining message is between one byte and 111 bytes inclusive, then the operation in Figure 7-249 is performed; if the length is between 112 bytes and 127 bytes inclusive, then the operation in Figure 7-250 is performed. The message digest, also called the output chaining value (OCV), is stored into the chaining-value field of the parameter block.

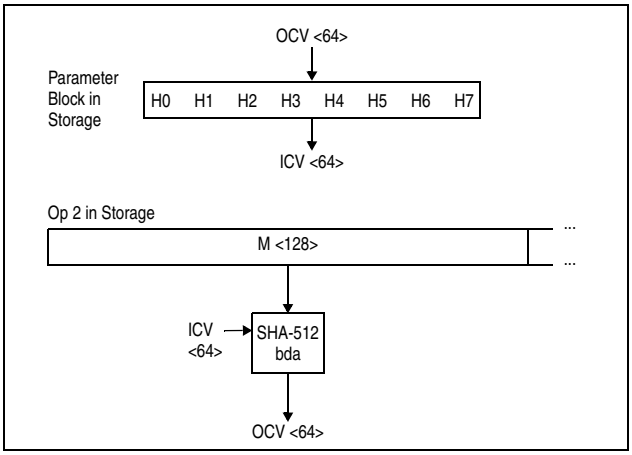


Figure 7-247. KLMD-SHA-512 Full Block (L ≥ 128)

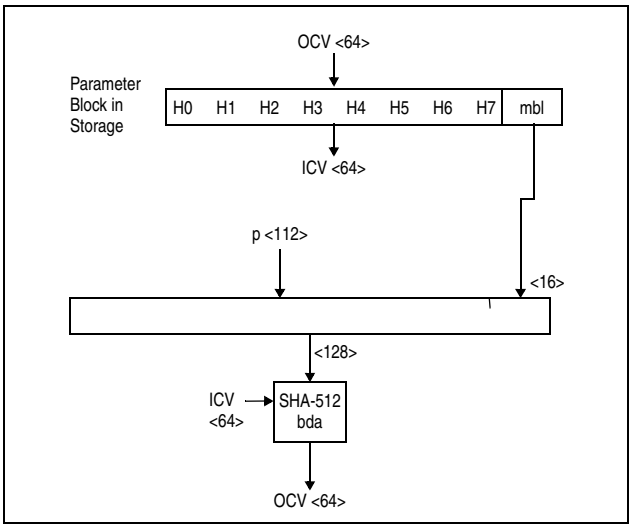


Figure 7-248. KLMD-SHA-512 Empty Block (L = 0)

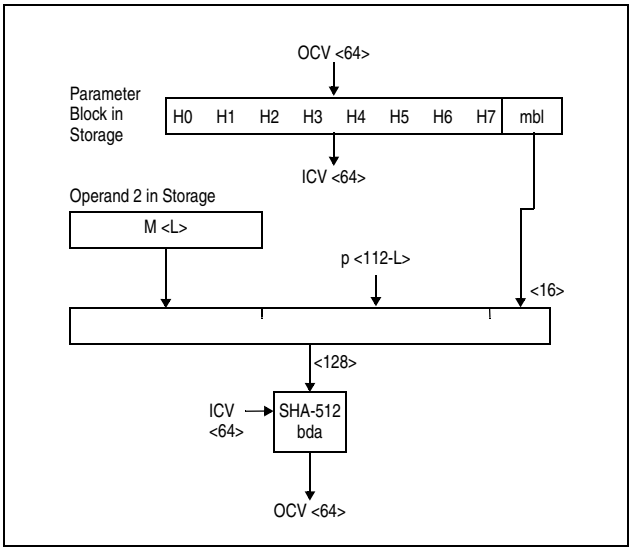


Figure 7-249. KLMD-SHA-512 Partial-Block Case 1 (1 ≤ L ≤ 111)

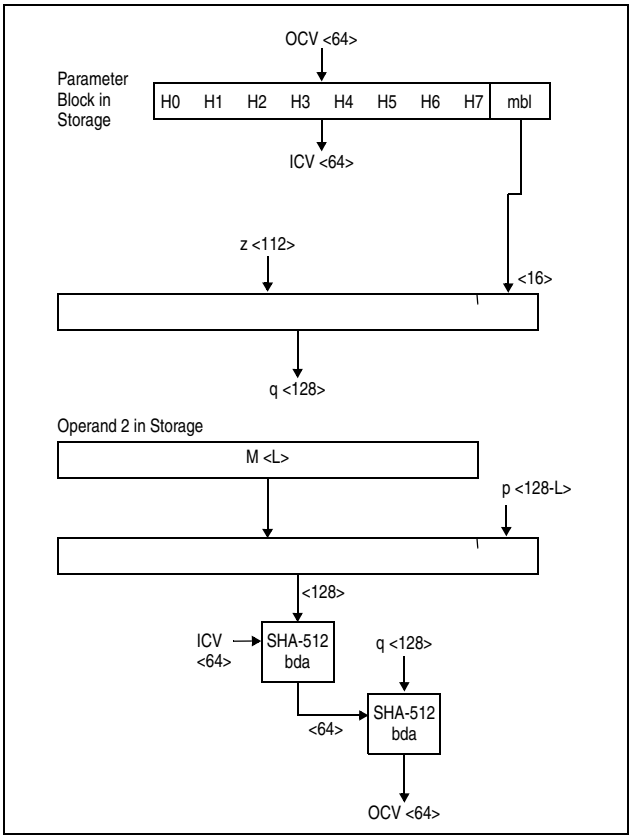


Figure 7-250. KLMD-SHA-512 Partial-Block Case 2 (112 ≤ L ≤ 127)

KLMD-SHA3-224 (KLMD Function Code 32)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

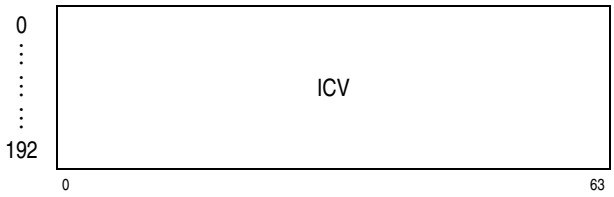


Figure 7-251. Parameter Block for KLMD-SHA3-224

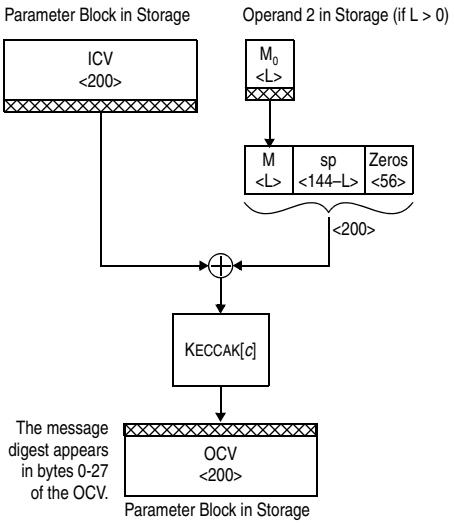
The initial chaining value (ICV) represents the 1600-bit state array used by the KECCAK[*c*] functions which implement the SHA-3 algorithms described in Reference [21.] on page xxx.

When the length of the second operand in general register $R_2 + 1$ is greater than or equal to 144 bytes, then the operation is identical to that described in the section “KIMD-SHA3-224 (KIMD Function Code 32)” on page 7-193. This operation proceeds until the length of the second operand is less than 144 bytes, at which point the operation continues as described below.

Any remaining bytes of the second operand are padded on the right to form a 144-byte message block as described for the symbol “sp<n>” on page 7-205. Padding occurs even when there are no remaining bytes in the second operand and does not alter the contents of the second operand.

A 224-bit (28-byte) message digest is generated for the padded message block using the KECCAK[*c*] algorithm with the 200-byte initial chaining value in the parameter block. The message digest is generated regardless of whether the second-operand length is zero. The entire 200-byte output of the KECCAK[*c*] algorithm, also called the output chaining value (OCV), is stored into the parameter block. The generated message digest is contained in bytes 0-27 of the parameter block. See programming note 10 on page 7-200 regarding the numbering of the bits in the parameter block and second operand as compared with the numbering of the bits in the state array.

The operation is shown in Figure 7-252.



Explanation:

⊠	The bit positions of each byte of the parameter block and the message block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
⊕	Bitwise exclusive-OR operation
<#>	Size of an item in bytes
ICV	Input chaining value (in the parameter block)
L	Length of the second operand (< 144 bytes)
M	Message block (when L > 0)
OCV	Output chaining value (in the parameter block); bytes 0-27 contain the 224-bit message digest
sp	SHA-3 padding algorithm

Figure 7-252. KLMD-SHA3-224 Processing (L < 144)

KLMD-SHA3-256 (KLMD Function Code 33)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

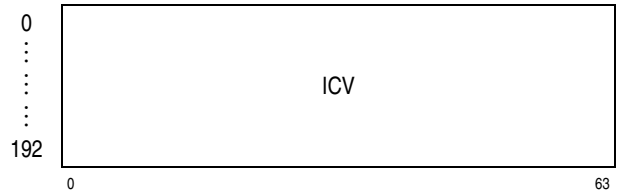


Figure 7-253. Parameter Block for KLMD-SHA3-256

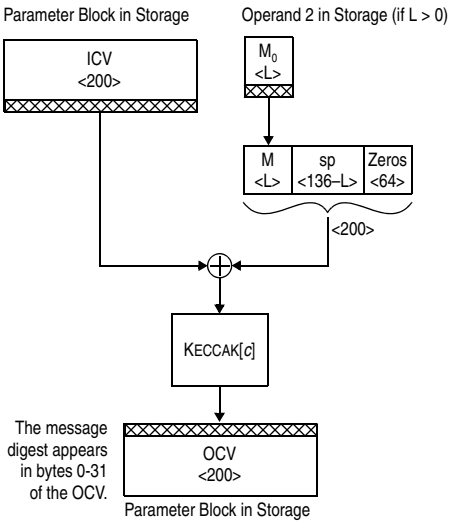
The initial chaining value (ICV) represents the 1600-bit state array used by the KECCAK[*c*] functions which implement the SHA-3 algorithms described in Reference [21.] on page xxx.

When the length of the second operand in general register $R_2 + 1$ is greater than or equal to 136 bytes, then the operation is identical to that described in the section "KIMD-SHA3-256 (KIMD Function Code 33)" on page 7-194. This operation proceeds until the length of the second operand is less than 136 bytes, at which point the operation continues as described below.

Any remaining bytes of the second operand are padded on the right to form a 136-byte message block as described for the symbol "sp<n>" on page 7-205. Padding occurs even when there are no remaining bytes in the second operand and does not alter the contents of the second operand.

A 256-bit (32-byte) message digest is generated for the padded message block using the KECCAK[*c*] algorithm with the 200-byte initial chaining value in the parameter block. The message digest is generated regardless of whether the second-operand length is zero. The entire 200-byte output of the KECCAK[*c*] algorithm, also called the output chaining value (OCV), is stored into the parameter block. The generated message digest is contained in bytes 0-31 of the parameter block. See programming note 10 on page page 7-200 regarding the numbering of the bits in the parameter block and second operand as compared with the numbering of the bits in the state array.

The operation is shown in Figure 7-254.



Explanation:

⊠	The bit positions of each byte of the parameter block and the message block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
⊕	Bitwise exclusive-OR operation
<#>	Size of an item in bytes
ICV	Input chaining value (in the parameter block)
L	Length of the second operand (< 136 bytes)
M	Message block (when L > 0)
OCV	Output chaining value (in the parameter block); bytes 0-31 contain the 256-bit message digest
sp	SHA-3 padding algorithm

Figure 7-254. KLMD-SHA3-256 Processing (L < 136)

KLMD-SHA3-384 (KLMD Function Code 34)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

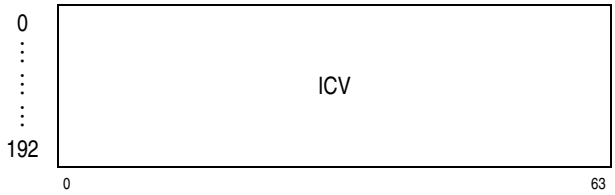


Figure 7-255. Parameter Block for KLMD-SHA3-384

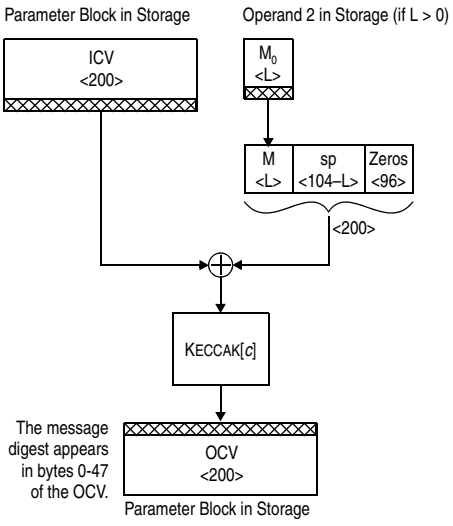
The initial chaining value (ICV) represents the 1600-bit state array used by the KECCAK[*c*] functions which implement the SHA-3 algorithms described in Reference [21.] on page xxx.

When the length of the second operand in general register $R_2 + 1$ is greater than or equal to 104 bytes, then the operation is identical to that described in the section “KIMD-SHA3-384 (KIMD Function Code 34)” on page 7-195. This operation proceeds until the length of the second operand is less than 104 bytes, at which point the operation continues as described below.

Any remaining bytes of the second operand are padded on the right to form a 104-byte message block as described for the symbol “sp<n>” on page 7-205. Padding occurs even when there are no remaining bytes in the second operand and does not alter the contents of the second operand.

A 384-bit (48-byte) message digest is generated for the padded message block using the KECCAK[*c*] algorithm with the 200-byte initial chaining value in the parameter block. The message digest is generated regardless of whether the second-operand length is zero. The entire 200-byte output of the KECCAK[*c*] algorithm, also called the output chaining value (OCV), is stored into the parameter block. The generated message digest is contained in bytes 0-47 of the parameter block. See programming note 10 on page 7-200 regarding the numbering of the bits in the parameter block and second operand as compared with the numbering of the bits in the state array.

The operation is shown in Figure 7-256.



Explanation:

⊠	The bit positions of each byte of the parameter block and the message block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
⊕	Bitwise exclusive-OR operation
<#>	Size of an item in bytes
ICV	Input chaining value (in the parameter block)
L	Length of the second operand (< 104 bytes)
M	Message block (when L > 0)
OCV	Output chaining value (in the parameter block); bytes 0-47 contain the 384-bit message digest
sp	SHA-3 padding algorithm

Figure 7-256. KLMD-SHA3-384 Processing (L < 104)

KLMD-SHA3-512 (KLMD Function Code 35)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

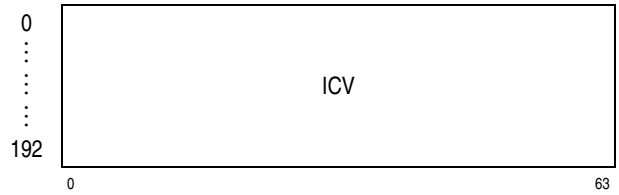


Figure 7-257. Parameter Block for KLMD-SHA3-512

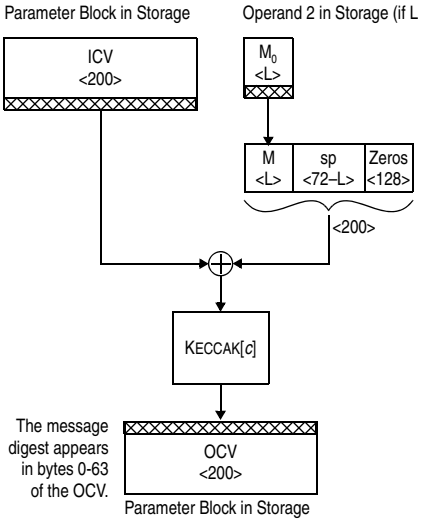
The initial chaining value (ICV) represents the 1600-bit state array used by the KECCAK[*c*] functions which implement the SHA-3 algorithms described in Reference [21.] on page xxx.

When the length of the second operand in general register $R_2 + 1$ is greater than or equal to 72 bytes, then the operation is identical to that described in the section “KIMD-SHA3-512 (KIMD Function Code 35)” on page 7-196. This operation proceeds until the length of the second operand is less than 72 bytes, at which point the operation continues as described below.

Any remaining bytes of the second operand are padded on the right to form a 72-byte message block as described for the symbol “sp<n>” on page 7-205. Padding occurs even when there are no remaining bytes in the second operand and does not alter the contents of the second operand.

A 512-bit (64-byte) message digest is generated for the padded message block using the KECCAK[*c*] algorithm with the 200-byte initial chaining value in the parameter block. The message digest is generated regardless of whether the second-operand length is zero. The entire 200-byte output of the KECCAK[*c*] algorithm, also called the output chaining value (OCV), is stored into the parameter block. The generated message digest is contained in bytes 0-63 of the parameter block. See programming note 10 on page 7-200 regarding the numbering of the bits in the parameter block and second operand as compared with the numbering of the bits in the state array.

The operation is shown in Figure 7-258.



Explanation:

⊠	The bit positions of each byte of the parameter block and the message block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
⊕	Bitwise exclusive-OR operation
<#>	Size of an item in bytes
ICV	Input chaining value (in the parameter block)
L	Length of the second operand (< 72 bytes)
M	Message block (when L > 0)
OCV	Output chaining value (in the parameter block); bytes 0-63 contain the 512-bit message digest
sp	SHA-3 padding algorithm

Figure 7-258. KLMD-SHA3-512 Processing (L < 72)

KLMD-SHAKE-128 (KLMD Function Code 36)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

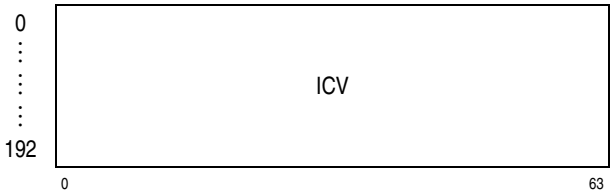


Figure 7-259. Parameter Block for KLMD-SHAKE-128

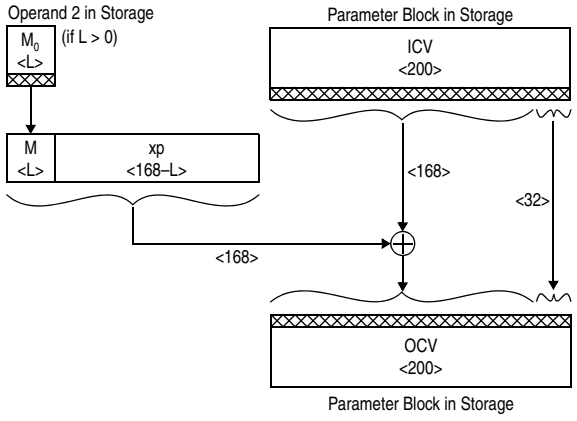
The initial chaining value (ICV) represents the 1600-bit state array used by the KECCAK[*c*] functions which implement the SHA-3 algorithms described in Reference [21.] on page xxx.

When the length of the second operand in general register $R_2 + 1$ is greater than or equal to 168 bytes, the operation is identical to that described in the section "KIMD-SHAKE-128 (KIMD Function Code 36)" on page 7-197, except that when the remaining second-operand length is zero, processing is as described below.

When the padding state, bit 55 of general register 0 is zero (indicating that padding has not yet been performed), the following occurs:

- Any remaining bytes of the second operand are padded on the right to form a 168-byte message block as described for the symbol "xp<n>" on page 7-205. Padding occurs even when there are no remaining bytes in the second operand and does not alter the contents of the second operand.
- The padding state is set to one, indicating that padding has been performed.
- The second-operand address in general register R_2 is incremented by the number of message bytes processed, and the second-operand length in general register R_2+1 is set to zero.
- The 168-byte padded message, is exclusive ORed with the contents of the leftmost 168 bytes of the state array (from the ICV in the parameter block or from the OCV resulting from the previous block's processing) to form an output chaining value that is used in the extended-output-function (XOF) processing. The rightmost 32 bytes of the state array are unchanged.

The padding operation is shown in Figure 7-260.



Explanation:

⊠	The bit positions of each byte of the parameter block and the message block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
⊕	Bitwise exclusive-OR operation
<#>	Size of an item in bytes
ICV	Input chaining value (in the parameter block)
L	Length of the second operand (< 168 bytes)
M	Message block (when L > 0)
OCV	Output chaining value (used in XOF processing, below)
xp	SHAKE padding algorithm

Figure 7-260. KLMD-SHAKE-128 Padding (L < 168)

Depending on the number of second-operand blocks processed when padding is completed, either (a) the output chaining value is stored into the parameter block, and the instruction completes by setting condition code 3 (partial completion), or (b) the operation continues with extended-output-function (XOF) processing, as described below.

When the padding state is one (indicating that padding has been performed for the message, either by the current or a previous execution of the instruction), extended-output-function (XOF) processing is performed as follows.

1. If the first-operand length in general register $R_1 + 1$ is zero, then the output chaining value is stored into the parameter block, and the instruction completes with condition code 0. If the first-operand length is zero at the beginning of the instruction, then it is model dependent whether the ICV is fetched from the parameter block and stored back unmodified as the OCV.
2. The KECCAK[*c*] function is invoked using the previous output-chaining value (OCV) as input, and replacing the output-chaining value.
3. General register R_1 contains the current address of the first operand, and general register $R_1 + 1$ contains the remaining length of the first operand. The number of bytes to be stored, *n*, is either the remaining first-operand length or 168, whichever is smaller.

   The first *n* bytes of the output-chaining value are stored at the first-operand location. See programming note 10 on page 7-200 regarding the numbering of the bits in the first operand as compared with the numbering of the bits in the state array.

   The first-operand address in general register R_1 is incremented by *n*, and the first-operand length general register $R_1 + 1$ is decremented by *n*.

Steps 1-3 of this process are repeated until the first-operand length becomes zero (in which case, the instruction completes with condition code 0) or until a CPU-determined number of bytes have been stored (in which case, the instruction completes with condition code 3). The output-chaining value is stored into bytes 0-199 of the parameter block regardless of whether condition code 0 or 3 is set. Figure 7-261 illustrates the XOF processing described above.

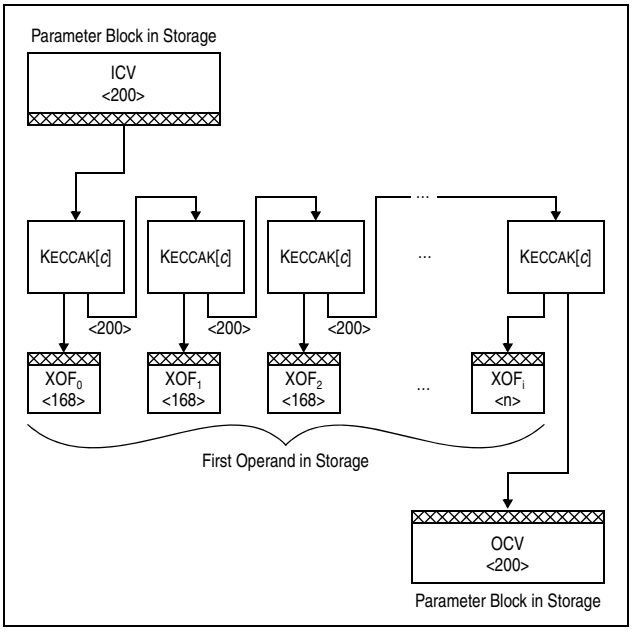


Figure 7-261. KLMD-SHAKE-128 Extended-Output-Function (XOF) Processing

Explanation:

⊠	The bit positions of each byte of the parameter block and the XOF block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
<#>	Size of an item in bytes
ICV	Input chaining value (either from the parameter block or from the output of the padding operation).
OCV	Output chaining value; (the final OCV appears in the parameter block)
n	The residual length of the final block of the first operand (less than or equal to 168 bytes)
XOF_i<n>	Block i of the extended-output-function results in the first operand.

Figure 7-261. KLMD-SHAKE-128 Extended-Output-Function (XOF) Processing (Continued)

KLMD-SHAKE-256 (KLMD Function Code 37)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-208 on page 7-189.

The parameter block used for the function has the following format:

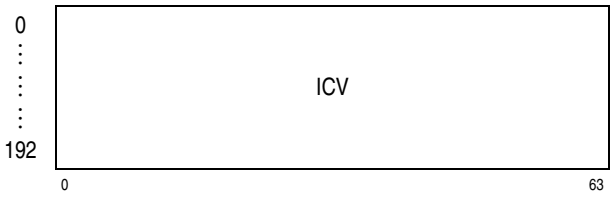


Figure 7-262. Parameter Block for KLMD-SHAKE-256

The initial chaining value (ICV) represents the 1600-bit state array used by the KECCAK[*c*] functions which implement the SHA-3 algorithms described in Reference [21.] on page xxx.

When the length of the second operand in general register R_2 + 1 is greater than or equal to 136 bytes, the operation is identical to that described in the section "KIMD-SHAKE-256 (KIMD Function Code 37)" on page 7-198, except that when the remaining second-operand length is zero, processing is as described below. This operation proceeds until the length of the second operand is less than 136 bytes, at which point the operation continues as described below.

When the padding state, bit 55 of general register 0 is zero (indicating that padding has not yet been performed), the following occurs:

- Any remaining bytes of the second operand are padded on the right to form a 136-byte message block as described for the symbol "xp<n>" on page 7-205. Padding occurs even when there are no remaining bytes in the second operand and does not alter the contents of the second operand.
- The padding state is set to one, indicating that padding has been performed.
- The second-operand address in general register R_2 is incremented by the number of message bytes processed, and the second-operand length in general register R_2+1 is set to zero.
- The 136-byte padded message, is exclusive ORed with the contents of the leftmost 136 bytes of the state array (from the ICV in the parameter block or from the OCV resulting from the previous block's processing) to form an output chaining value that is used in the extended-output-function (XOF) processing. The rightmost 64 bytes of the state array are unchanged.

The padding operationis shown in Figure 7-263.

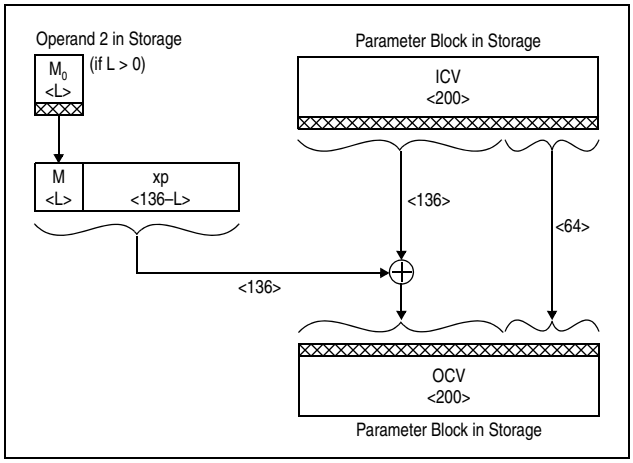


Figure 7-263. KLMD-SHAKE-256 Padding (L < 136)

Explanation:

⊠	The bit positions of each byte of the parameter block and the XOF block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
⊕	Bitwise exclusive-OR operation
<#>	Size of an item in bytes
ICV	Input chaining value (in the parameter block)
L	Length of the second operand (< 136 bytes)
M	Message block (when L > 0)
OCV	Output chaining value (used in XOF processing, below)
xp	SHAKE padding algorithm

Figure 7-263. KLMD-SHAKE-256 Padding (L < 136)

Depending on the number of second-operand blocks processed when padding is completed, either (a) the output chaining value is stored into the parameter block, and the instruction completes by setting condition code 3 (partial completion), or (b) the operation continues with extended-output-function (XOF) processing, as described below.

When the padding state is one (indicating that padding has been performed for the message, either by the current or a previous execution of the instruction), extended-output-function (XOF) processing is performed as follows.

1. If the first-operand length in general register $R_1 + 1$ is zero, then the output chaining value is stored into the parameter block, and the instruction completes with condition code 0. If the first-operand length is zero at the beginning of the instruction, then it is model dependent whether the ICV is fetched from the parameter block and stored back unmodified as the OCV.

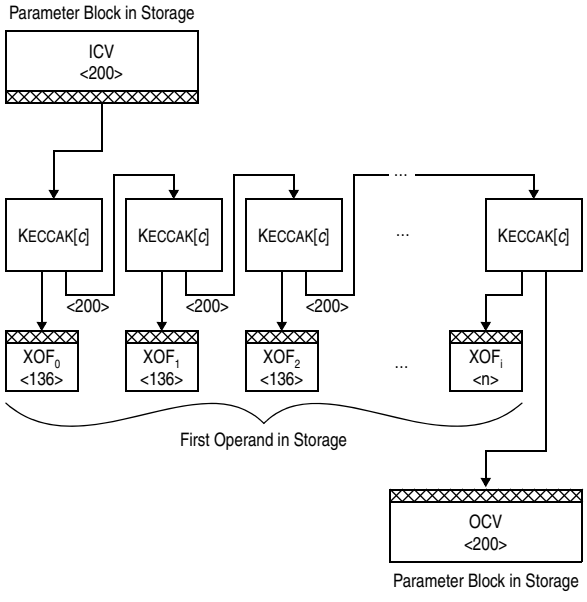
2. The KECCAK[*c*] function is invoked using the previous output-chaining value (OCV) as input, and replacing the output-chaining value.

3. General register R_1 contains the current address of the first operand, and general register $R_1 + 1$ contains the remaining length of the first operand. The number of bytes to be stored, *n*, is either the remaining first-operand length or 136, whichever is smaller.

   The first *n* bytes of the output-chaining value are stored at the first-operand location. See programming note 10 on page 7-200 regarding the numbering of the bits in the first operand as compared with the numbering of the bits in the state array.

   The first-operand address in general register R_1 is incremented by *n*, and the first-operand length general register $R_1 + 1$ is decremented by *n*.

Steps 1-3 of this process are repeated until the first-operand length becomes zero (in which case, the instruction completes with condition code 0) or until a CPU-determined number of bytes have been stored (in which case, the instruction completes with condition code 3). The output-chaining value is stored into bytes 0-199 of the parameter block regardless of whether condition code 0 or 3 is set. Figure 7-264 illustrates the XOF processing described above.



Explanation:

⊠	The bit positions of each byte of the parameter block and the XOF block(s) appear in storage in descending order from left to right, respective to their corresponding bit positions in the bytes of the KECCAK[*c*] state array.
<#>	Size of an item in bytes
ICV	Input chaining value (either from the parameter block or from the output of the padding operation).
OCV	Output chaining value; (the final OCV appears in the parameter block)
n	The residual length of the final block of the first operand (less than or equal to 136 bytes)
XOF_i<n>	Block i of the extended-output-function results in the first operand.

Figure 7-264. KLMD-SHAKE-256 Extended-Output-Function (XOF) Processing

Special Conditions

A specification exception is recognized and no other action is taken if any of the following occurs:

- Bit 56 of general register 0 is not zero.
- Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
- The R_2 field designates an odd-numbered register or general register 0.
- For the KLMD-SHAKE functions, either of the following is true:
  - The R_1 field designates an odd-numbered register, general register 0, or register R_2.
  - The second-operand length is nonzero, and the padding state is one.

Resulting Condition Code:

0 Normal completion
1 --
2 --
3 Partial completion

Program Exceptions:

- Access (fetch, operand 2 and message bit length; fetch and store, chaining value; store, operand 1)
- Operation (if the message-security assist is not installed)
- Specification
- Transaction constraint

1.-6.	Exceptions with the same priority as the priority of program-interruption conditions for the general case.
7.A	Access exceptions for second instruction halfword.
7.B	Operation exception.
7.C	Transaction constraint
8.a	Specification exception due to invalid function code, or invalid register number.
8.b	Specification exception due to nonzero second-operand length when the padding state is one (SHAKE functions only).

Figure 7-265. Priority of Execution: KLMD

9.	Condition code 0 due to second-operand length originally zero.
10.	Access exceptions for an access to the parameter block or second operand.
11.	Access exceptions for an access to the first operand (SHAKE functions only).
12.	Condition code 0 due to normal completion (for functions other than SHAKE, second-operand length originally nonzero, but stepped to zero; for SHAKE functions, second-operand length zero and first-operand length stepped to zero).
13.	Condition code 3 due to partial completion (for functions other than SHAKE, second-operand length still nonzero; for SHAKE functions, either first- or second-operand length still nonzero).

Figure 7-265. Priority of Execution: KLMD (Continued)

Programming Notes:

1. The performance of the query function (code 0) may be significantly slower than that of simply examining a location in storage. If the program needs to frequently test for the availability of a function, it should perform the query function once during initialization; subsequently it should examine the stored results of the query function in memory with an instruction such as TEST UNDER MASK.
2. Programming notes 2-10 for COMPUTE INTERMEDIATE MESSAGE DIGEST are also applicable to COMPUTE LAST MESSAGE DIGEST.
3. For the KLMD-SHAKE functions, when condition code 3 is set during the XOF processing, the first operand address and length in general registers R_1 and $R_1 + 1$, respectively, are updated such that the program can simply branch back to the instruction to continue the operation.

   For unusual situations, the CPU protects against endless reoccurrence for the no-progress case. Thus, the program can safely branch back to the instruction whenever condition code 3 is set with no exposure to an endless loop.
4. For the KLMD-SHA-1, KLMD-SHA-256, and KLMD-SHA-512 functions, if the length of the second operand is nonzero initially and condition code 0 is set, the registers are updated in the same manner as for condition code 3; the chaining value in this case is such that additional oper-

ands can be processed as if they were part of the same chain.

5. For the KLMD-SHA-1, KLMD-SHA-256, and KLMD-SHA-512 functions, when processing the last message part, the program must compute the length of the original message in bits and place this length value in the message-bit-length field of the parameter block, and use the COMPUTE LAST MESSAGE DIGEST instruction.

6. The COMPUTE LAST MESSAGE DIGEST instruction does not require the second operand to be a multiple of the data block size. It first processes complete blocks, and may set condition code 3 before processing all blocks. After processing all complete blocks, it then performs the padding operation including the remaining portion of the second operand. This may require one or two iterations of the designated block digest algorithm.

7. The COMPUTE LAST MESSAGE DIGEST instruction provides the SHA padding for messages that are a multiple of eight bits in length. If a SHA function is to be applied to a bit string which is not a multiple of eight bits, the program must perform the SHA padding and use the COMPUTE INTERMEDIATE MESSAGE DIGEST instruction.

8. The following applies to the message-bit length (MBL) in the parameter block of KLMD-SHA-1, KLMD-SHA-256, and KLMD-SHA-512.

   a. The MBL is completely independent of the second-operand length in general register R_2.

   b. Regardless of whether the instruction ends with condition code 0 or 3, the MBL is not decremented by the number of bytes processed.

   c. In normal usage, the MBL is expected to be eight times the total size of the message in bytes. If the program supplies a MBL that is not a multiple of eight, the results will be algorithmically correct, but may not be usable in any practical application.

   d. The secure-hash algorithm described in Reference [15.] on page xxx allows for message-bit length that are not multiples of eight. COMPUTE LAST MESSGE DIGEST requires the message-bit length to be a multiple of eight.

9. For the KLMD-SHAKE functions, the following applies:

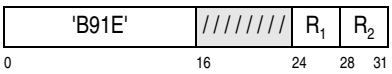
   a. The padding-state, bit 55 of general register 0, should be set to zero prior to the first execution of KLMD for a message, and the padding state should not be altered by the program for any subsequent executions of KLMD for the same message until normal completion occurs.

   b. If padding of the final (short or null) block of the second operand is performed when the first-operand length is zero, then the padded block is exclusive ORed with the contents of the state array, the result is stored as the output-chaining value in the parameter block, and the instruction completes with condition code 0. The KECCAK[c] function is not invoked in this case.

10. The KLMD SHA-3 and SHAKE functions perform padding according to the adopted NIST SHA-3 specification (see Reference [21.] on page xxx). Earlier draft versions of the SHA-3 specification used different padding bit sequences. Software that was designed according to earlier draft SHA-3 specifications can still benefit from the KIMD SHA-3 and SHAKE functions if the software performs the padding of the last message block.

COMPUTE MESSAGE AUTHENTICATION CODE

KMAC R_1,R_2 [RRE]



A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction and the R_1 field are ignored.

Bit positions 57-63 of general register 0 contain the function code. Figure 7-266 shows the assigned

function codes. All other function codes are unassigned. Bit 56 of general register 0 must be zero; otherwise, a specification exception is recognized. All other bits of general register 0 are ignored.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register 1 constitute the address.

The function codes for COMPUTE MESSAGE AUTHENTICATION CODE are as follows.

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
0	KMAC-Query *	16	—
1	KMAC-DEA *	16	8
2	KMAC-TDEA-128 *	24	8
3	KMAC-TDEA-192 *	32	8
9	KMAC-Encrypted-DEA	40	8
10	KMAC-Encrypted-TDEA-128	48	8
11	KMAC-Encrypted-TDEA-192	56	8
18	KMAC-AES-128	32	16
19	KMAC-AES-192	40	16
20	KMAC-AES-256	48	16
26	KMAC-Encrypted-AES-128	64	16
27	KMAC-Encrypted-AES-192	72	16
28	KMAC-Encrypted-AES-256	80	16

Explanation:

— Not applicable

* Function is also defined in the ESA/390 architectural mode and the ESA/390-compatibility mode. It is unpredictable whether other function codes are available in the ESA/390-compatibility mode.

Figure 7-266. Function Codes for COMPUTE MESSAGE AUTHENTICATION CODE

All other function codes are unassigned.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R_2 and $R_2 + 1$ are ignored.

For all other functions, the second operand is processed as specified by the function code using an initial chaining value in the parameter block, and the result replaces the initial chaining value. The operation also uses a cryptographic key in the parameter block. The operation proceeds until the end of the second-operand location is reached or a CPU-determined number of bytes have been processed, whichever occurs first. The result is indicated in the condition code.

The R_2 field designates an even-odd pair of general registers and must designate an even-numbered register other than general register 0; otherwise, a specification exception is recognized.

The location of the leftmost byte of the second operand is specified by the contents of the R_2 general register. The number of bytes in the second-operand location is specified in general register $R_2 + 1$.

As part of the operation, the address in general register R_2 is incremented by the number of bytes processed from the second operand, and the length in general register $R_2 + 1$ is decremented by the same number. The formation and updating of the address and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general register R_2 constitute the address of the second operand, and the contents of bit positions 0-39 are ignored; bits 40-63 of the updated address replace the corresponding bits in general register R_2, carries out of bit position 40 of the updated address are ignored and, the contents of bit positions 32-39 of general register R_2 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register R_2 constitute the address of the second operand, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated address replace the corresponding bits in general register R_2, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general register R_2 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register R_2 constitute the address of the second operand; bits 0-63 of the updated address replace the contents of general register R_2 and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_2 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of bit positions 32-63 of general register $R_2 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of general register $R_2 + 1$.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_2 and $R_2 + 1$, always remain unchanged.

Figure 7-267 shows the contents of the general registers just described.

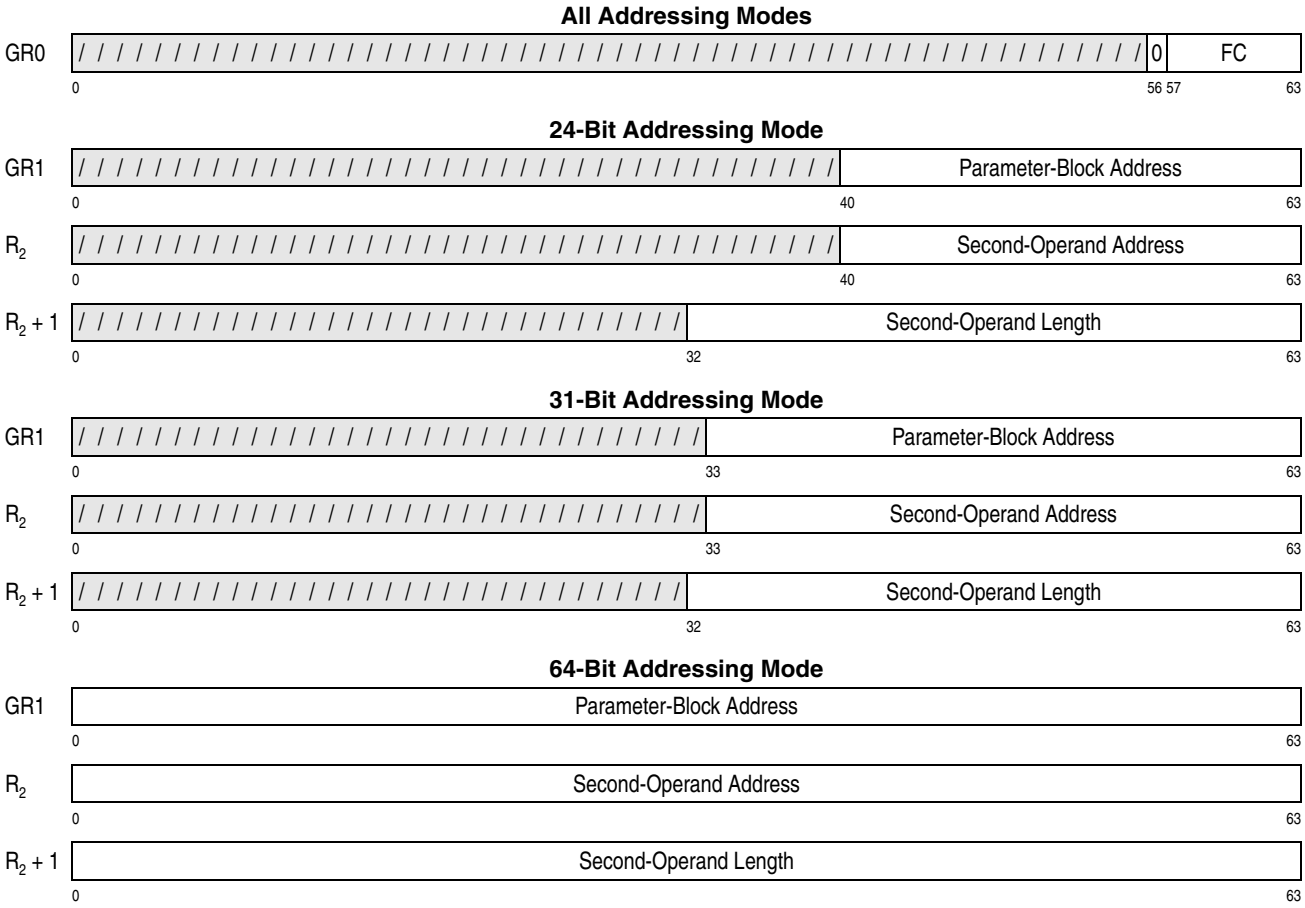


Figure 7-267. General Register Assignment for KMAC

In the access-register mode, access registers 1 and R_2 specify the address spaces containing the parameter block and second operand, respectively.

The result is obtained as if processing starts at the left end of the second operand and proceeds to the right, block by block. The operation is ended when all source bytes in the second operand have been processed (called normal completion), or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

When the initial-chaining-value field overlaps any portion of the second operand, the result in the chaining-value field is unpredictable.

Normal completion occurs when the number of bytes in the second operand as specified in general register $R_2 + 1$ have been processed.

When the operation ends due to normal completion, condition code 0 is set and the resulting value in $R_2 + 1$ is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in $R_2 + 1$ is nonzero.

When the second-operand length is initially zero, the second operand and the parameter block are not accessed, general registers R_2 and $R_2 + 1$ are not changed, and condition code 0 is set.

A PER storage-alteration event may be recognized for the portion of the parameter block that is stored. A PER zero-address-detection event may be recognized for the second-operand location and for the parameter block. When PER events are detected for one or more locations, it is unpredictable which location is identified in the PER access identification (PAID) and PER ASCE ID (AI).

As observed by this CPU, other CPUs, and channel programs, references to the parameter block and storage operand may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

For functions that perform a comparison of the encrypted cryptographic key with the wrapping-key verification-pattern register, it is unpredictable whether access exceptions and PER zero-address-detection events are recognized for the first and second operands when the comparison results in a mismatch.

Access exceptions may be reported for a larger portion of the second operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of the second operand nor for locations more than 4K bytes beyond the current location being processed.

Symbols Used in Function Descriptions

The following symbols are used in the subsequent description of the COMPUTE MESSAGE AUTHENTICATION CODE functions. For data-encryption-algorithm (DEA) functions, the DEA-key-parity bit in each byte of the DEA key is ignored, and the operation proceeds normally, regardless of the DEA-key parity of the key. Further description of the data-encryption algorithm may be found in Reference [13.] on page xxx. Further description of the AES standard may be found in Reference [14.] on page xxx.

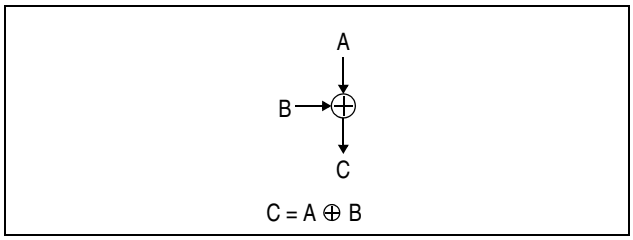
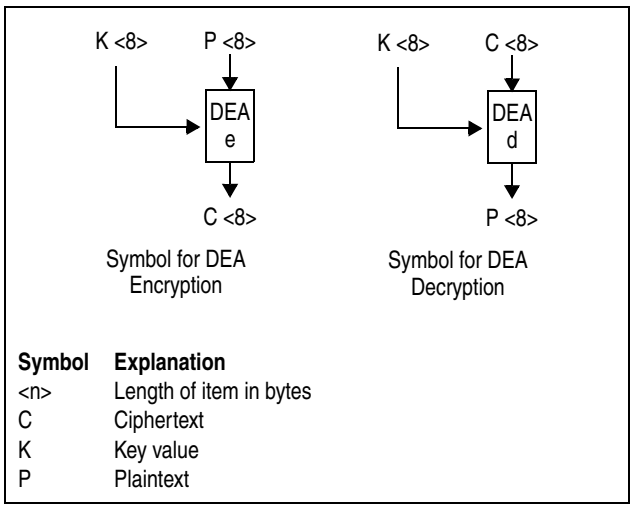
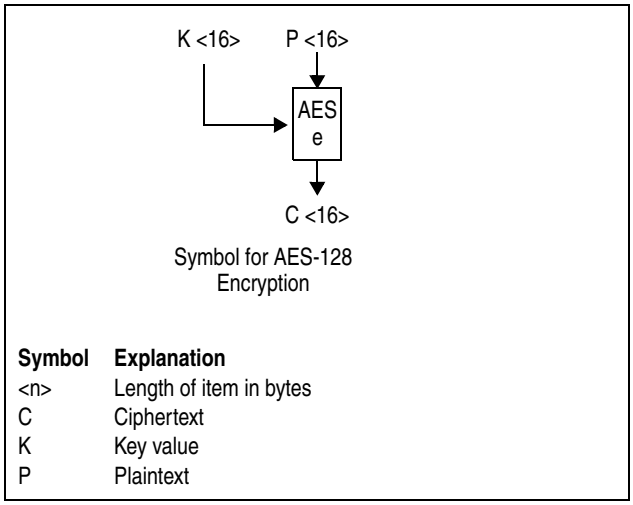


Figure 7-268. Symbol For Bit-Wise Exclusive OR



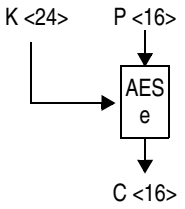
Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-269. Symbols for DEA Encryption and Decryption



Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

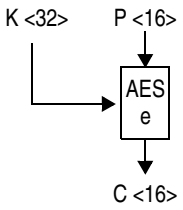
Figure 7-270. Symbols for AES-128 Encryption



Symbol for AES-192 Encryption

Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-271. Symbols for AES-192 Encryption



Symbol for AES-256 Encryption

Symbol	Explanation
<n>	Length of item in bytes
C	Ciphertext
K	Key value
P	Plaintext

Figure 7-272. Symbols for AES-256 Encryption

KMAC-Query (Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-267 on page 7-221.

The parameter block used for the function has the following format:

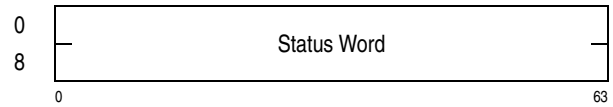


Figure 7-273. Parameter Block for KMAC-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the KMAC instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KMAC-Query function completes; condition codes 1 and 3 are not applicable to this function.

KMAC-DEA (Function Code 1)

KMAC-Encrypted-DEA (Function Code 9)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-267 on page 7-221.

The parameter block used for the KMAC-DEA function has the following format:

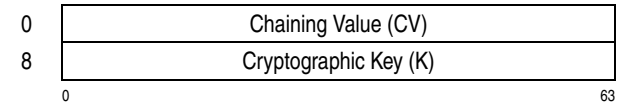


Figure 7-274. Parameter Block for KMAC-DEA

For the KMAC-DEA function, the initial chaining value is in byte offsets 0-7 of the parameter block and the cryptographic key is in byte offsets 8-15 of the parameter block.

The parameter block used for the KMAC-Encrypted-DEA function has the following format:

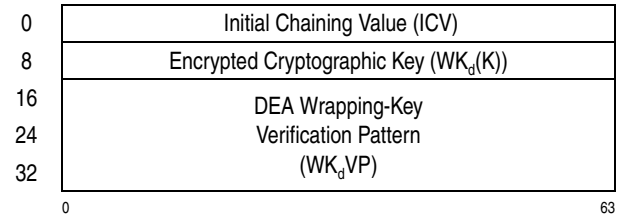


Figure 7-275. Parameter Block for KMAC-Encrypted-DEA

For the KMAC-Encrypted-DEA function, the contents of byte offsets 16-39 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-7 of the parameter block contain the initial chaining value, and the contents of byte offsets 8-15 of the parameter block are deciphered using the DEA wrapping key to obtain the 64-bit cryptographic key. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.)

The description in the following paragraph applies to both functions.

The message authentication code for the 8-byte message blocks (M1, M2, …, Mn) in operand 2 is computed using the DEA algorithm with the 64-bit cryptographic key and the 64-bit initial chaining value.

The message authentication code, also called the output chaining value (OCV), is stored in the initial-chaining-value field of the parameter block. The operation is shown in Figure 7-276.

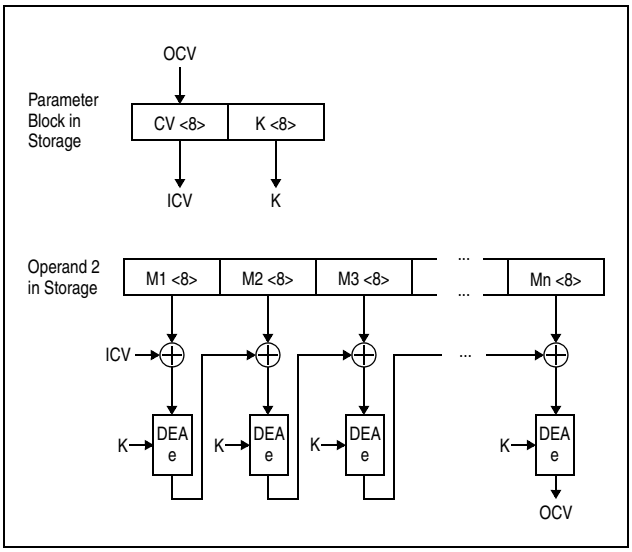


Figure 7-276. MAC Computation Using 64-bit DEA Key

KMAC-TDEA-128 (Function Code 2)

KMAC-Encrypted-TDEA-128 (Function Code 10)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-267 on page 7-221.

The parameter block used for the KMAC-TDEA-128 function has the following format:

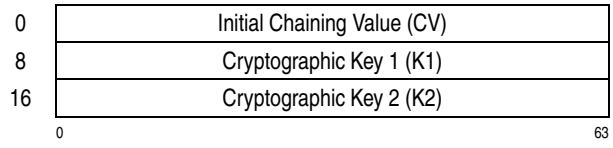


Figure 7-277. Parameter Block for KMAC-TDEA-128

For the KMAC-TDEA-128 function, the initial chaining value is in byte offsets 0-7 of the parameter block and the cryptographic key is in byte offsets 8-23 of the parameter block.

The parameter block used for the KMAC-Encrypted-TDEA-128 function has the following format:

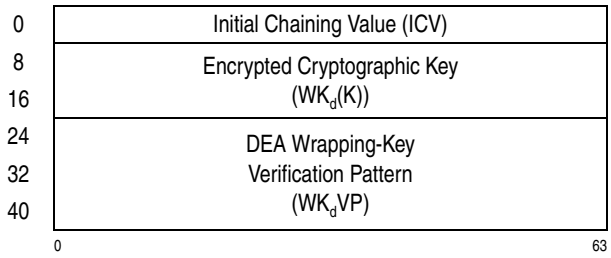


Figure 7-278. Parameter Block for KMAC-Encrypted-TDEA-128

For the KMAC-Encrypted-TDEA-128 function, the contents of byte offsets 24-47 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-7 of the parameter block contain the initial chaining value, and the contents of byte offsets 8-23 of the parameter block are deciphered using the DEA wrapping key to obtain the 128-bit cryptographic key, K. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.)

The description in the following paragraph applies to both functions.

The message authentication code for the 8-byte message blocks (M1, M2, …, Mn) in operand 2 is computed using the TDEA algorithm with the two 64-bit cryptographic keys and the 64-bit initial chaining value.

The message authentication code, also called the output chaining value (OCV), is stored in the initial-

chaining-value field of the parameter block. The operation is shown in Figure 7-279.

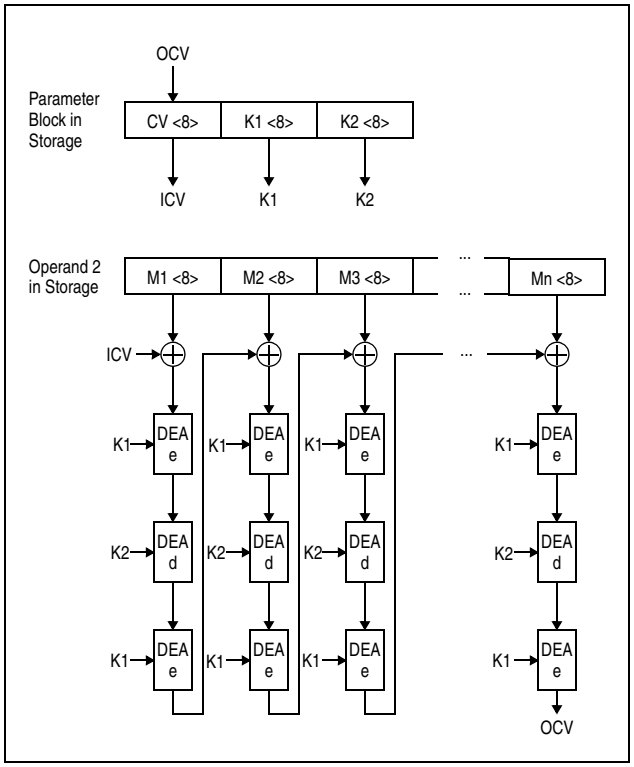


Figure 7-279. MAC Computation Using 128-bit TDEA Key

KMAC-TDEA-192 (Function Code 3)

KMAC-Encrypted-TDEA-192 (Function Code 11)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-267 on page 7-221.

The parameter block used for the KMAC-TDEA-192 function has the following format:

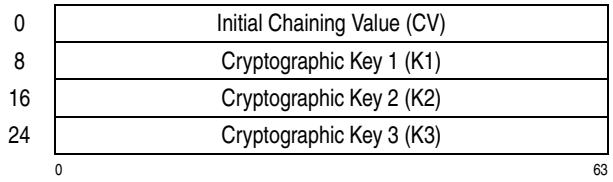


Figure 7-280. Parameter Block for KMAC-TDEA-192

For the KMAC-TDEA-192 function, the initial chaining value is in byte offsets 0-7 of the parameter block and the cryptographic key is in byte offsets 8-31 of the parameter block.

The parameter block used for the KMAC-Encrypted-TDEA-192 function has the following format:

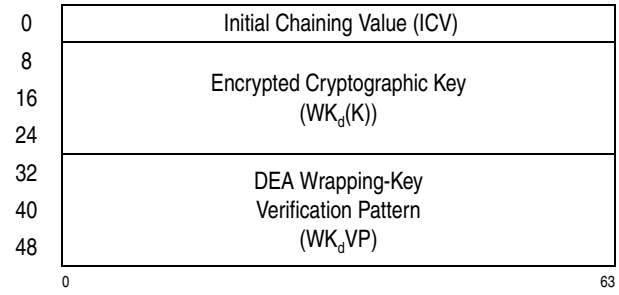


Figure 7-281. Parameter Block for KMAC-Encrypted-TDEA-192

For the KMAC-Encrypted-TDEA-192 function, the contents of byte offsets 32-55 of the parameter block are compared with the contents of the DEA wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-7 of the parameter block contain the initial chaining value, and the contents of byte offsets 8-31 of the parameter block are deciphered using the DEA wrapping key to obtain the 192-bit cryptographic key, K. (See the section, “Protection of Cryptographic Keys” on page 7-424, for details.)

The description in the following paragraph applies to both functions.

The message authentication code for the 8-byte message blocks (M1, M2, …, Mn) in operand 2 is computed using the TDEA algorithm with the three 64-bit cryptographic keys and the 64-bit chaining value.

The message authentication code, also called the output chaining value (OCV), is stored in the initial-

chaining-value field of the parameter block. The operation is shown in Figure 7-282.

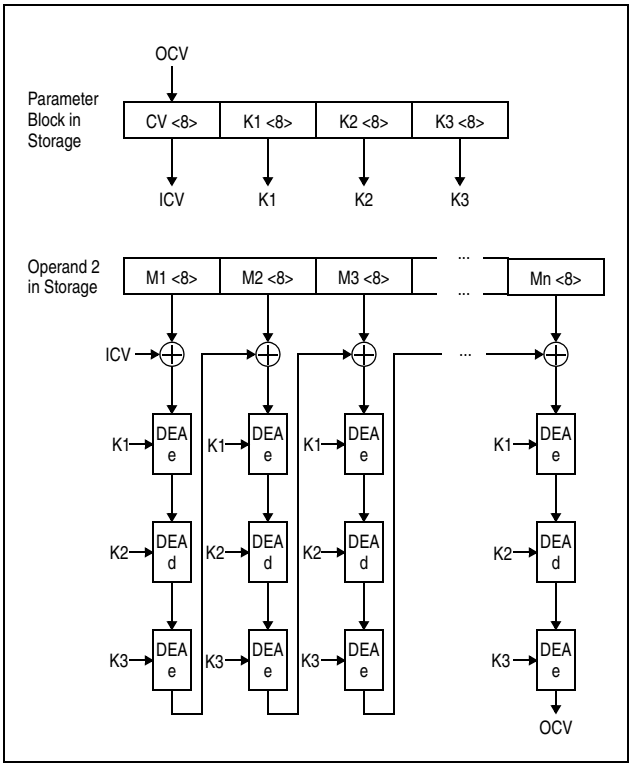


Figure 7-282. MAC Computation Using 192-bit TDEA Key

KMAC-AES-128 (Function Code 18)

KMAC-Encrypted-AES-128 (Function Code 26)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-267 on page 7-221.

The parameter block used for the KMAC-AES-128 function has the following format:

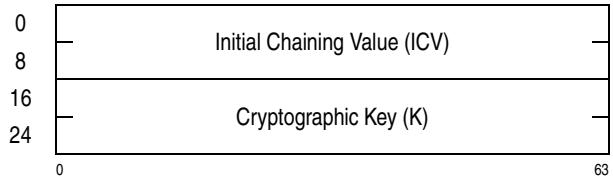


Figure 7-283. Parameter Block for KMAC-AES-128

For the KMAC-AES-128 function, the initial-chaining value is in byte offsets 0-15 of the parameter block and the cryptographic key is in byte offsets 16-31of the parameter block.

The parameter block used for the KMAC-Encrypted-AES-128 function has the following format:

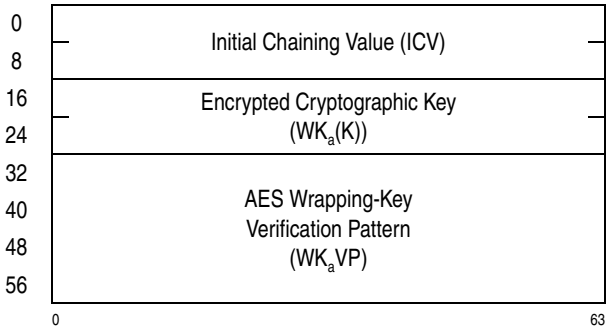


Figure 7-284. Parameter Block for KMAC-Encrypted-AES-128

For the KMAC-Encrypted-AES-128 function, the contents of byte offsets 32-63 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-15 of the parameter block contain the initial chaining value, and the contents of byte offsets 16-31 of the parameter block are deciphered using the AES wrapping key to obtain the 128-bit cryptographic key, K. (See the section “Protection of Cryptographic Keys” on page 7-424 for details.)

The description in the following paragraph applies to both functions.

The message authentication code for the 16-byte message blocks (M1, M2, …, Mn) in operand 2 is computed using the AES algorithm with the 128-bit cryptographic key and the 128-bit chaining value.

The message authentication code, also called the output chaining value (OCV), is stored in the initial-

chaining-value field of the parameter block. The operation is shown in Figure 7-285.

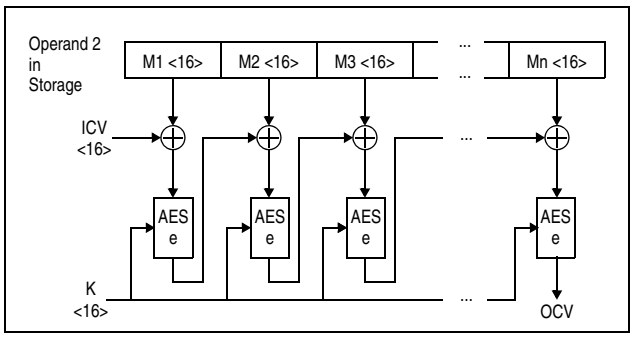


Figure 7-285. KMAC-AES-128

KMAC-AES-192 (Function Code 19)

KMAC-Encrypted-AES-192 (Function Code 27)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-267 on page 7-221.

The parameter block used for the KMAC-AES-192 function has the following format:

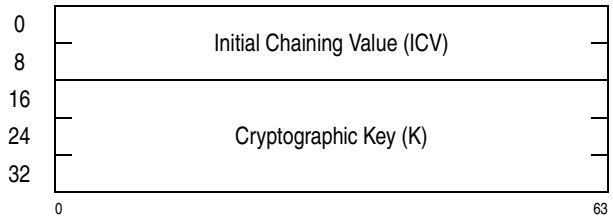


Figure 7-286. Parameter Block for KMAC-AES-192

For the KMAC-AES-192 function, the initial-chaining value is in byte offsets 0-15 of the parameter block and the cryptographic key is in byte offsets 16-39 of the parameter block.

The parameter block used for the KMAC-Encrypted-AES-192 function has the following format:

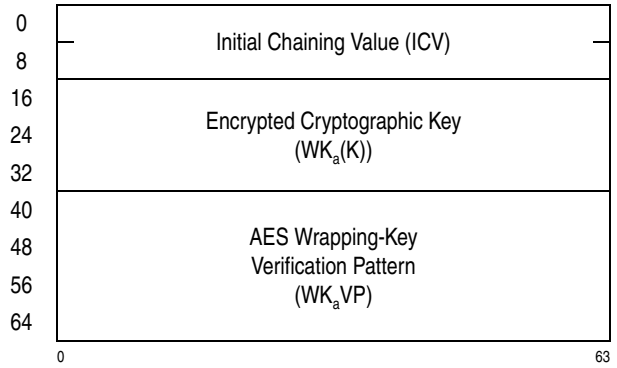


Figure 7-287. Parameter Block for KMAC-Encrypted-AES-192

For the KMAC-Encrypted-AES-192 function, the contents of byte offsets 40-71 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-15 of the parameter block contain the initial chaining value, and the contents of byte offsets 16-39 of the parameter block are deciphered using the AES wrapping key to obtain the 192-bit cryptographic key, K. (See the section “Protection of Cryptographic Keys” on page 7-424 for details.)

The description in the following paragraph applies to both functions.

The message authentication code for the 16-byte message blocks (M1, M2, …, Mn) in operand 2 is computed using the AES algorithm with the 192-bit cryptographic key and the 128-bit chaining value.

The message authentication code, also called the output chaining value (OCV), is stored in the initial-

chaining-value field of the parameter block. The operation is shown in Figure 7-288.

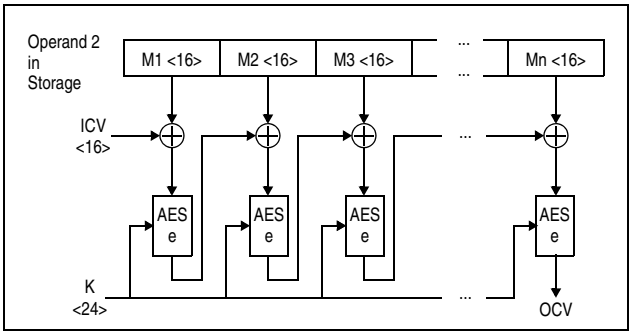


Figure 7-288. KMAC-AES-192

KMAC-AES-256 (Function Code 20)

KMAC-Encrypted-AES-256 (Function Code 28)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-267 on page 7-221.

The parameter block used for the KMAC-AES-256 function has the following format:

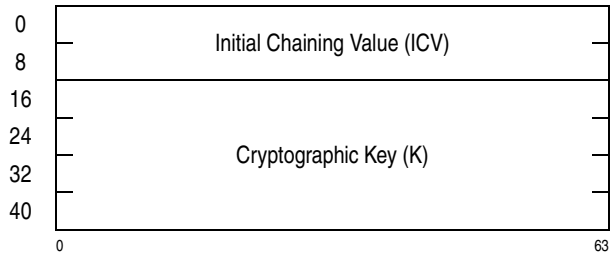


Figure 7-289. Parameter Block for KMAC-AES-256

For the KMAC-AES-256 function, the initial-chaining value is in byte offsets 0-15 of the parameter block and the cryptographic key is in byte offsets 16-47 of the parameter block.

The parameter block used for the KMAC-Encrypted-AES-256 function has the following format:

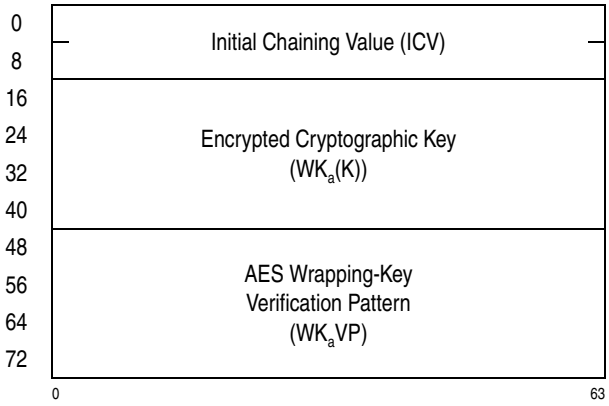


Figure 7-290. Parameter Block for KMAC-Encrypted-AES-256

For the KMAC-Encrypted-AES-256 function, the contents of byte offsets 48-79 of the parameter block are compared with the contents of the AES wrapping-key verification-pattern register. If they mismatch, the first-operand and parameter-block locations remain unchanged, and the operation is completed by setting condition code 1. If they match, byte offsets 0-15 of the parameter block contain the initial chaining value, and the contents of byte offsets 16-47 of the parameter block are deciphered using the AES wrapping key to obtain the 256-bit cryptographic key, K. (See the section “Protection of Cryptographic Keys” on page 7-424 for details.)

The description in the following paragraph applies to both functions.

The message authentication code for the 16-byte message blocks (M1, M2, …, Mn) in operand 2 is computed using the AES algorithm with the 256-bit cryptographic key and the 128-bit chaining value.

The message authentication code, also called the output chaining value (OCV), is stored in the initial-

chaining-value field of the parameter block. The operation is shown in Figure 7-291.

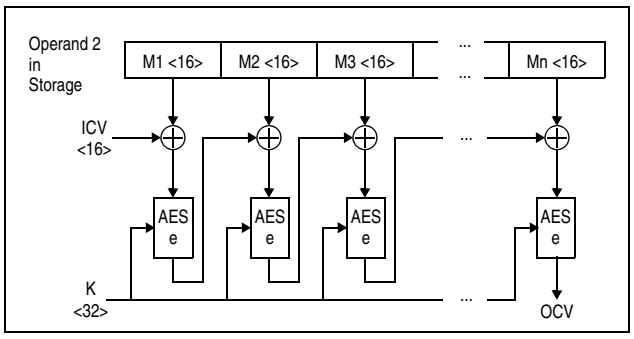


Figure 7-291. KMAC-AES-256

Special Conditions

A specification exception is recognized and no other action is taken if any of the following occurs:

1. Bit 56 of general register 0 is not zero.
2. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
3. The R_2 field designates an odd-numbered register or general register 0.
4. The second-operand length is not a multiple of the data block size of the designated function (see Figure 7-266 on page 7-220 to determine the data block size for COMPUTE MESSAGE AUTHENTICATION CODE functions).

Resulting Condition Code:

0 Normal completion
1 Verification-pattern mismatch
2 --
3 Partial completion

Program Exceptions:

- Access (fetch, operand 2, cryptographic key, and wrapping-key verification pattern; fetch and store, chaining value)
- Operation (if the message-security assist is not installed)
- Specification
- Transaction constraint

1.-6.	Exceptions with the same priority as the priority of program-interruption conditions for the general case.
7.A	Access exceptions for second instruction halfword.
7.B	Operation exception.
7.C	Transaction constraint
8.	Specification exception due to invalid function code or invalid register number.
9.	Specification exception due to invalid operand length.
10.	Condition code 0 due to second-operand length originally zero.
11.A.1	Access exceptions for an access to the parameter block.
11.A.2.	Condition code 1 due to verification-pattern mismatch.
11.B	Access exceptions for an access to the parameter block or second operand.
12.	Condition code 0 due to normal completion (second-operand length originally nonzero, but stepped to zero).
13.	Condition code 3 due to partial completion (second-operand length still nonzero).

Figure 7-292. Priority of Execution: KMAC

Programming Notes:

1. The performance of the query function (code 0) may be significantly slower than that of simply examining a location in storage. If the program needs to frequently test for the availability of a function, it should perform the query function once during initialization; subsequently it should examine the stored results of the query function in memory with an instruction such as TEST UNDER MASK.
2. When condition code 3 is set, the second operand address and length in general registers R_2 and $R_2 + 1$, respectively, and the initial-chaining-value in the parameter block are usually updated such that the program can simply branch back to the instruction to continue the operation. For unusual situations, the CPU protects against endless reoccurrence for the no-progress case.

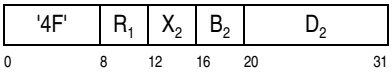
Thus, the program can safely branch back to the instruction whenever condition code 3 is set with no exposure to an endless loop.

3. If the length of the second operand is nonzero initially and condition code 0 is set, the registers are updated in the same manner as for condition code 3; the initial chaining value in this case is such that additional operands can be processed as if they were part of the same chain.

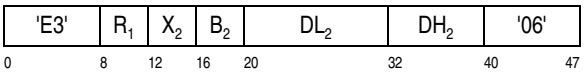
4. Before processing the first part of a message, the program must set the initial values for the initial-chaining-value field. To comply with ANSI X9.9, ANSI X9.19, or Reference [17.] on page xxx, the initial chaining value shall be set to all binary zeros.

CONVERT TO BINARY

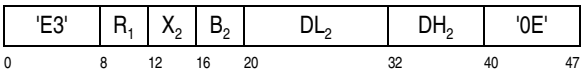
CVB $R_1,D_2(X_2,B_2)$ [RX-a]



CVBY $R_1,D_2(X_2,B_2)$ [RXY-a]



CVBG $R_1,D_2(X_2,B_2)$ [RXY-a]



The second operand is changed from decimal to binary, and the result is placed at the first-operand location.

For CONVERT TO BINARY (CVB, CVBY), the second operand occupies eight bytes in storage, and, for CONVERT TO BINARY (CVBG), the second operand occupies sixteen bytes in storage. The second operand has the format of signed-packed-decimal data, as described in Chapter 8, “Decimal Instructions.” It is checked for valid sign and digit codes, and a general-operand data exception is recognized when an invalid code is detected.

For CONVERT TO BINARY (CVB, CVBY), the result of the conversion is a 32-bit signed binary integer, which is placed in bit positions 32-63 of general register R_1. Bits 0-31 of the register remain unchanged. The maximum positive number that can be converted and still be contained in 32 bit positions is 2,147,483,647; the maximum negative number (the negative number with the greatest absolute value) that can be converted is -2,147,483,648.

For any decimal number outside this range, the operation is completed by placing the 32 rightmost bits of the binary result in the register, and a fixed-point-divide exception is recognized.

For CONVERT TO BINARY (CVBG), the result of the conversion is a 64-bit signed binary integer, which is placed in bit positions 0-63 of general register R_1. The maximum positive number that can be converted and still be contained in a 64-bit register is 9,223,372,036,854,775,807; the maximum negative number (the negative number with the greatest absolute value) that can be converted is -9,223,372,036,854,775,808. For any decimal number outside this range, a fixed-point-divide exception is recognized, and the operation is suppressed.

The displacement for CVB is treated as a 12-bit unsigned binary integer. The displacement for CVBY and CVBG is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2)
- Data with DXC 0, general operand
- Fixed-point divide
- Operation (CVBY, if the long-displacement facility is not installed)
- Transaction constraint

Programming Notes:

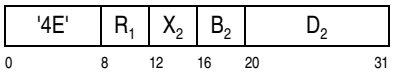
1. An example of the use of the CONVERT TO BINARY instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. When the second operand is negative, the result is in two’s-complement notation.

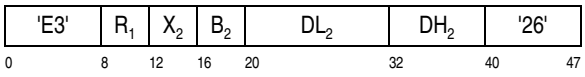
3. The storage-operand references for CONVERT TO BINARY may be multiple-access references. (See “Storage-Operand Consistency” on page 5-125.)

CONVERT TO DECIMAL

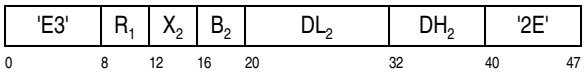
CVD $R_1,D_2(X_2,B_2)$ [RX-a]



CVDY $R_1,D_2(X_2,B_2)$ [RXY-a]



CVDG $R_1,D_2(X_2,B_2)$ [RXY-a]



The first operand is changed from binary to decimal, and the result is stored at the second-operand location.

For CONVERT TO DECIMAL (CVD, CVDY), the first operand is treated as a 32-bit signed binary integer, and the result occupies eight bytes in storage. For CONVERT TO DECIMAL (CVDG), the first operand is treated as a 64-bit signed binary integer, and the result occupies sixteen bytes in storage.

The result is in the format for signed-packed-decimal data, as described in Chapter 8, "Decimal Instructions." The rightmost four bits of the result represent the sign. A positive sign is encoded as 1100; a negative sign is encoded as 1101.

The displacement for CVD is treated as a 12-bit unsigned binary integer. The displacement for CVDY and CVDG is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

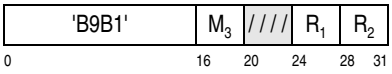
- Access (store, operand 2)
- Operation (CVDY, if the long-displacement facility is not installed)
- Transaction constraint

Programming Notes:

1. An example of the use of the CONVERT TO DECIMAL instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. For CVD and CVDY, the number to be converted is a 32-bit signed binary integer obtained from a general register. Since 15 decimal digits are available for the result, and the decimal equivalent of 31 bits requires at most 10 decimal digits, an overflow cannot occur. Similarly, for CVDG, 31 decimal digits are available, the decimal equivalent of 63 bits is at most 19 digits, and an overflow cannot occur.
3. The storage-operand references for CONVERT TO DECIMAL may be multiple-access references. (See "Storage-Operand Consistency" on page 5-125.)

CONVERT UTF-16 TO UTF-32

CU24 $R_1,R_2[,M_3]$ [RRF-c]



The two-byte UTF-16 (Unicode) characters of the second operand are converted to UTF-32 characters and placed at the first-operand location. The UTF-32 characters are four bytes. The operation proceeds until the end of the first or second operand is reached, a CPU-determined number of characters have been converted, or an invalid Unicode character is encountered, whichever occurs first. The result is indicated in the condition code.

The R_1 and R_2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and the second operand is designated by the contents of general registers R_1 and R_2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the entire contents of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-293.

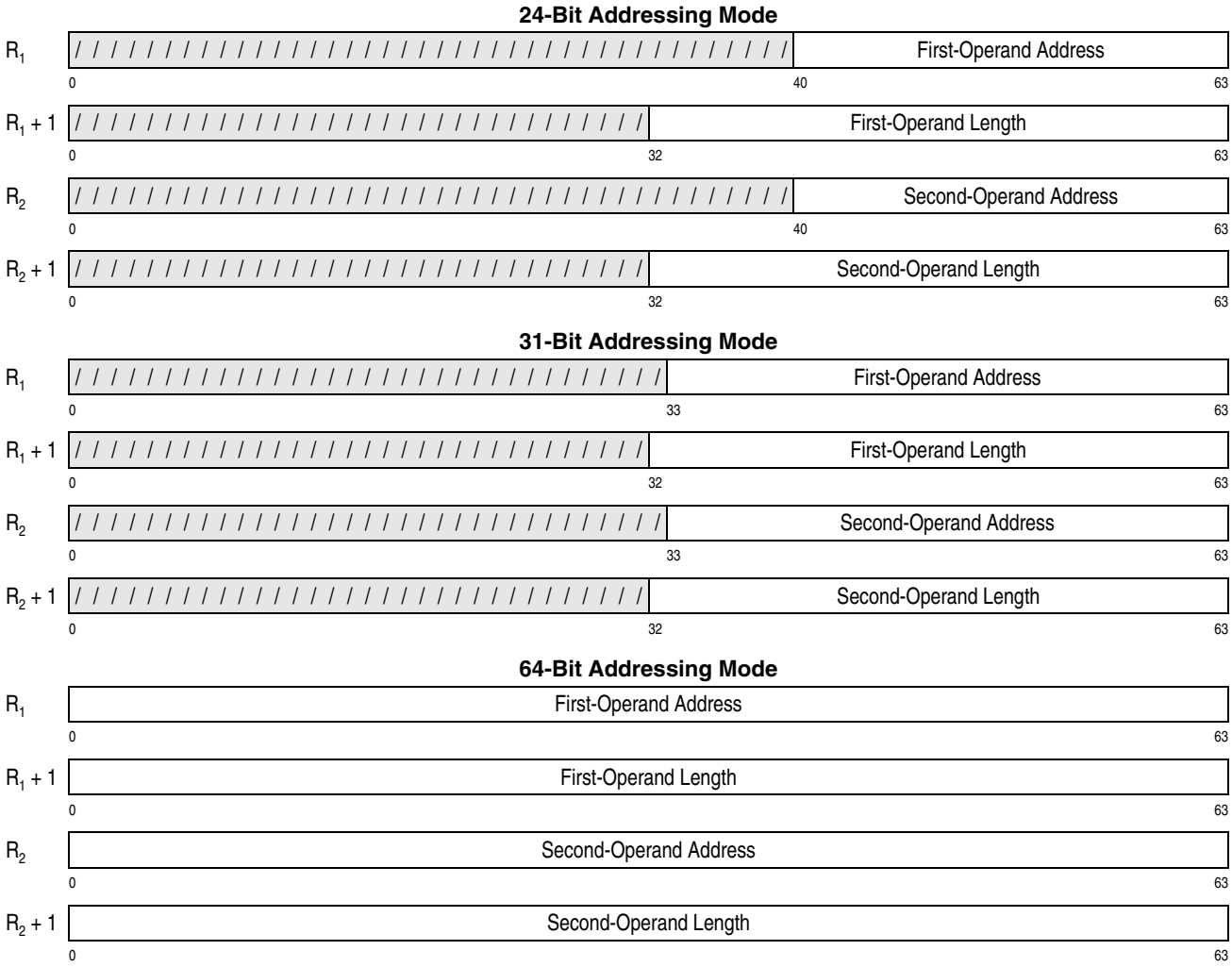
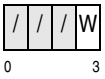


Figure 7-293. Register Contents for CONVERT UTF-16 TO UTF-32

When the ETF3-enhancement facility is installed, the M_3 field has the following format:



The bits of the M_3 field are defined as follows:

- ***Unassigned:*** Bits 0-2 are unassigned and should contain zeros; otherwise, the program may not operate compatibly in the future.
- ***Enhanced Well-Formedness-Checking (W):*** The W bit, bit 3 of the M_3 field, controls whether enhanced well-formedness checking of the Unicode characters is performed. When the W bit is zero, enhanced checking is not performed. When the W bit is one, enhanced checking is performed, as described below.

When the ETF3-enhancement facility is not installed, the M_3 field is ignored.

The characters of the second operand are selected one by one for conversion, proceeding left to right. The characters resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted, a CPU-determined number of second-operand characters have been converted, or on some models when the W bit of the M_3 field is one, an invalid Unicode character is detected.

To show the method of converting a UTF-16 character to a UTF-32 character, the bits of a Unicode character are identified by letters as follows:

Unicode Character Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	a b c d e f g h	i j k l m n o p

In the case of a Unicode surrogate pair, which is a character pair consisting of a character called a high surrogate followed by a character called a low surrogate, the bits are identified by letters as follows:

Unicode High Surrogate Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	1 1 0 1 1 0 a b	c d e f g h i j

Unicode Low Surrogate Bit Numbers	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Identifying Bit Letters	1 1 0 1 1 1 k l	m n o p q r s t

A low surrogate (that is, a Unicode character in the range of DC00 to DFFF hex) that is not immediately preceded by a high surrogate (that is, a Unicode character in the range of D800 to DBFF hex) is called an isolated low surrogate.

Any Unicode character in the range 0000 to D7FF or E000 to FFFF hex is converted to a four-byte UTF-32 character as follows:

Unicode Character	a b c d e f g h	i j k l m n o p		
UTF-32 Character	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	a b c d e f g h	i j k l m n o p

On some models, an isolated low surrogate (that is, a Unicode character in the range of DC00 to DFFF hex) is also converted to a four-byte UTF-32 character as shown above, regardless of whether the ETF3-enhancement facility is installed and regardless of the W bit of the M_3 field. On other models, when the ETF3-enhancement facility is installed, the W bit of the M_3 field is one, and the next Unicode character is an isolated low surrogate, condition code 2 is set.

Any Unicode surrogate pair starting with a high surrogate in the range D800 to DBFF hex is converted to a four-byte UTF-32 character as follows:

Unicode Characters	1 1 0 1 1 0 a b	c d e f g h i j	1 1 0 1 1 1 k l	m n o p q r s t
UTF-32 Character	0 0 0 0 0 0 0 0	0 0 0 u v w x y	e f g h i j k l	m n o p q r s t
where uvwxy = abcd + 1				

When the ETF3-enhancement facility is not installed, or when the W bit of the M_3 field is zero, the first six bits of an expected Unicode low surrogate are ignored. When the ETF3-enhancement facility is installed, and the W bit is one, the first six bits of the Unicode low surrogate must contain 110111 binary; otherwise, the Unicode low surrogate is invalid, and condition code 2 is set.

The second-operand location is considered exhausted when it does not contain at least two remaining bytes or at least four remaining bytes when the first two bytes are a Unicode high surrogate. The first-operand location is considered exhausted when it does not contain at least four remaining bytes.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been converted, condition code 3 is set.

When the conditions for setting condition codes 1 and 2 are both met, condition code 2 is set.

When the operation is completed, the contents of general register $R_2 + 1$ are decremented by the number of bytes converted, and the contents of general register R_2 are incremented by the same number. Also, the contents of general register $R_1 + 1$ are decremented by the number of bytes placed at the first-operand location, and the contents of general register R_1 are incremented by the same number. When general registers R_1 and R_2 are updated in the 24-bit

or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_2, and $R_2 + 1$, always remain unchanged.

When condition code 2 is set, the following applies:

- When a Unicode high surrogate immediately precedes an invalid low surrogate, general register R_2 contains the address of the Unicode high surrogate.
- When an isolated Unicode low surrogate is detected, general register R_2 contains the address of the isolated low surrogate.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

When the R_1 register is the same register as the R_2 register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

Resulting Condition Code:

0 Entire second operand processed
1 End of first operand reached
2 Invalid Unicode low surrogate
3 CPU-determined number of characters converted

Program Exceptions:

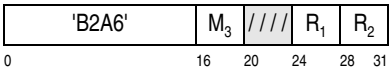
- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 3 is not installed)
- Specification
- Transaction constraint

Programming Notes:

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.
2. The storage-operand references of CONVERT UTF-16 TO UTF-32 may be multiple-access references. (See “Storage-Operand Consistency” on page 5-125.)
3. The CONVERT UTF-16 TO UTF-32 instruction supports UTF-16 and UTF-32 characters only in the big-endian encoding.
4. As observed by this CPU, other CPUs, and the I/O subsystem, inconsistent results may be briefly stored in the first-operand location. See the section “Effects of CPU Retry” on page 11-3 for further details.
5. Determining that isolated Unicode low surrogates are invalid is a model-dependent behavior that is not available on the IBM z13 and earlier models.

CONVERT UTF-16 TO UTF-8
CONVERT UNICODE TO UTF-8

CU21 $R_1,R_2[,M_3]$ [RRF-c]
CUUTF $R_1,R_2[,M_3]$ [RRF-c]



The two-byte Unicode characters of the second operand are converted to UTF-8 characters and placed at the first-operand location. The UTF-8 characters are one, two, three, or four bytes, depending on the Unicode characters that are converted. The operation proceeds until the end of the first or second operand is reached or a CPU-determined number of characters have been converted, whichever occurs first. The result is indicated in the condition code.

The R_1 and R_2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and the second operand is designated by the contents of general registers R_1 and R_2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the entire contents of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-294.

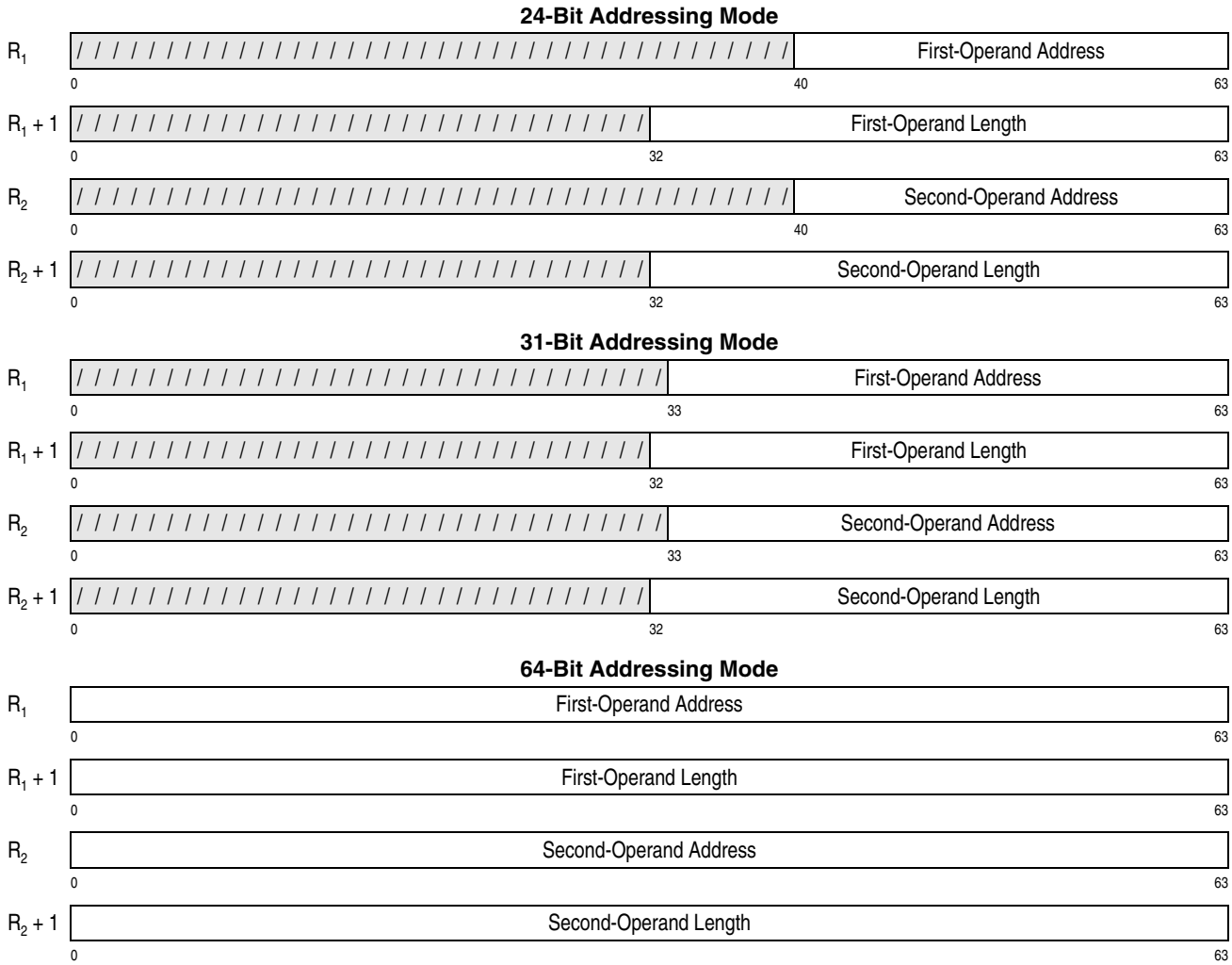
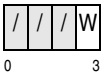


Figure 7-294. Register Contents for CONVERT UTF-16 TO UTF-8

When the ETF3-enhancement facility is installed, the M_3 field has the following format:



The bits of the M_3 field are defined as follows:

- ***Unassigned:*** Bits 0-2 are unassigned and should contain zeros; otherwise, the program may not operate compatibly in the future.
- ***Enhanced Well-Formedness-Checking (W):*** The W bit, bit 3 of the M_3 field, controls whether enhanced well-formedness checking of the Unicode characters is performed. When the W bit is zero, enhanced checking is not performed. When the W bit is one, enhanced checking is performed, as described below.

When the ETF3-enhancement facility is not installed, the M_3 field is ignored. When the ETF3-enhancement facility is installed in the z/Architecture architectural mode, it is unpredictable whether the M_3 field is ignored in the ESA/390-compatibility mode.

The characters of the second operand are selected one by one for conversion, proceeding left to right. The bytes resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted, a CPU-determined number of second-operand characters have been converted, or an invalid Unicode character is encountered in the second operand.

To show the method of converting a Unicode character to a UTF-8 character, the bits of a Unicode character are identified by letters as follows:

Unicode Character Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	a b c d e f g h	i j k l m n o p

In the case of a Unicode surrogate pair, which is a character pair consisting of a character called a high surrogate followed by a character called a low surrogate, the bits are identified by letters as follows:

Unicode High Surrogate Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	1 1 0 1 1 0 a b	c d e f g h i j

Unicode Low Surrogate Bit Numbers	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Identifying Bit Letters	1 1 0 1 1 1 k l	m n o p q r s t

A low surrogate (that is, a Unicode character in the range of DC00 to DFFF hex) that is not immediately preceded by a high surrogate (that is, a Unicode character in the range of D800 to DBFF hex) is called an isolated low surrogate.

Any Unicode character in the range 0000 to 007F hex is converted to a one-byte UTF-8 character as follows:

Unicode Character	0 0 0 0 0 0 0 0	0 j k l m n o p
UTF-8 Character	0 j k l m n o p	

Any Unicode character in the range 0080 to 07FF hex is converted to a two-byte UTF-8 character as follows:

Unicode Character	0 0 0 0 0 f g h	i j k l m n o p
UTF-8 Character	1 1 0 f g h i j	1 0 k l m n o p

Any Unicode character in the range 0800 to D7FF or E000 to FFFF hex is converted to a three-byte UTF-8 character as follows:

Unicode Character	a b c d e f g h	i j k l m n o p	
UTF-8 Character	1 1 1 0 a b c d	1 0 e f g h i j	1 0 k l m n o p

On some models, an isolated low surrogate (that is, a Unicode character in the range of DC00 to DFFF hex) is also converted to a three-byte UTF-8 character as shown above, regardless of whether the ETF3-enhancement facility is installed and regardless of the W bit of the M_3 field. On other models, when the ETF3-enhancement facility is installed, the W bit of the M_3 field is one, and the next Unicode character is an isolated low surrogate, condition code 2 is set.

Any Unicode surrogate pair starting with a high surrogate in the range D800 to DBFF hex is converted to a four-byte UTF-8 character as follows:

Unicode Characters	1 1 0 1 1 0 a b	c d e f g h i j	1 1 0 1 1 1 k l	m n o p q r s t
UTF-8 Character	1 1 1 1 0 u v w	1 0 x y e f g h	1 0 i j k l m n	1 0 o p q r s t
where uvwxy = abcd + 1				

When the ETF3-enhancement facility is not installed, or when the W bit of the M_3 field is zero, the first six bits of an expected Unicode low surrogate are ignored. When the ETF3-enhancement facility is installed, and the W bit is one, the first six bits of the Unicode low surrogate must contain 110111 binary; otherwise, the Unicode low surrogate is invalid, and condition code 2 is set.

The second-operand location is considered exhausted when it does not contain at least two remaining bytes or at least four remaining bytes when the first two bytes are a Unicode high surrogate. The first-operand location is considered exhausted when it does not contain at least the one, two, three, or four remaining bytes required to contain the UTF-8 character resulting from the conversion of the next second-operand character or surrogate pair.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been converted, condition code 3 is set.

When the conditions for setting condition codes 1 and 2 are both met, condition code 2 is set.

When the operation is completed, the contents of general register $R_2 + 1$ are decremented by the number of bytes converted, and the contents of general register R_2 are incremented by the same number. Also, the contents of general register $R_1 + 1$ are decremented by the number of bytes placed at the first-operand location, and the contents of general register R_1 are incremented by the same number. When general registers R_1 and R_2 are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_2, and $R_2 + 1$, always remain unchanged.

When condition code 2 is set, the following applies:

- When a Unicode high surrogate immediately precedes an invalid low surrogate, general register R_2 contains the address of the Unicode high surrogate.
- When an isolated Unicode low surrogate is detected, general register R_2 contains the address of the isolated low surrogate.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

When the R_1 register is the same register as the R_2 register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

Resulting Condition Code:

0 Entire second operand processed
1 End of first operand reached
2 Invalid Unicode low surrogate
3 CPU-determined number of characters converted

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)

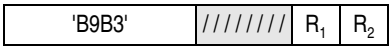
- Specification
- Transaction constraint

Programming Notes:

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.
2. The storage-operand references of CONVERT UNICODE TO UTF-8 may be multiple-access references. (See "Storage-Operand Consistency" on page 5-125.)
3. The CONVERT UTF-16 TO UTF-8 (CONVERT UNICODE TO UTF-8) instruction supports UTF-16 characters only in the big-endian encoding.
4. As observed by this CPU, other CPUs, and the I/O subsystem, inconsistent results may be briefly stored in the first-operand location. See the section "Effects of CPU Retry" on page 11-3 for further details.
5. Determining that isolated Unicode low surrogates are invalid is a model-dependent behavior that is not available on the IBM z13 and earlier models.

CONVERT UTF-32 TO UTF-16

CU42 R_1,R_2 [RRE]



The four-byte UTF-32 characters of the second operand are converted to two-byte UTF-16 characters and placed at the first-operand location. The operation proceeds until the end of the first or second operand is reached, a CPU-determined number of characters have been converted, or an invalid UTF-32 character is encountered, whichever occurs first. The result is indicated in the condition code.

The R_1 and R_2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and the second operand is designated by the contents of general registers R_1 and R_2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the entire contents of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-295.

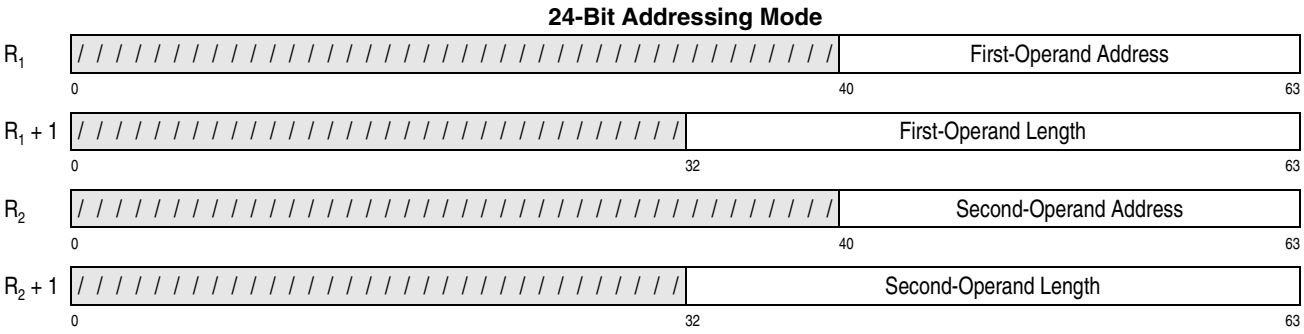


Figure 7-295. Register Contents for CONVERT UTF-32 TO UTF-16 (Part 1 of 2)

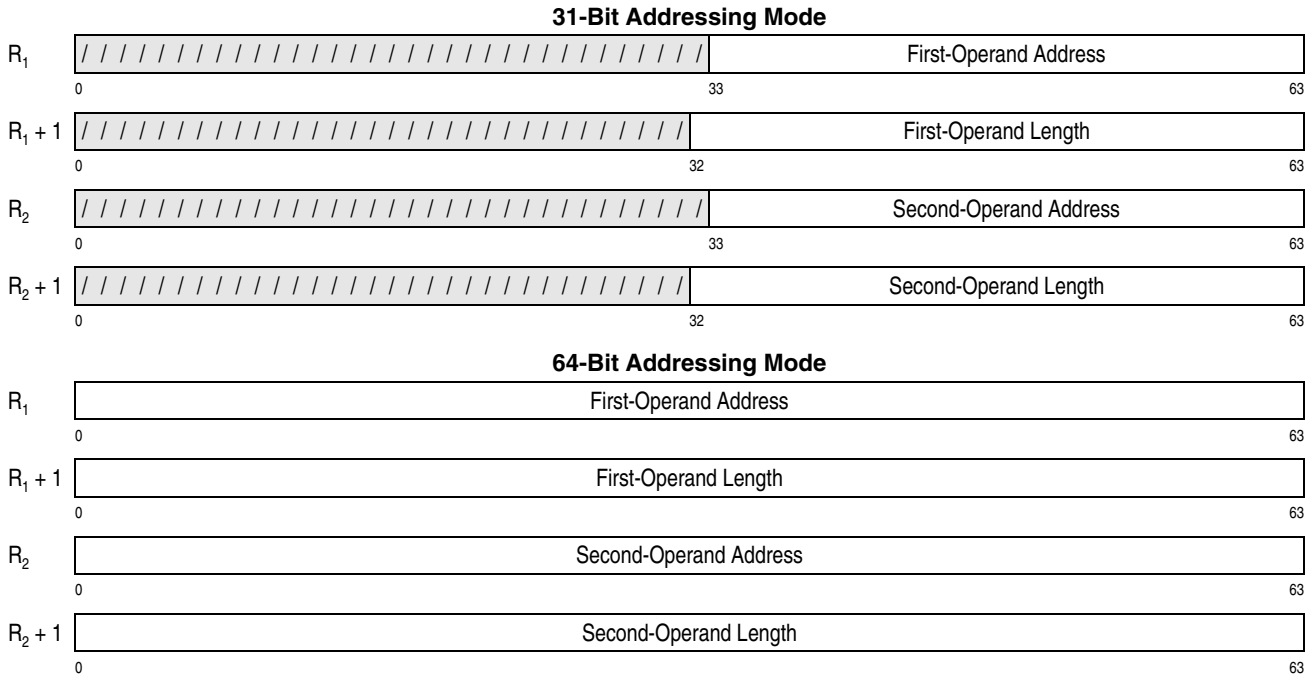


Figure 7-295. Register Contents for CONVERT UTF-32 TO UTF-16 (Part 2 of 2)

The characters of the second operand are selected one by one for conversion, proceeding left to right. The bytes resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted, a CPU-determined number of second-operand characters have been converted, or an invalid UTF-32 character is encountered in the second operand.

To show the method of converting a UTF-32 character to a UTF-16 character, the bits of a UTF-32 character in the range 00000000 to 0000D7FF or 0000DC00 to 0000FFFF hex are identified by letters as follows:

UTF-32 Character Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0

UTF-32 Character Bit Numbers	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Identifying Bit Letters	a b c d e f g h	i j k l m n o p

surrogate followed by a character called a low surrogate.

The bits of a UTF-32 character in the range 00010000 to 0010FFFF hex are identified by letters as follows:

UTF-32 Character Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	0 0 0 0 0 0 0 0	0 0 0 u v w x y

UTF-32 Character Bit Numbers	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Identifying Bit Letters	e f g h i j k l	m n o p q r s t

The bits of a UTF-16 character in the range 0000 to D7FF or DC00 to FFFF hex are identified by letters as follows:

UTF-16 Character Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	a b c d e f g h	i j k l m n o p

In the case of a UTF-16 surrogate pair, which is a character pair consisting of a character called a high

gate, the bits are identified by letters as follows:

UTF-16 High Surrogate Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	1 1 0 1 1 0 a b	c d e f g h i j

UTF-16 Low Surrogate Bit Numbers	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Identifying Bit Letters	1 1 0 1 1 1 k l	m n o p q r s t

When the contents of the UTF-32 characters are in the range 00000000 to 0000D7FF or 0000E000 to 0000FFFF hex, the character is converted to a two-byte UTF-16 character as follows:

UTF-32 Character	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	a b c d e f g h	i j k l m n o p
UTF-16 Character	a b c d e f g h	i j k l m n o p		

On some models, when the contents of the UTF-32 character are in the range of 0000DC00 to 0000DFFF hex, the character is also converted to a two-byte UTF-16 character as described above.

When the contents of a UTF-32 character are in the range 00010000 to 0010FFFF hex, the character is converted to two two-byte UTF-16 characters (a surrogate pair) as follows:

UTF-32 Character	0 0 0 0 0 0 0 0	0 0 0 u v w x y	e f g h i j k l	m n o p q r s t
UTF-16 Characters	1 1 0 1 1 0 a b	c d e f g h i j	1 1 0 1 1 1 k l	m n o p q r s t
where zabcd = uvwxy - 1				

The high order bit (z) produced by the subtract operation is necessarily zero and is ignored.

The second-operand location is considered exhausted when it does not contain at least four remaining bytes. The first-operand location is considered exhausted when it does not contain at least two remaining bytes or at least four remaining bytes in the case when a UTF-32 character in the range 00010000 to 0010FFFF hex is converted.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been processed, condition code 3 is set.

When the contents of the next UTF-32 character are in the range 0000D800 to 0000DBFF or 00110000 to FFFFFFFF hex, the character is invalid, and condition code 2 is set. On some models, when the contents of the next UTF-32 character are in the range of 0000DC00 to 0000DFFF hex, the character is invalid, and condition code 2 is set.

When the conditions for setting condition codes 1 and 2 are both met, condition code 2 is set.

When the operation is completed, the contents of general register $R_2 + 1$ are decremented by the number of bytes converted, and the contents of general register R_2 are incremented by the same number. Also, the contents of general register $R_1 + 1$ are decremented by the number of bytes placed at the first-operand location, and the contents of general register R_1 are incremented by the same number. When general registers R_1 and R_2 are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_2, and $R_2 + 1$, always remain unchanged.

When condition code 2 is set, general register R_2 contains the address of the invalid UTF-32 character.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

When the R_1 register is the same register as the R_2 register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes,

access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

Resulting Condition Code:

0 Entire second operand processed
1 End of first operand reached
2 Invalid UTF-32 character
3 CPU-determined number of characters processed

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 3 is not installed)
- Specification
- Transaction constraint

Programming Notes:

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.

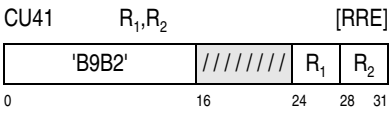
2. The storage-operand references of CONVERT UTF-32 TO UTF-16 may be multiple-access references. (See “Storage-Operand Consistency” on page 5-125.)

3. The CONVERT UTF-32 TO UTF-16 instruction supports UTF-16 and UTF-32 characters only in the big-endian encoding.

4. As observed by this CPU, other CPUs, and the I/O subsystem, inconsistent results may be briefly stored in the first-operand location. See the section “Effects of CPU Retry” on page 11-3 for further details.

5. Determining that UTF-32 characters in the range of 0000DC00 to 0000DFFF are invalid is a model-dependent behavior that is not available on the IBM z13 and earlier models.

CONVERT UTF-32 TO UTF-8



The four-byte UTF-32 characters of the second operand are converted to UTF-8 characters and placed at the first-operand location. The UTF-8 characters are one, two, three, or four bytes, depending on the Unicode characters that are converted. The operation proceeds until the end of the first or second operand is reached or a CPU-determined number of characters have been converted, whichever occurs first. The result is indicated in the condition code.

The R_1 and R_2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and the second operand is designated by the contents of general registers R_1 and R_2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the entire contents of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-296.

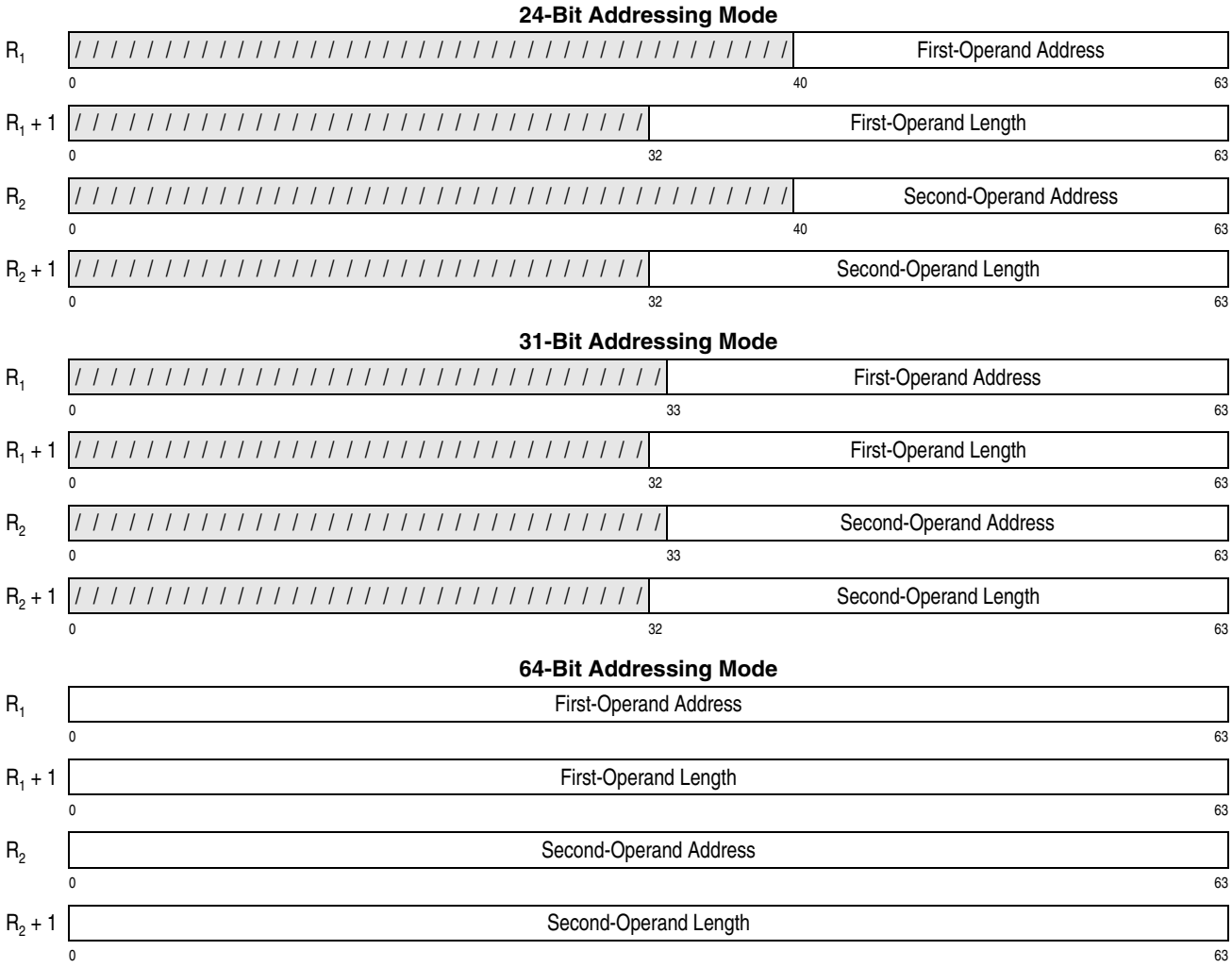


Figure 7-296. Register Contents for CONVERT UTF-32 TO UTF-8

The characters of the second operand are selected one by one for conversion, proceeding left to right. The bytes resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted, a CPU-determined number of second-operand characters have been converted, or on some models, until an invalid Unicode character is detected.

To show the method of converting a UTF-32 character to a UTF-8 character, the bits of a UTF-32 character in the range 00000000 to 0000D7FF or

The bits of a UTF-32 character in the range 00010000 to 0010FFFF hex (UTF-16 surrogate pair) 0000DC00 to 0000FFFF hex are identified by letters as follows:

UTF-32 Character Bit Numbers	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Identifying Bit Letters	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

UTF-32 Character Bit Numbers	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Identifying Bit Letters	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p

are identified by letters as follows:

UTF-32 Character Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	0 0 0 0 0 0 0 0	0 0 0 u v w x y

UTF-32 Character Bit Numbers	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Identifying Bit Letters	e f g h i j k l	m n o p q r s t

Any UTF-32 character in the range 00000000 to 0000007F hex is converted to a one-byte UTF-8 character as follows:

UTF-32 Character	00000000	00000000	00000000	0jklmnop
UTF-8 Character	0jklmnop			

Any UTF-32 character in the range 00000080 to 000007FF hex is converted to a two-byte UTF-8 character as follows:

UTF-32 Character	00000000	00000000	00000fgh	ijklmnop
UTF-8 Character	110fghij	10klmnop		

Any UTF-32 character in the range 00000800 to 0000D7FF or 0000E000 to 0000FFFF hex is converted to a three-byte UTF-8 character as follows:

UTF-32 Character	00000000	00000000	abcdefgh	ijklmnop
UTF-8 Character	1110abcd	10efghij	10klmnop	

On some models, any UTF-32 character the range of 0000DC00 to 0000DFFF hex is converted to a three-byte UTF-8 character as described above.

Any UTF-32 character in the range 00010000 to 0010FFFF hex is converted to a four-byte UTF-8 character as follows:

UTF-32 Character	00000000	000uvwxy	efghijkl	mnopqrst
UTF-8 Character	11110uvw	10xyefgh	10ijklmn	10opqrst

The second-operand location is considered exhausted when it does not contain at least four remaining bytes. The first-operand location is considered exhausted when it does not contain at least the one, two, three, or four remaining bytes required to contain the UTF-8 character resulting from the conversion of the next second-operand character or surrogate pair.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been converted, condition code 3 is set.

When the contents of the next UTF-32 character are in the range 0000D800 to 0000DBFF or 00110000 to FFFFFFFF hex, the character is invalid, and condition code 2 is set. On some models, when the contents of the next UTF-32 character are in the range of 0000DC00 to 0000DFFF hex, the character is invalid, and condition code 2 is set.

When the conditions for setting condition codes 1 and 2 are both met, condition code 2 is set.

When the operation is completed, the contents of general register $R_2 + 1$ are decremented by the number of bytes converted, and the contents of general register R_2 are incremented by the same number. Also, the contents of general register $R_1 + 1$ are decremented by the number of bytes placed at the first-operand location, and the contents of general register R_1 are incremented by the same number. When general registers R_1 and R_2 are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_2, and $R_2 + 1$, always remain unchanged.

When condition code 2 is set, general register R_2 contains the address of the invalid UTF-32 character.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may

be a different amount each time the instruction is executed.

When the R_1 register is the same register as the R_2 register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

Resulting Condition Code:

0 Entire second operand processed
1 End of first operand reached
2 Invalid UTF-32 character.
3 CPU-determined number of characters converted

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 3 is not installed)
- Specification
- Transaction constraint

Programming Notes:

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.

2. The storage-operand references of CONVERT UTF-32 TO UTF-8 may be multiple-access references. (See “Storage-Operand Consistency” on page 5-125.)

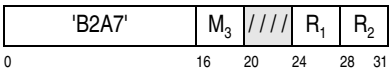
3. The CONVERT UTF-32 TO UTF-8 instruction supports UTF-32 characters only in the big-endian encoding.

4. As observed by this CPU, other CPUs, and the I/O subsystem, inconsistent results may be briefly stored in the first-operand location. See the section “Effects of CPU Retry” on page 11-3 for further details.

5. Determining that UTF-32 characters in the range of 0000DC00 to 0000DFFF are invalid is a model-dependent behavior that is not available on the IBM z13 and earlier models.

CONVERT UTF-8 TO UTF-16
CONVERT UTF-8 TO UNICODE

CU12 $R_1,R_2[,M_3]$ [RRF-c]
CUTFU $R_1,R_2[,M_3]$ [RRF-c]



The one-, two-, three-, or four-byte UTF-8 characters of the second operand are converted to two-byte Unicode characters and placed at the first-operand location. The operation proceeds until the end of the first or second operand is reached, a CPU-determined number of characters have been converted, or an invalid UTF-8 character is encountered, whichever occurs first. The result is indicated in the condition code.

The R_1 and R_2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and the second operand is designated by the contents of general registers R_1 and R_2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the entire contents of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 consti-

tute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-297.

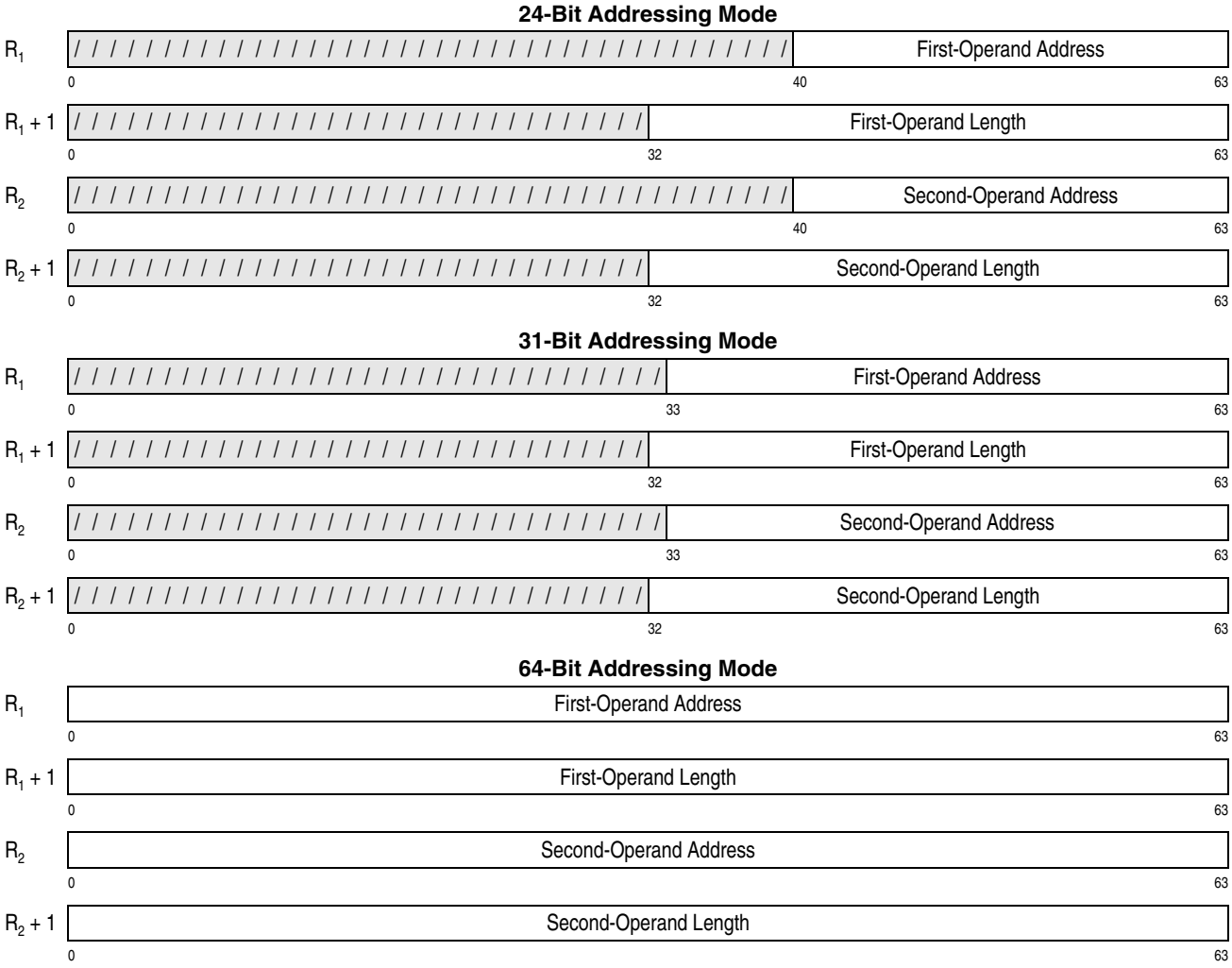
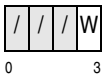


Figure 7-297. Register Contents for CONVERT UTF-8 TO UTF-16

When the ETF3-enhancement facility is installed, the M_3 field has the following format:



The bits of the M_3 field are defined as follows:

- ***Unassigned:*** Bits 0-2 are unassigned and should contain zeros; otherwise, the program may not operate compatibly in the future.
- ***Enhanced Well-Formedness-Checking (W):*** The W bit, bit 3 of the M_3 field, controls whether enhanced well-formedness checking of the UTF-8 characters is performed. When the W bit is zero, enhanced checking is not performed. When the W bit is one, enhanced checking is performed, as described below.

When the ETF3-enhancement facility is not installed, the M_3 field is ignored. When the ETF3-enhancement facility is installed in the z/Architecture architectural mode, it is unpredictable whether the M_3 field is ignored in the ESA/390-compatibility mode.

The characters of the second operand are selected one by one for conversion, proceeding left to right. The bytes resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted, a CPU-determined number of second-operand characters have been converted, or an invalid UTF-8 character is encountered in the second operand.

To show the method of converting a UTF-8 character to a Unicode character, the bits of a Unicode character are identified by letters as follows:

Unicode Character Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	a b c d e f g h	i j k l m n o p

In the case of a Unicode surrogate pair, which is a character pair consisting of a character called a high surrogate followed by a character called a low surrogate, the bits are identified by letters as follows:

Unicode High Surrogate Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	1 1 0 1 1 0 a b	c d e f g h i j

Unicode Low Surrogate Bit Numbers	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Identifying Bit Letters	1 1 0 1 1 1 k l	m n o p q r s t

Conversion of a UTF-8 character to a Unicode character is as follows:

1. When the contents of the first byte of a UTF-8 character are in the range 00 to 7F hex, the character is a one-byte character, and it is converted to a two-byte Unicode character as follows:

UTF-8 Character	0 j k l m n o p	
Unicode Character	0 0 0 0 0 0 0 0	0 j k l m n o p

2. When the contents of the first byte of the UTF-8 character are in the range 80 to BF hex, the character is invalid. When the ETF3-enhancement facility is installed, the W bit of the M_3 field is one, and the contents of the first byte of the UTF-8 character are in the range C0 to C1 hex, the character is also invalid.

3. When the ETF3-enhancement facility is not installed or the W bit of the M_3 field is zero, and the contents of the first byte of the UTF-8 character are in the range of C0 to DF hex; or when the ETF3-enhancement facility is installed, the W bit is one, and the contents of the first byte of the UTF-8 character are in the range of C2 to DF hex; the character is a two-byte character, and it is converted to a two-byte Unicode character as follows:

UTF-8 Character	1 1 0 f g h i j	1 0 k l m n o p
Unicode Character	0 0 0 0 0 f g h	i j k l m n o p

   When the ETF3-enhancement facility is not installed or when the W bit of the M_3 field is zero, the first two bits in the second byte of the UTF-8 character are ignored. When the ETF3-enhancement facility is installed and the W bit of the M_3 field is one, the contents of the second byte of the UTF-8 character must be in the range 80 to BF; otherwise the character is invalid.

4. When the contents of the first byte of a UTF-8 character are in the range E0 to EF hex, the character is a three-byte character, and it is converted to a two-byte Unicode character as follows:

UTF-8 Character	1 1 1 0 a b c d	1 0 e f g h i j	1 0 k l m n o p
Unicode Character	a b c d e f g h	i j k l m n o p	

   When the ETF3-enhancement facility is not installed or when the W bit of the M_3 field is zero, the first two bits in the second and third bytes of the UTF-8 character are ignored.

   When the ETF3-enhancement facility is installed and the W bit of the M_3 field is one, the contents of the second and third bytes of the UTF-8 character must be as follows:

   - When the first byte is E0 hex, the second and third bytes must be in the ranges A0 to BF and 80 to BF, respectively.
   - When the first byte is in the range E1 to EC hex or EE to EF, the second and third bytes must both be in the range 80 to BF hex.
   - When the first byte is ED hex, the second and third bytes must be in the ranges 80 to 9F and 80 to BF, respectively.

   Otherwise, the character is invalid.

5. When the ETF3-enhancement facility is not installed or the W bit of the M_3 field is zero, and the contents of the first byte of the UTF-8 character are in the range of F0 to F7 hex; or when the ETF3-enhancement facility is installed, the W bit is one, and the contents of the first byte of the UTF-8 character are in the range of F0 to F4 hex; the character is a four-byte character, and it is converted to two two-byte Unicode characters (a surrogate pair) as follows:

UTF-8 Character	1 1 1 1 0 u v w	1 0 x y e f g h	1 0 i j k l m n	1 0 o p q r s t
Unicode Characters	1 1 0 1 1 0 a b	c d e f g h i j	1 1 0 1 1 1 k l	m n o p q r s t
where zabcd = uvwxy - 1				

   When the ETF3-enhancement facility is not installed or when the W bit of the M_3 field is zero, the first two bits in the second, third, and fourth bytes of the UTF-8 character are ignored, and the high order bit (z) produced by the subtract operation should be zero but is ignored.

   When the ETF3-enhancement facility is installed and the W bit of the M_3 field is one, the contents of the second, third, and fourth bytes of the UTF-8 character must be as follows:

   - When the first byte is F0 hex, the second, third, and fourth bytes must be in the ranges 90 to BF, 80 to BF, and 80 to BF, respectively.
   - When the first byte is in the range F1 to F3, the second, third, and fourth bytes must all be in the range 80 to BF hex.
   - When the first byte is F4 hex, the second, third, and fourth bytes must be in the ranges 80 to 8F, 80 to BF, and 80 to BF, respectively.

   Otherwise, the character is invalid.

6. When the ETF3-enhancement facility is installed, the W bit of the M_3 field is one, and the contents of the first byte of the UTF-8 character are in the range of F5 to F7 hex, the character is invalid.
7. When the contents of the first byte of the UTF-8 character are in the range of F8-FF, the character is invalid.

If an invalid character is encountered, condition code 2 is set, and general register R_2 contains the address of the first byte of the invalid UTF-8 character.

The second-operand location is considered exhausted when it does not contain at least one remaining byte or when it does not contain at least the two, three, or four remaining bytes required to contain the two-, three-, or four-byte UTF-8 character indicated by the contents of the first remaining byte. The first-operand location is considered exhausted when it does not contain at least two remaining bytes or at least four remaining bytes in the case when a four byte UTF-8 character is to be converted.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been processed, condition code 3 is set.

When the conditions for setting condition codes 1 and 2 are both met, condition code 2 is set.

When the operation is completed, the contents of general register $R_2 + 1$ are decremented by the number of bytes converted, and the contents of general register R_2 are incremented by the same number. Also, the contents of general register $R_1 + 1$ are decremented by the number of bytes placed at the first-operand location, and the contents of general register R_1 are incremented by the same number. When general registers R_1 and R_2 are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_2, and $R_2 + 1$, always remain unchanged.

When condition code 2 is set, general register R_2 contains the address of the invalid UTF-8 character.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

When the R_1 register is the same register as the R_2 register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

Resulting Condition Code:

0 Entire second operand processed
1 End of first operand reached
2 Invalid UTF-8 character
3 CPU-determined number of characters processed

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Specification
- Transaction constraint

Programming Notes:

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.

2. When the ETF3-enhancement facility is not installed, or when the W bit of the M_3 operand is zero, bits 0 and 1 of the continuation bytes of multiple-byte UTF-8 characters are not checked in order to improve the performance of the conversion. Therefore, invalid continuation bytes are not detected.

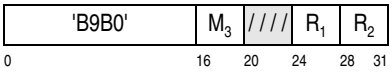
3. The storage-operand references of CONVERT UTF-8 TO UTF-16 may be multiple-access references. (See “Storage-Operand Consistency” on page 5-125.)

4. The CONVERT UTF-8 TO UTF-16 (CONVERT UTF-8 TO UNICODE) instruction supports UTF-16 characters only in the big-endian encoding.

5. As observed by this CPU, other CPUs, and the I/O subsystem, inconsistent results may be briefly stored in the first-operand location. See the section “Effects of CPU Retry” on page 11-3 for further details.

CONVERT UTF-8 TO UTF-32

CU14 $R_1,R_2[,M_3]$ [RRF-c]



The one-, two-, three-, or four-byte UTF-8 characters of the second operand are converted to four-byte UTF-32 characters and placed at the first-operand location. The operation proceeds until the end of the first or second operand is reached, a CPU-determined number of characters have been converted, or an invalid UTF-8 character is encountered, whichever occurs first. The result is indicated in the condition code.

The R_1 and R_2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and the second operand is designated by the contents of general registers R_1 and R_2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the entire contents of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions

0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-298.

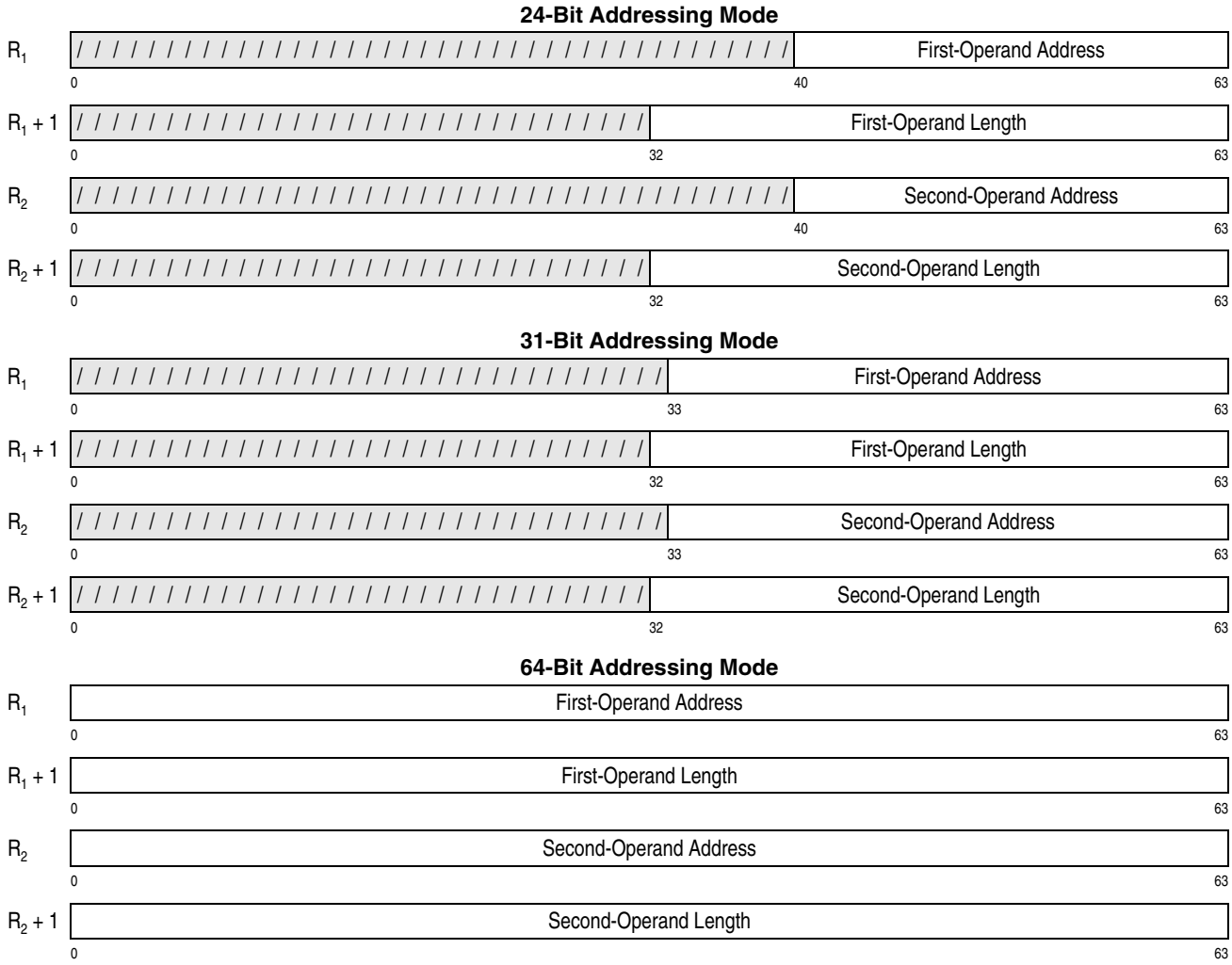
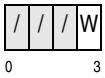


Figure 7-298. Register Contents for CONVERT UTF-8 TO UTF32

When the ETF3-enhancement facility is installed, the M_3 field has the following format:



The bits of the M_3 field are defined as follows:

- ***Unassigned:*** Bits 0-2 are unassigned and should contain zeros; otherwise, the program may not operate compatibly in the future.
- ***Enhanced Well-Formedness-Checking (W):*** The W bit, bit 3 of the M_3 field, controls whether enhanced well-formedness checking of the UTF-8 characters is performed. When the W bit is zero, enhanced checking is not performed. When the W bit is one, enhanced checking is performed, as described below.

When the ETF3-enhancement facility is not installed, the M_3 field is ignored.

The characters of the second operand are selected one by one for conversion, proceeding left to right. The bytes resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted, a CPU-determined number of second-operand characters have

been converted, or an invalid UTF-8 character is encountered in the second operand.

To show the method of converting a UTF-8 character to a UTF-32 character, the bits of a UTF-32 character in the range 00000000 to 0000D7FF or 0000DC00 to 0000FFFF hex are identified by letters as follows:

UTF-32 Character Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0

UTF-32 Character Bit Numbers	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Identifying Bit Letters	a b c d e f g h	i j k l m n o p

The bits of a UTF-32 character in the range 00010000 0010FFFF hex (UTF-16 surrogate pair) are identified by letters as follows:

UTF-32 Character Bit Numbers	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
Identifying Bit Letters	0 0 0 0 0 0 0 0	0 0 0 u v w x y

UTF-32 Character Bit Numbers	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Identifying Bit Letters	e f g h i j k l	m n o p q r s t

Conversion of a UTF-8 character to a UTF-32 character is as follows:

1. When the contents of the first byte of a UTF-8 character are in the range 00 to 7F hex, the character is a one-byte character, and it is converted to a four-byte UTF-32 character as follows:

UTF-8 Character	0 j k l m n o p			
UTF-32 Character	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 j k l m n o p

2. When the contents of the first byte of the UTF-8 character are in the range 80 to BF hex, the character is invalid. When the ETF3-enhancement facility is installed, the W bit of the M_3 field is one, and the contents of the first byte of the UTF-8 character are in the range C0 to C1 hex, the character is also invalid.

3. When the ETF3-enhancement facility is not installed or the W bit of the M_3 field is zero, and the contents of the first byte of the UTF-8 character are in the range of C0 to DF hex; or when the ETF3-enhancement facility is installed, the W bit is one, and the contents of the first byte of the UTF-8 character are in the range of C2 to DF hex; the character is a two-byte character, and it is converted to a four-byte UTF-32 character as follows:

UTF-8 Character	1 1 0 f g h i j	1 0 k l m n o p		
UTF-32 Character	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 f g h	i j k l m n o p

When the ETF3-enhancement facility is not installed or when the W bit of the M_3 field is zero, the first two bits in the second byte of the UTF-8 character are ignored. When the ETF3-enhancement facility is installed and the W bit of the M_3 field is one, the contents of the second byte of the UTF-8 character must be in the range 80 to BF; otherwise the character is invalid.

4. When the contents of the first byte of a UTF-8 character are in the range E0 to EF hex, the character is a three-byte character, and it is converted to a four-byte UTF-32 character as follows:

UTF-8 Character	1 1 1 0 a b c d	1 0 e f g h i j	1 0 k l m n o p	
UTF-32 Character	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	a b c d e f g h	i j k l m n o p

When the ETF3-enhancement facility is not installed or when the W bit of the M_3 field is zero, the first two bits in the second and third bytes of the UTF-8 character are ignored.

When the ETF3-enhancement facility is installed and the W bit of the M_3 field is one, the contents of the second and third bytes of the UTF-8 character must be as follows:

- When the first byte is E0 hex, the second and third bytes must be in the ranges A0 to BF and 80 to BF, respectively.
- When the first byte is in the range E1 to EC hex or EE to EF, the second and third bytes must both be in the range 80 to BF hex.
- When the first byte is ED hex, the second and third bytes must be in the ranges 80 to 9F and 80 to BF, respectively.

Otherwise, the character is invalid.

5. When the ETF3-enhancement facility is not installed or the W bit of the M_3 field is zero, and the contents of the first byte of the UTF-8 character are in the range of F0 to F7 hex; or when the ETF3-enhancement facility is installed, the W bit is one, and the contents of the first byte of the UTF-8 character are in the range of F0 to F4 hex; the character is a four-byte character, and it is converted to a four-byte UTF-32 character (a surrogate pair) as follows:

UTF-8 Character	11110uvw	10xyefgh	10ijklmn	10opqrst
UTF-32 Character	00000000	000uvwxy	efghijkl	mnopqrst

When the ETF3-enhancement facility is not installed or when the W bit of the M_3 field is zero, the first two bits in the second, third, and fourth bytes of the UTF-8 character are ignored.

When the ETF3-enhancement facility is installed and the W bit of the M_3 field is one, the contents of the second, third, and fourth bytes of the UTF-8 character must be as follows:

- When the first byte is F0 hex, the second, third, and fourth bytes must be in the ranges 90 to BF, 80 to BF, and 80 to BF, respectively.
- When the first byte is in the range F1 to F3, the second, third, and fourth bytes must all be in the range 80 to BF hex.
- When the first byte is F4 hex, the second, third, and fourth bytes must be in the ranges 80 to 8F, 80 to BF, and 80 to BF, respectively.

Otherwise, the character is invalid.

6. When the ETF3-enhancement facility is installed, the W bit of the M_3 field is one, and the contents of the first byte of the UTF-8 character are in the range of F5 to F7 hex, the character is invalid.
7. When the contents of the first byte of the UTF-8 character are in the range of F8-FF, the character is invalid.

If an invalid character is encountered, condition code 2 is set, and general register R_2 contains the address of the first byte of the invalid UTF-8 character.

The second-operand location is considered exhausted when it does not contain at least one remaining byte or when it does not contain at least the two, three, or four remaining bytes required to contain the two-, three-, or four-byte UTF-8 character indicated by the contents of the first remaining byte. The first-operand location is considered exhausted when it does not contain at least four remaining bytes.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been processed, condition code 3 is set.

When the conditions for setting condition codes 1 and 2 are both met, condition code 2 is set.

When the operation is completed, the contents of general register $R_2 + 1$ are decremented by the number of bytes converted, and the contents of general register R_2 are incremented by the same number. Also, the contents of general register $R_1 + 1$ are decremented by the number of bytes placed at the first-operand location, and the contents of general register R_1 are incremented by the same number. When general registers R_1 and R_2 are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_2, and $R_2 + 1$, always remain unchanged.

When condition code 2 is set, general register R_2 contains the address of the invalid UTF-8 character.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

When the R_1 register is the same register as the R_2 register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

Resulting Condition Code:

0 Entire second operand processed
1 End of first operand reached
2 Invalid UTF-8 character
3 CPU-determined number of characters processed

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 3 is not installed)
- Specification
- Transaction constraint

Programming Notes:

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.

2. When the ETF3-enhancement facility is not installed, or when the W bit of the M_3 operand is zero, bits 0 and 1 of the continuation bytes of multiple-byte UTF-8 characters are not checked in order to improve the performance of the conversion. Therefore, invalid continuation bytes are not detected.

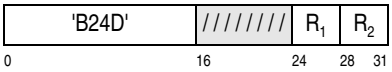
3. The storage-operand references of CONVERT UTF-8 TO UTF-32 may be multiple-access references. (See “Storage-Operand Consistency” on page 5-125.)

4. The CONVERT UTF-8 TO UTF-32 instruction supports UTF-32 characters only in the big-endian encoding.

5. As observed by this CPU, other CPUs, and the I/O subsystem, inconsistent results may be briefly stored in the first-operand location. See the section “Effects of CPU Retry” on page 11-3 for further details.

COPY ACCESS

CPYA R_1,R_2 [RRE]



The contents of access register R_2 are placed in access register R_1.

Condition Code: The code remains unchanged.

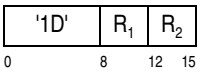
Program Exceptions:

- Transaction constraint

DIVIDE

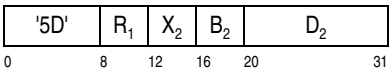
Register-and-register format:

DR R_1,R_2 [RR]



Register-and-storage format:

D $R_1,D_2(X_2,B_2)$ [RX-a]



The 64-bit first operand (the dividend) is divided by the 32-bit second operand (the divisor), and the 32-bit remainder and quotient are placed at the first-operand location.

The R_1 field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The dividend is treated as a 64-bit signed binary integer. The leftmost 32 bits of the dividend are in bit positions 32-63 of general register R_1, and the rightmost 32 bits are in bit positions 32-63 of general register $R_1 + 1$.

The divisor, remainder, and quotient are treated as 32-bit signed binary integers. For DIVIDE (DR), the divisor is in bit positions 32-63 of general register R_2. The remainder is placed in bit positions 32-63 of general register R_1, and the quotient is placed in bit positions 32-63 of general register $R_1 + 1$. Bits 0-31 of the registers remain unchanged.

The sign of the quotient is determined by the rules of algebra, and the remainder has the same sign as the dividend, except that a zero quotient or a zero remainder is always positive.

When the divisor is zero, or when the magnitudes of the dividend and divisor are such that the quotient cannot be expressed by a 32-bit signed binary integer, a fixed-point-divide exception is recognized. This includes the case of division of zero by zero.

Condition Code: The code remains unchanged.

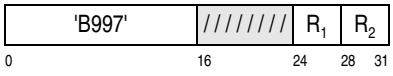
Program Exceptions:

- Access (fetch, operand 2 of D only)
- Fixed-point divide
- Specification
- Transaction constraint

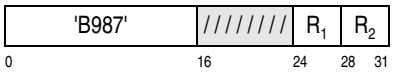
DIVIDE LOGICAL

Register-and-register formats:

DLR R_1,R_2 [RRE]

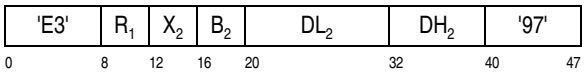


DLGR R_1,R_2 [RRE]

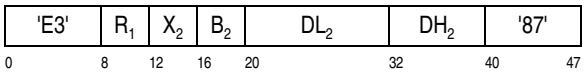


Register-and-storage formats:

DL $R_1,D_2(X_2,B_2)$ [RXY-a]



DLG $R_1,D_2(X_2,B_2)$ [RXY-a]



The 64-bit or 128-bit first operand (the dividend) is divided by the 32-bit or 64-bit second operand (the divisor), and the 32-bit or 64-bit remainder and quotient are placed at the first-operand location.

The R_1 field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

For DIVIDE LOGICAL (DLR, DL), the dividend is treated as a 64-bit unsigned binary integer. The leftmost 32 bits of the dividend are in bit positions 32-63 of general register R_1, and the rightmost 32 bits are in bit positions 32-63 of general register $R_1 + 1$.

The divisor, remainder, and quotient are treated as 32-bit unsigned binary integers. For DIVIDE LOGICAL (DLR), the divisor is in bit positions 32-63 of general register R_2. The remainder is placed in bit positions 32-63 of general register R_1, and the quotient is placed in bit positions 32-63 of general register $R_1 + 1$. Bits 0-31 of the registers remain unchanged.

For DIVIDE LOGICAL (DLGR, DLG), the dividend is treated as a 128-bit unsigned binary integer. The leftmost 64 bits of the dividend are in general register R_1, and the rightmost 64 bits are in general register $R_1 + 1$. The divisor, remainder, and quotient are treated as 64-bit unsigned binary integers. The remainder is placed in general register R_1, and the quotient is placed in general register $R_1 + 1$.

When the divisor is zero, or when the magnitudes of the dividend and divisor are such that the quotient cannot be expressed as a 32-bit unsigned binary integer for DIVIDE LOGICAL (DLR, DL), or a 64-bit unsigned binary integer for DIVIDE LOGICAL (DLGR, DLG), a fixed-point-divide exception is recognized. This includes the case of division of zero by zero.

Condition Code: The code remains unchanged.

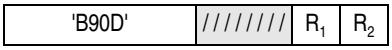
Program Exceptions:

- Access (fetch, operand 2 of DL or DLG only)
- Fixed-point divide
- Specification
- Transaction constraint

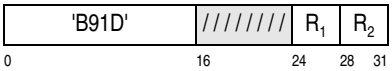
DIVIDE SINGLE

Register-and-register formats:

DSGR R_1,R_2 [RRE]

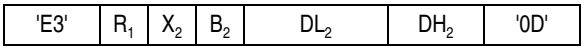


DSGFR R_1,R_2 [RRE]

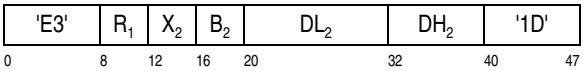


Register-and-storage formats:

DSG $R_1,D_2(X_2,B_2)$ [RXY-a]



DSGF $R_1,D_2(X_2,B_2)$ [RXY-a]



The 64-bit contents of general register $R_1 + 1$ (the dividend) are divided by the 64-bit or 32-bit second operand (the divisor), the 64-bit remainder is placed in general register R_1, and the 64-bit quotient is placed in general register $R_1 + 1$.

The R_1 field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The dividend, quotient, and remainder are treated as 64-bit signed binary integers. For DIVIDE SINGLE (DSGR, DSG), the divisor is treated as a 64-bit signed binary integer. For DIVIDE SINGLE (DSGFR, DSGF), the divisor is treated as a 32-bit signed binary integer. For DSGFR, the divisor is in bit positions 32-63 of general register R_2.

The sign of the quotient is determined by the rules of algebra, and the remainder has the same sign as the dividend, except that a zero quotient or a zero remainder is always positive.

When the divisor is zero, or when the magnitudes of the dividend and divisor are such that the quotient cannot be expressed by a 64-bit signed binary integer, a fixed-point-divide exception is recognized. This includes the case of division of zero by zero.

Condition Code: The code remains unchanged.

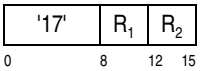
Program Exceptions:

- Access (fetch, operand 2 of DSG and DSGF only)
- Fixed-point divide
- Specification
- Transaction constraint

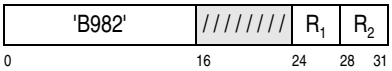
EXCLUSIVE OR

Register-and-register formats:

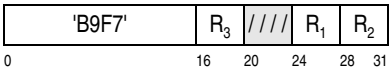
XR R_1,R_2 [RR]



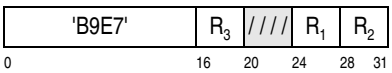
XGR R_1,R_2 [RRE]



XRK R_1,R_2,R_3 [RRF-a]

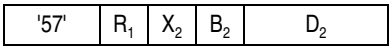


XGRK R_1,R_2,R_3 [RRF-a]

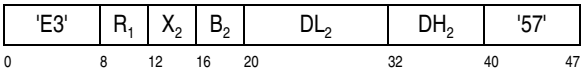


Register-and-storage formats:

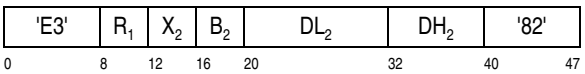
X $R_1,D_2(X_2,B_2)$ [RX-a]



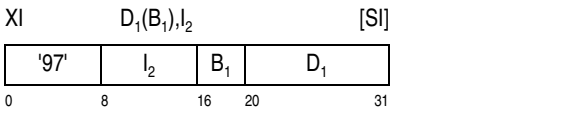
XY $R_1,D_2(X_2,B_2)$ [RXY-a]

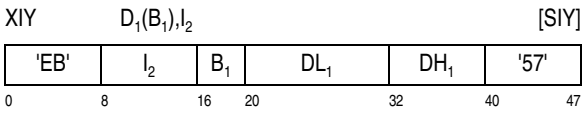


XG $R_1,D_2(X_2,B_2)$ [RXY-a]

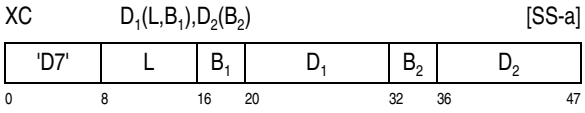


Storage-and-immediate formats:





Storage-and-storage format:



For X, XC, XG, XGR, XI, XIY, XR, and XY, the EXCLUSIVE OR of the first and second operands is placed at the first-operand location. For XGRK and XRK, the EXCLUSIVE OR of the second and third operands is placed at the first-operand location.

The connective EXCLUSIVE OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the bits in the corresponding bit positions in the two operands are unlike; otherwise, the result bit is set to zero.

For EXCLUSIVE OR (XC), each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes.

For EXCLUSIVE OR (XI, XIY), the first operand is one byte in length, and only one byte is stored. When the interlocked-access facility 2 is installed, the update of the first operand appears to be an interlocked-update reference as observed by other CPUs and channel programs, and a specific-operand-serialization operation is performed.

For EXCLUSIVE OR (X, XR, XRK, and XY), the operands are 32 bits, and for EXCLUSIVE OR (XG, XGR, and XGRK), they are 64 bits.

The displacements for X, XI, and both operands of XC are treated as 12-bit unsigned binary integers. The displacement for XY, XIY, and XG is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero
1 Result not zero
2 --
3 --

Program Exceptions:

- Access (fetch, operand 2, X, XY, XG, and XC; fetch and store, operand 1, XI, XIY, and XC)
- Operation (XY and XIY, if the long-displacement facility is not installed; XGRK and XRK, if the distinct-operands facility is not installed)
- Transaction constraint (XC)

Programming Notes:

1. An example of the use of the EXCLUSIVE OR instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. EXCLUSIVE OR may be used to invert a bit, an operation particularly useful in testing and setting programmed binary switches.

3. A field EXCLUSIVE-ORed with itself becomes all zeros.

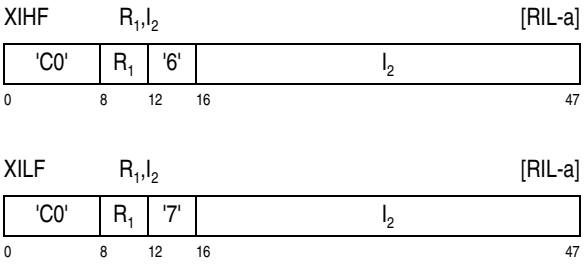
4. For EXCLUSIVE OR (XR or XGR), the sequence A EXCLUSIVE-OR B, B EXCLUSIVE-OR A, A EXCLUSIVE-OR B results in the exchange of the contents of A and B without the use of an additional general register.

5. Accesses to the first operand of EXCLUSIVE OR (XC) – and, when the interlocked-access facility 2 is not installed, accesses to the first operand of EXCLUSIVE OR (XI, XIY) – consist in fetching a first-operand byte from storage and subsequently storing the updated value. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, these instructions cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for OR (OI) in “Multiprogramming and Multiprocessing Examples” on page A-45.

   When the interlocked-access facility 2 is installed, EXCLUSIVE OR (XI, XIY) can be safely used to update a location in storage, even if the possibility exists that another CPU or a

channel program may also be updating the location.

EXCLUSIVE OR IMMEDIATE



The second operand is EXCLUSIVE ORed with bits of the first operand, and the result replaces those bits of the first operand. The remainder of the first operand remains unchanged.

For each instruction, the bits of the first operand that are EXCLUSIVE ORed with the second operand and then replaced are as follows:

Instruction	Bits Exclusive ORed and Replaced
XIHF	0-31
XILF	32-63

The connective EXCLUSIVE OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the bits in the corresponding bit positions in the two operands are unlike; otherwise, the result bit is set to zero.

The condition code is set based on the result of the 32-bit EXCLUSIVE OR operation.

Resulting Condition Code:

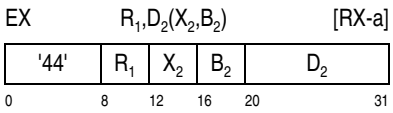
0 Result is zero
1 Result is not zero
2 --
3 --

Program Exceptions:

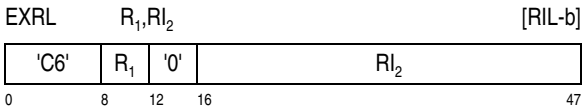
- Operation, if the extended-immediate facility is not installed)

Programming Note: The setting of the condition code is based only on the bits that are exclusive ORed and replaced.

EXECUTE



EXECUTE RELATIVE LONG



The single instruction at the second-operand address is modified by the contents of general register R_1, and the resulting instruction, called the target instruction, is executed.

When the R_1 field is not zero, bits 8-15 of the instruction designated by the second-operand address are ORed with bits 56-63 of general register R_1. The ORing does not change either the contents of general register R_1 or the instruction in storage, and it is effective only for the interpretation of the instruction to be executed. When the R_1 field is zero, no ORing takes place.

The target instruction may be two, four, or six bytes in length. The execution and exception handling of the target instruction are exactly as if the target instruction were obtained in normal sequential operation, except for the instruction address and the instruction-length code.

The instruction address in the current PSW is increased by the length of the execute-type instruction (EXECUTE or EXECUTE RELATIVE LONG). This updated address and the instruction-length code of the execute-type instruction are used, for example, as part of the link information when the target instruction is BRANCH AND LINK. When the target instruction is a successful branching instruction, the instruction address in the current PSW is replaced by the branch address specified by the target instruction.

When the target instruction is any of the following instructions, an execute exception is recognized.

- Another execute-type instruction
- TRANSACTION ABORT
- TRANSACTION BEGIN

- TRANSACTION END

The effective address of EXECUTE must be even; otherwise, a specification exception is recognized.

When the target instruction is two or three halfwords in length but can be executed without fetching its second or third halfword, it is unpredictable whether access exceptions are recognized for the unused halfwords. Access exceptions are not recognized for the second-operand address when the address is odd.

The second-operand address of an execute-type instruction is an instruction address rather than a logical address; thus, the target instruction is fetched from the primary address space when in the primary-space, secondary-space, or access-register mode.

For EXECUTE RELATIVE LONG, the contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the address of the target instruction in storage.

Condition Code: The code may be set by the target instruction.

Program Exceptions:

- Access (fetch, target instruction)
- Execute
- Operation (EXRL, when the execute-extensions facility is not installed)
- Specification (EX)
- Transaction constraint

Programming Notes:

1. An example of the use of the EXECUTE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. The ORing of eight bits from the general register with the designated instruction permits the indirect specification of the length, index, mask, immediate-data, register, or extended-op-code field.

3. The fetching of the target instruction is considered to be an instruction fetch for purposes of program-event recording and for purposes of reporting access exceptions.

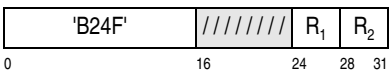
4. An access or specification exception may be caused by execute-type instructions or by the target instruction, except that EXECUTE RELATIVE LONG cannot cause a specification exception due to a misaligned target operand.

5. When an interruptible instruction is made the target of an execute-type instruction, the program normally should not designate any register updated by the interruptible instruction as the R_1, X_2, or B_2 register for EXECUTE or R_1 register for EXECUTE RELATIVE LONG. Otherwise, on resumption of execution after an interruption, or if the instruction is refetched without an interruption, the updated values of these registers will be used in the execution of the execute-type instruction. Similarly, the program should normally not let the destination field in storage of an interruptible instruction include the location of an execute-type instruction, since the new contents of the location may be interpreted when resuming execution.

6. Exception conditions that occur during the execution of EXECUTE or EXECUTE RELATIVE LONG are recognized before exception conditions for the target instruction.

EXTRACT ACCESS

EAR R_1,R_2 [RRE]



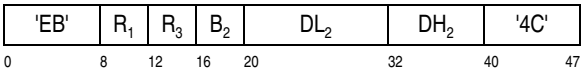
The contents of access register R_2 are placed in bit positions 32-63 of general register R_1. Bits 0-31 of general register R_1 remain unchanged.

Condition Code: The code remains unchanged.

Program Exceptions: None.

EXTRACT CPU ATTRIBUTE

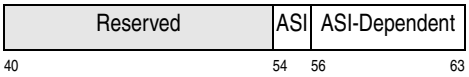
ECAG $R_1,R_3,D_2(B_2)$ [RSY-a]



Information regarding the specified attribute of the storage subsystem is placed into the first-operand location. The first operand is 64 bits.

The second-operand address is not used to address data; rather, the rightmost 24 bits of the address are treated as a code specifying which attribute is returned in general register R_1.

The rightmost 24 bits of the second-operand address are defined as follows:



Attribute-Set Indication (ASI): Bit positions 54-55 of the second-operand address contain a 2-bit unsigned binary integer indicating the set of attributes from which an attribute to be extracted is selected. The attribute-set indications are:

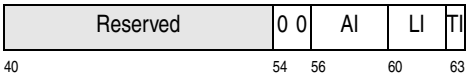
0 Cache attributes

1 CPU attributes

All other attribute-set-indication values are reserved.

Cache-Attribute-Set Operation

The rightmost 24 bits of the second-operand address are defined as follows:



The attribute, level, and type indications for the cache-attribute set are as follows:

Attribute Indication (AI): Bit positions 56-59 of the second-operand address contain a 4-bit unsigned binary integer indicating the cache attribute to be extracted, as follows:

0 Extract topology summary

1 Extract line size of the cache, in bytes

2 Extract total size of the cache, in bytes

3 Extract set-associativity level of the cache

4-15 Reserved

Level Indication (LI): Bit positions 60-62 of the second-operand address contain a 3-bit unsigned binary integer indicating the level of the cache for which the cache attribute is to be extracted, with 0 indicating the first-level cache, 1 indicating the second-level cache and so forth. If a cache level is not implemented on a model, its corresponding level indication is reserved.

Type Indication (TI): Bit 63 of the second-operand address indicates the type of cache for which the cache attribute is to be extracted, with 0 indicating the data cache and 1 indicating the instruction cache. When a cache level has a unified data and instruction cache, the same result is returned regardless of the type indication.

When the attribute indication is zero, the level indication (LI) and type indication (TI) are ignored.

CPU-Attribute-Set Operation

The rightmost 24 bits of the second-operand address are defined as follows:

The attribute indication for the CPU-attribute set are as follows:

Attribute Indication (AI): Bit positions 56-59 of the second-operand address contain a 4-bit unsigned binary integer indicating the CPU attribute to be extracted, as follows:

0 Extract CPU speed

1-15 Reserved

Reserved: Bit positions 60-63 of the second-operand address are reserved and ignored.

Common Operation

Bits 0-39 of the second-operand address are ignored. Bits 40-53 of the second-operand address are reserved and should contain zeros.

Bits 0-63 of general register R_1 are set to ones if any of the following apply:

- A reserved bit position in the second-operand address contains one and is otherwise not specifically designated as ignored.
- A reserved ASI, AI, or, when applicable, LI value is specified and is otherwise not specifically designated as ignored.

- The specified ASI and AI values and, when applicable, the LI value are all valid, but the model does not provide the data.

The contents of general register R_3 are ignored, however the R_3 field should specify register 0; otherwise, the program may not operate compatibly in the future.

Cache-Attribute Results (ASI = 0)

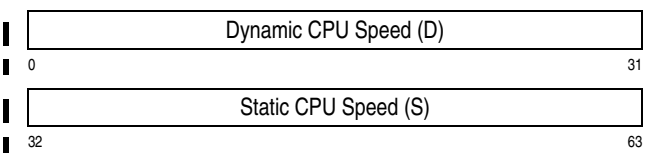
When the cache attribute indication is zero, a summary of each level of cache is returned in general register R_1. Each summary field is eight bits, where bits 0-7 of the register contain the summary for the first-level cache, bits 8-15 contain the summary for the second-level cache, and so forth. The contents of an eight-bit summary field are as follows:

Bits	Meaning
0-3	Reserved, stored as zeros
4-5	Cache scope, as follows: 00 Cache does not exist at this level 01 Cache is private to the CPU 10 Cache may be shared by multiple CPUs 11 Reserved
6-7	When bit positions 4-5 contain a nonzero value, bit positions 6-7 contain the cache type, as follows: 00 Separate instruction and data caches exist at this level 01 Only an instruction cache exists at this level 10 Only a data cache exists at this level 11 A unified instruction and data cache exists at this level When bit positions 4-5 contain zeros, bit positions 6-7 also contain zeros.

When the cache attribute indication is one, two, or three, the attribute value is returned in general register R_1, as a 64-bit unsigned integer.

CPU-Attribute Results (ASI = 1)

When the CPU attribute indication is zero, the CPU speeds are returned in general register R_1, specified in the following format:



The fields are defined as follows:

Bits	Meaning
0-31	Bit positions 0-31 (D) contain a 32-bit unsigned binary integer whose value represents the dynamic CPU speed, encoded as the approximate number of CPU cycles per microsecond.
32-63	Bit positions 32-63 (S) contain a 32-bit unsigned binary integer whose value represents the static CPU speed, encoded as the approximate number of CPU cycles per microsecond.

The dynamic CPU speed may or may not be the same as the static CPU speed. The static CPU speed reflects a CPU running at its nominal capacity and corresponds to the particular model. The dynamic CPU speed reflects a CPU running at a changed capacity caused by, for example, a manual control set to a power-save position.

Condition Code: The code remains unchanged.

Program Exceptions:

- Operation (if the general-instructions-extension facility is not installed)
- Transaction constraint

Programming Notes:

1. Knowledge of the cache line size is useful when determining the placement of PREFETCH DATA and PREFETCH DATA RELATIVE LONG instructions.

2. The value placed in the first-operand location is model dependent and may differ significantly from one model to another. For example, the size of the data cache on one model may be smaller or larger than the size of the data cache on another model. To ensure compatible operation across multiple models, the program should not rely on any particular value being returned in the first-operand location.

3. When the attribute-set indication is 00 binary and the attribute indication is 0000 binary, bit positions 4-5 of each 8-bit field returned in general register R_1 contain a cache-scope indication. When the cache-scope indication is 10 binary, the cache may be shared by more than one CPU

in the configuration. The System Library publication for the model may provide further explanation of cache topology.

4. When the attribute-set indication is 00 binary and the attribute indication is 0000 binary, the summary indications are returned in contiguous 8-bit fields in general register R_1. Thus if the program scans the resulting summary-indication fields from left to right and encounters an 8-bit field containing all zeros, no further meaningful fields exist.

5. Cache information returned reflects the depth specified by the level indication and the cache information accessible from the issuing CPU. The total sizes and associativity do not include other, parallel cache structures accessible from other CPUs of the configuration.

6. The static CPU speed (S) may be less than, equal to, or greater than the dynamic CPU speed (D), as follows:

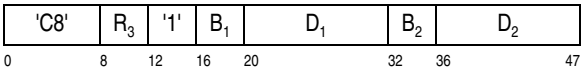
   - When the CPU is operating at nominal capacity, the D and S values are equal.
   - The sections “Capacity-Change Reason (CCR)” on page 10-143 and “Capacity-Adjustment Indication (CAI)” on page 10-143 describe various reasons why a CPU may not be operating at nominal capacity, in which case, D may be less than S.
   - Under certain circumstances, D may be greater than S.

7. Depending on the model, the dynamic CPU speed (D) values may be different from one CPU to another. Similarly, the static CPU speed values may be different from one CPU to another.

8. The dynamic CPU speed returned for CPU-attribute-indicator zero is the same as would be returned by the QUERY SAMPLING INFORMATION (QSI) instruction of the CPU-measurement facility. The value returned is also subject to the same factors that affect the value returned by QSI.

EXTRACT CPU TIME

ECTG $D_1(B_1),D_2(B_2),R_3$ [SSF]



The value of the current CPU timer is subtracted from the first operand, and the difference is placed in general register 0. The second operand is placed unchanged in general register 1. The eight bytes at the third operand location replace the contents of general register R_3.

The first and second operands and the results in general registers 0, 1, and R_3 are treated as 64-bit unsigned binary integers.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general register R_3 constitute the address of the third operand, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the register constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address. In the access-register mode, the contents of access register R_3 are used for accessing the third operand.

All operands are fetched before any general register is modified. If the R_3 field designates general register 0, it is unpredictable whether the result placed in the register is the first operand minus the CPU timer or the third operand. If the R_3 field designates general register 1, it is unpredictable whether the result placed in the register is the second operand or the third operand.

The contents of the registers just described are shown in Figure 7-299.

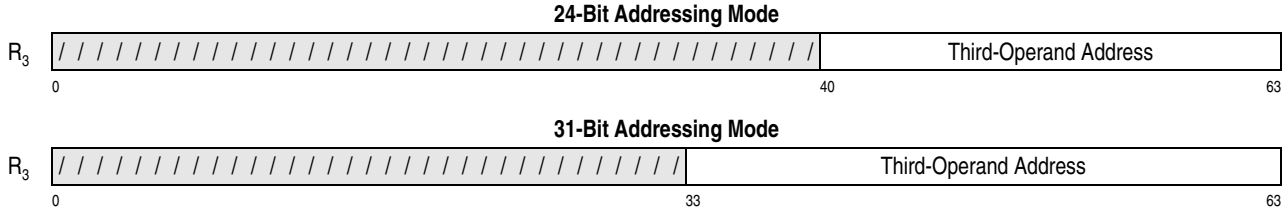


Figure 7-299. General Register Assignment for EXTRACT CPU TIME (Part 1 of 2)

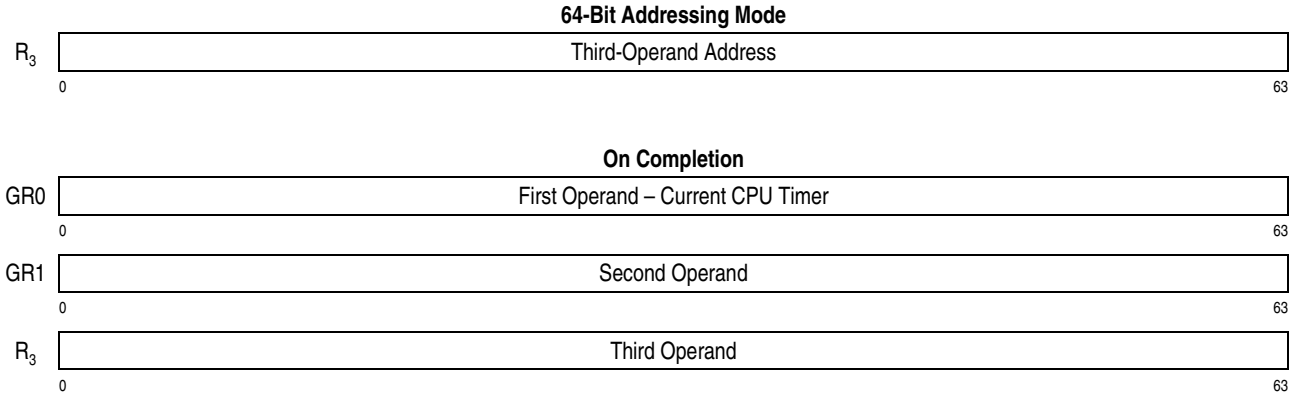


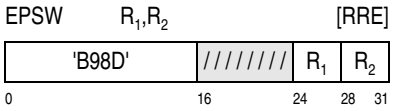
Figure 7-299. General Register Assignment for EXTRACT CPU TIME (Part 2 of 2)

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operands 1, 2, and 3)
- Operation (the extract-CPU-time facility is not installed)
- Transaction constraint

EXTRACT PSW



Bits 0-11 and 13-31 of the current PSW are placed in bit positions 32-43 and 45-63 of the first operand, and bits 0-31 of the operand remain unchanged. When the configuration is in the z/Architecture architectural mode, bit 44 of the first operand is set to zero, and when the configuration is in the ESA/390-compatibility mode, bit 44 of the first operand is set to one.

Subsequently, bits 32-63 of the current PSW are placed in bit positions 32-63 of the second operand, and bits 0-31 of the operand remain unchanged. The action associated with the second operand is not performed if the R_2 field is zero.

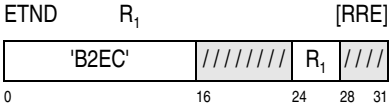
Bits 0-63 of the 128-bit PSW are illustrated in Figure 4-2 on page 4-5.

Condition Code: The code remains unchanged.

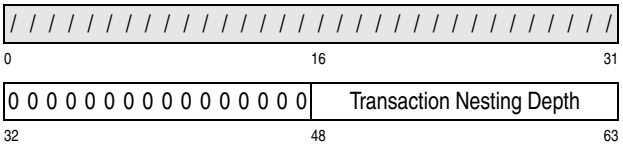
Program Exceptions:

- Transaction constraint

EXTRACT TRANSACTION NESTING DEPTH



The current transaction nesting depth is placed in bits 48-63 of general register R_1. Bits 0-31 of the register remain unchanged, and bits 32-47 of the register are set to zero, as illustrated below.



A special-operation exception is recognized and the operation is suppressed if the transactional-execution control, bit 8 of control register 0, is zero.

Condition Code: The code remains unchanged.

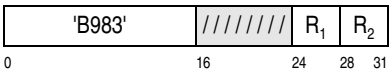
Program Exceptions:

- Operation (transactional-execution facility not installed)
- Special operation
- Transaction constraint

Programming Note: A transaction nesting depth of zero indicates that the CPU is not in the transactional-execution mode.

FIND LEFTMOST ONE

FLOGR R_1,R_2 [RRE]



Bits 0-63 of general register R_2 are scanned left to right for the leftmost one bit. A 64-bit binary integer designating the bit position of the leftmost one bit, or 64 if there is no one bit, is placed in general register R_1. When a one bit is found in the second operand, the original contents of general register R_2, with the leftmost one bit set to zero, are placed in general register $R_1 + 1$. When a one bit is not found, the contents of general register $R_1 + 1$ are set to zeros.

The condition code indicates whether or not a one bit was found in the second operand.

The R_1 field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

Resulting Condition Code:

0 No one bit found
1 --
2 One bit found
3 --

Program Exceptions:

- Operation (if the extended-immediate facility is not installed)
- Specification

Programming Notes:

1. An example of the use of the FIND LEFTMOST ONE instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

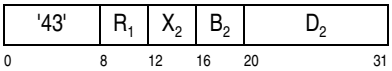
2. When the R_1 and R_2 fields designate the same register, the original contents of general register R_2 are replaced by the resulting bit-position value, or 64 if no one bit was found.

3. When the R_2 field designates general register $R_1 + 1$, the leftmost one bit (if any) of general register R_2 is set to zero; if no one bit is found, the entire register is set to zero.

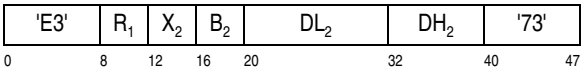
4. When the R_2 field designates neither the even nor the odd registers designated by the R_1 field, then general register R_2 is not modified.

INSERT CHARACTER

IC $R_1,D_2(X_2,B_2)$ [RX-a]



ICY $R_1,D_2(X_2,B_2)$ [RXY-a]



The byte at the second-operand location is inserted into bit positions 56-63 of general register R_1. The remaining bits in the register remain unchanged.

The displacement for IC is treated as a 12-bit unsigned binary integer. The displacement for ICY is treated as a 20-bit signed binary integer.

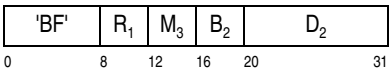
Condition Code: The code remains unchanged.

Program Exceptions:

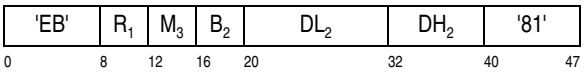
- Access (fetch, operand 2)
- Operation (ICY, if the long-displacement facility is not installed)

INSERT CHARACTERS UNDER MASK

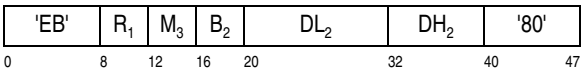
ICM $R_1,M_3,D_2(B_2)$ [RS-b]



ICMY $R_1,M_3,D_2(B_2)$ [RSY-b]



ICMH $R_1,M_3,D_2(B_2)$ [RSY-b]



Bytes from contiguous locations beginning at the second-operand address are inserted into general register R_1 under control of a mask.

The contents of the M_3 field are used as a mask. These four bits, left to right, correspond one for one with four bytes, left to right, of general register R_1. For INSERT CHARACTERS UNDER MASK (ICM, ICMY), the four bytes to which the mask bits correspond are in the low-order half, bit positions 32-63 of general register R_1. For INSERT CHARACTERS UNDER MASK (ICMH), the four bytes are in the high-order half, bit positions 0-31, of the register. The byte positions corresponding to ones in the mask are filled, left to right, with bytes from successive storage locations beginning at the second-operand address. When the mask is not zero, the length of the second operand is equal to the number of ones in the mask. The bytes in the general register corresponding to zeros in the mask remain unchanged. For ICM and ICMY, bits 0-31 of the register remain unchanged, and, for ICMH, bits 32-63 remain unchanged.

The resulting condition code is based on the mask and on the value of the bits inserted. When the mask is zero or when all inserted bits are zeros, the condition code is set to 0. When the inserted bits are not all zeros, the code is set according to the leftmost bit of the storage operand: if this bit is one, the code is set to 1; if this bit is zero, the code is set to 2.

When the mask is not zero, exceptions associated with storage-operand access are recognized only for the number of bytes specified by the mask. When the mask is zero, access exceptions are recognized for one byte at the second-operand address.

The displacement for ICM is treated as a 12-bit unsigned binary integer. The displacement for ICMY and ICMH is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 All inserted bits zeros, or mask bits all zeros
1 Leftmost inserted bit one
2 Leftmost inserted bit zero, and not all inserted bits zeros
3 --

Program Exceptions:

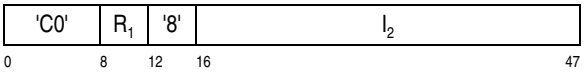
- Access (fetch, operand 2)
- Operation (ICMY, if the long-displacement facility is not installed)

Programming Notes:

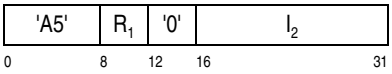
1. Examples of the use of the INSERT CHARACTERS UNDER MASK instruction are given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. The condition code for INSERT CHARACTERS UNDER MASK (ICM, ICMY only) is defined such that, when the mask is 1111, the instruction causes the same condition code to be set as for LOAD AND TEST (LTR only) Thus, the instruction may be used as a storage-to-register load-and-test operation.
3. INSERT CHARACTERS UNDER MASK (ICM, ICMY) with a mask of 1111 or 0001 performs a function similar to that of a LOAD (L) or INSERT CHARACTER (IC) instruction, respectively, with the exception of the condition-code setting. However, the performance of INSERT CHARACTERS UNDER MASK may be slower.

INSERT IMMEDIATE

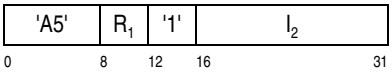
IIHF R_1,I_2 [RIL-a]



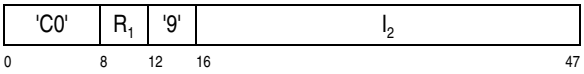
IIHH R_1,I_2 [RI-a]



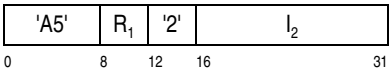
IIHL R_1,I_2 [RI-a]



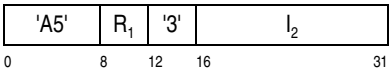
IILF R_1,I_2 [RIL-a]



IILH R_1,I_2 [RI-a]



IILL R_1,I_2 [RI-a]



The second operand is placed in bit positions of the first operand. The remainder of the first operand remains unchanged.

For each instruction, the bit positions of the first operand that are loaded with the second operand are as follows:

Instruction	Bit Positions Loaded
IIHF	0-31
IIHH	0-15
IIHL	16-31
IILF	32-63
IILH	32-47
IILL	48-63

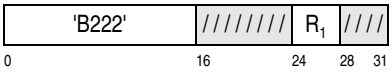
Condition Code: The code remains unchanged.

Program Exceptions:

- Operation (IIHF and IILF, if the extended-immediate facility is not installed)

INSERT PROGRAM MASK

IPM R_1 [RRE]



The condition code and program mask from the current PSW are inserted into bit positions 34 and 35 and 36-39, respectively, of general register R_1. Bits 32 and 33 of the register are set to zeros; bits 0-31 and 40-63 are left unchanged.

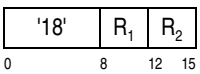
Condition Code: The code remains unchanged.

Program Exceptions: None.

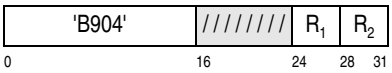
LOAD

Register-and-register formats:

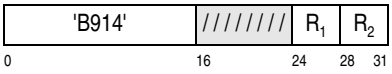
LR R_1,R_2 [RR]



LGR R_1,R_2 [RRE]

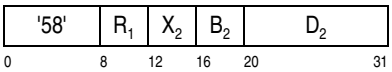


LGFR R_1,R_2 [RRE]

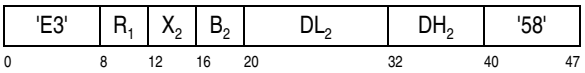


Register-and-storage formats:

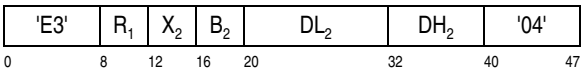
L $R_1,D_2(X_2,B_2)$ [RX-a]



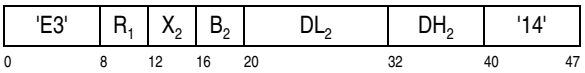
LY $R_1,D_2(X_2,B_2)$ [RXY-a]



LG $R_1,D_2(X_2,B_2)$ [RXY-a]



LGF $R_1,D_2(X_2,B_2)$ [RXY-a]

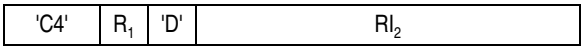


LOAD IMMEDIATE

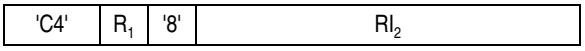
LGFI R_1,I_2 [RIL-a]

LOAD RELATIVE LONG

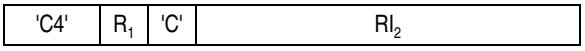
LRL R_1,RI_2 [RIL-b]



LGRL R_1,RI_2 [RIL-b]



LGFRL R_1,RI_2 [RIL-b]



The second operand is placed unchanged at the first-operand location, except that, for LOAD (LGFR, LGF), LOAD IMMEDIATE (LGFI), and LOAD RELATIVE LONG (LGFRL), it is sign extended.

For LOAD (LR, L, LY) and LOAD RELATIVE LONG (LRL), the operands are 32 bits, and, for LOAD (LGR, LG), and LOAD RELATIVE LONG (LGRL), the operands are 64 bits. For LOAD (LGFR, LGF), LOAD IMMEDIATE (LGFI), and LOAD RELATIVE LONG (LGFRL), the second operand is treated as a 32-bit signed binary integer, and the first operand is treated as a 64-bit signed binary integer.

The displacement for L is treated as a 12-bit unsigned binary integer. The displacement for LY, LG, and LGF is treated as a 20-bit signed binary integer.

For LOAD IMMEDIATE, the contents of the I_2 field form the second operand directly.

For LOAD RELATIVE LONG, the contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the address of the second operand in storage. When DAT is on, the second operand is accessed using the same addressing-space mode as that used to access the instruction. When DAT is off, the second operand is accessed using a real address.

For LOAD RELATIVE LONG (LRL, LGFRL), the second operand must be aligned on a word boundary, and for LOAD RELATIVE LONG (LGRL), the second operand must be aligned on a doubleword boundary; otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

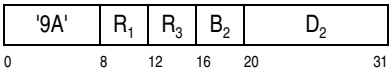
- Access (fetch, operand 2 of L, LY, LG, LGF, LRL, LGRL, and LGFRL only)
- Operation (LY, if the long-displacement facility is not installed; LGFI, if the extended-immediate facility is not installed; LRL, LGRL, LGFRL, if the general-instructions-extension facility is not installed)
- Specification (LRL, LGFRL, LGRL only)

Programming Notes:

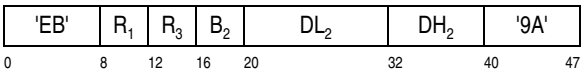
1. An example of the use of the LOAD instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. For LOAD RELATIVE LONG, the second operand must be aligned on an integral boundary corresponding to the operand’s size.
3. When LOAD RELATIVE LONG is the target of an execute-type instruction, the second-operand address is relative to the target address.

LOAD ACCESS MULTIPLE

LAM $R_1,R_3,D_2(B_2)$ [RS-a]



LAMY $R_1,R_3,D_2(B_2)$ [RSY-a]



The set of access registers starting with access register R_1 and ending with access register R_3 is loaded from the locations designated by the second-operand address.

The storage area from which the contents of the access registers are obtained starts at the location designated by the second-operand address and continues through as many storage words as the number of access registers specified. The access registers are loaded in ascending order of their register numbers, starting with access register R_1 and continuing up to and including access register R_3, with access register 0 following access register 15.

The displacement for LAM is treated as a 12-bit unsigned binary integer. The displacement for LAMY is treated as a 20-bit signed binary integer.

The second operand must be designated on a word boundary; otherwise, a specification exception is recognized.

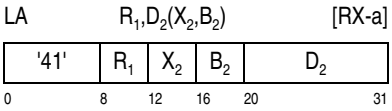
Condition Code: The code remains unchanged.

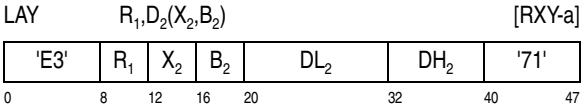
Program Exceptions:

- Access (fetch, operand 2)

- Operation (LAMY, if the long-displacement facility is not installed)
- Specification
- Transaction constraint

LOAD ADDRESS





The address specified by the X_2, B_2, and D_2 fields is placed in general register R_1. The address computation follows the rules for address arithmetic.

In the 24-bit addressing mode, the address is placed in bit positions 40-63, bits 32-39 are set to zeros, and bits 0-31 remain unchanged. In the 31-bit addressing mode, the address is placed in bit positions 33-63, bit 32 is set to zero, and bits 0-31 remain unchanged. In the 64-bit addressing mode, the address is placed in bit positions 0-63.

The displacement for LA is treated as a 12-bit unsigned binary integer. The displacement for LAY is treated as a 20-bit signed binary integer.

No storage references for operands take place, and the address is not inspected for access exceptions.

Condition Code: The code remains unchanged.

Program Exceptions:

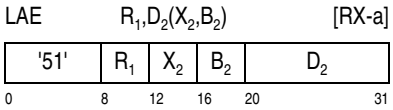
- Operation (LAY if the long-displacement facility is not installed)

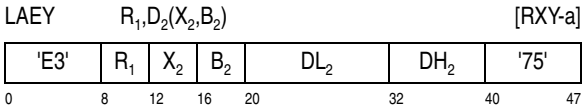
Programming Notes:

1. An example of the use of the LOAD ADDRESS instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. LOAD ADDRESS may be used to increment the rightmost bits of a general register, other than register 0, by the contents of the D_2 field of the instruction. LOAD ADDRESS (LAY) may also be used to decrement the rightmost bits of a register, other than register 0. The register to be incremented should be designated by R_1 and by either X_2 (with B_2 set to zero) or B_2 (with X_2 set to zero). The instruction increments 24 bits in the 24-bit addressing mode, 31 bits in the 31-bit addressing mode, and 64 bits in the 64-bit addressing mode.

LOAD ADDRESS EXTENDED





The address specified by the X_2, B_2, and D_2 fields is placed in general register R_1. Access register R_1 is loaded with a value that depends on the current value of the address-space-control bits, bits 16 and 17 of the PSW. If the address-space-control bits are 01 binary, the value placed in the access register also depends on whether the B_2 field is zero or nonzero.

The address computation follows the rules for address arithmetic. In the 24-bit addressing mode, the address is placed in bit positions 40-63 of general register R_1, bits 32-39 are set to zeros, and bits 0-31 remain unchanged. In the 31-bit addressing mode, the address is placed in bit positions 33-63, bit 32 is set to zero, and bits 0-31 remain unchanged. In the 64-bit addressing mode, the address is placed in bit positions 0-63.

The value placed in access register R_1 is as shown in the following table:

PSW Bits 16 and 17	Value Placed in Access Register R_1
00	00000000 hex (zeros in bit positions 0-31)
10	00000001 hex (zeros in bit positions 0-30 and one in bit position 31)
01	If B_2 field is zero: 00000000 hex (zeros in bit positions 0-31) If B_2 field is nonzero: Contents of access register B_2

PSW Bits 16 and 17	Value Placed in Access Register R_1
11	00000002 hex (zeros in bit positions 0-29 and 31, and one in bit position 30)

However, when PSW bits 16 and 17 are 01 binary and the B_2 field is nonzero, bit positions 0-6 of access register B_2 must contain all zeros; otherwise, the results in general register R_1 and access register R_1 are unpredictable.

The displacement for LAE is treated as a 12-bit unsigned binary integer. The displacement for LAEY is treated as a 20-bit signed binary integer.

No storage references for operands take place, and the address is not inspected for access exceptions.

Condition Code: The code remains unchanged.

Program Exceptions:

- Operation (LAEY; if the general-instructions-extension facility is not installed)
- Transaction constraint

Programming Notes:

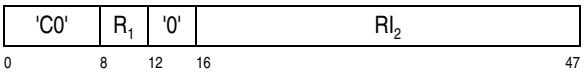
1. When DAT is on, the different values of the address-space-control bits correspond to translation modes as follows:

PSW Bits 16 and 17	Translation Mode
00	Primary-space mode
10	Secondary-space mode
01	Access-register mode
11	Home-space mode

2. In the access-register mode, the value 00000000 hex in an access register designates the primary address space, and the value 00000001 hex designates the secondary address space. The value 00000002 hex designates the home address space if the control program assigns entry 2 of the dispatchable-unit access list as designating the home address space and places a zero access-list-entry sequence number (ALESN) in that entry.

LOAD ADDRESS RELATIVE LONG

LARL R_1,RI_2 [RIL-b]



The address specified by the RI_2 field is placed in general register R_1. The address computation follows the rules for the branch address of BRANCH RELATIVE ON CONDITION LONG and BRANCH RELATIVE AND SAVE LONG.

In the 24-bit addressing mode, the address is placed in bit positions 40-63, bits 32-39 are set to zeros, and bits 0-31 remain unchanged. In the 31-bit addressing mode, the address is placed in bit positions 33-63, bit 32 is set to zero, and bits 0-31 remain unchanged. In the 64-bit addressing mode, the address is placed in bit positions 0-63.

The contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the computed address.

No storage references for operands take place, and the address is not inspected for access exceptions.

Condition Code: The code remains unchanged.

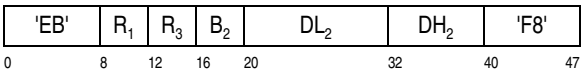
Program Exceptions: None.

Programming Notes:

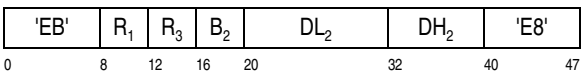
1. Only even addresses (halfword addresses) can be generated. If an odd address is desired, LOAD ADDRESS can be used to add one to an address formed by LOAD ADDRESS RELATIVE LONG.
2. When LOAD ADDRESS RELATIVE LONG is the target of an execute-type instruction, the address produced is relative to the location of the LOAD ADDRESS RELATIVE LONG instruction, not of the execute-type instruction. This is consistent with the operation of the relative-branch instructions.

LOAD AND ADD

LAA $R_1,R_3,D_2(B_2)$ [RSY-a]



LAAG $R_1,R_3,D_2(B_2)$ [RSY-a]



The second operand is added to the third operand, and the sum is placed at the second-operand location. Subsequently, the original contents of the second operand (prior to the addition) are placed unchanged at the first-operand location.

For LAA, the operands are treated as being 32-bit signed binary integers. For LAAG, the operands are treated as being 64-bit signed binary integers.

All accesses to the second-operand location appear to be a block-concurrent interlocked-update reference as observed by other CPUs and the I/O subsystem. A specific-operand-serialization operation is performed.

The displacement is treated as a 20-bit signed binary integer.

The second operand of LAA must be designated on a word boundary. The second operand of LAAG must be designated on a doubleword boundary. Otherwise, a specification exception is recognized.

Resulting Condition Code:

0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

Program Exceptions:

- Access (fetch and store, operand 2)
- Fixed-point overflow
- Operation (if the interlocked-access facility 1 is not installed)
- Specification
- Transaction constraint

Programming Notes:

1. Except for the case where the R_1 and R_3 fields designate the same register, general register R_3 is unchanged.

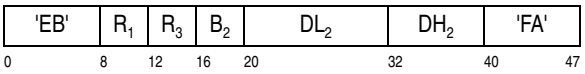
2. The operation of LOAD AND ADD, LOAD AND ADD LOGICAL, LOAD AND AND, LOAD AND EXCLUSIVE OR, and LOAD AND OR may be expressed as follows.

   ```
   temp ← operand_2;
   operand_2 ← temp OP operand_3;
   operand_1 ← temp;
   ```

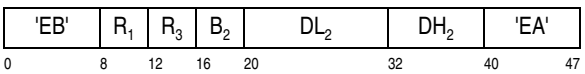
   OP represents the arithmetic or logical operation being performed by the instruction.

LOAD AND ADD LOGICAL

LAAL $R_1,R_3,D_2(B_2)$ [RSY-a]



LAALG $R_1,R_3,D_2(B_2)$ [RSY-a]



The second operand is added to the third operand, and the sum is placed at the second-operand location. Subsequently, the original contents of the second operand (prior to the addition) are placed unchanged at the first-operand location.

For LAAL, the operands are treated as being 32-bit unsigned binary integers. For LAALG, the operands are treated as being 64-bit unsigned binary integers.

All accesses to the second-operand location appear to be a block-concurrent interlocked-update reference as observed by other CPUs and the I/O subsystem. A specific-operand-serialization operation is performed.

The displacement is treated as a 20-bit signed binary integer.

The second operand of LAAL must be designated on a word boundary. The second operand of LAALG must be designated on a doubleword boundary. Otherwise, a specification exception is recognized.

Resulting Condition Code:

0 Result zero; no carry
1 Result not zero; no carry
2 Result zero; carry
3 Result not zero; carry

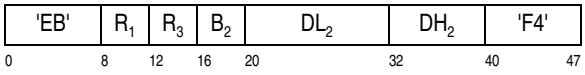
Program Exceptions:

- Access (fetch and store, operand 2)
- Operation (if the interlocked-access facility 1 is not installed)
- Specification
- Transaction constraint

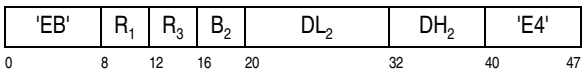
Programming Note: See the programming notes for LOAD AND ADD.

LOAD AND AND

LAN $R_1,R_3,D_2(B_2)$ [RSY-a]



LANG $R_1,R_3,D_2(B_2)$ [RSY-a]



The AND of the second operand and third operand is placed at the second-operand location. Subsequently, the original contents of the second operand (prior to the AND operation) are placed unchanged at the first-operand location.

For LAN, the operands are 32 bits. For LANG, the operands are 64 bits.

The connective AND is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit positions in both operands contain ones; otherwise, the result bit is set to zero.

All accesses to the second-operand location appear to be a block-concurrent interlocked-update reference as observed by other CPUs and the I/O subsystem. A specific-operand-serialization operation is performed.

The displacement is treated as a 20-bit signed binary integer.

The second operand of LAN must be designated on a word boundary. The second operand of LANG must be designated on a doubleword boundary. Otherwise, a specification exception is recognized.

Resulting Condition Code:

0 Result zero
1 Result not zero
2 --
3 --

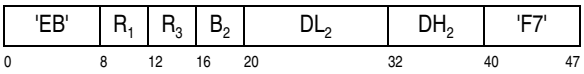
Program Exceptions:

- Access (fetch and store, operand 2)
- Operation (if the interlocked-access facility 1 is not installed)
- Specification
- Transaction constraint

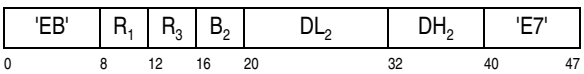
Programming Note: See the programming notes for LOAD AND ADD.

LOAD AND EXCLUSIVE OR

LAX $R_1,R_3,D_2(B_2)$ [RSY-a]



LAXG $R_1,R_3,D_2(B_2)$ [RSY-a]



The EXCLUSIVE OR of the second operand and third operand is placed at the second-operand location. Subsequently, the original contents of the second operand (prior to the EXCLUSIVE OR operation) are placed unchanged at the first-operand location.

For LAX, the operands are 32 bits. For LAXG, the operands are 64 bits.

The connective exclusive OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the bits in the corresponding bit positions in the two operands are unlike; otherwise, the result bit is set to zero.

All accesses to the second-operand location appear to be a block-concurrent interlocked-update reference as observed by other CPUs and the I/O subsys-

tem. A specific-operand-serialization operation is performed.

The displacement is treated as a 20-bit signed binary integer.

The second operand of LAX must be designated on a word boundary. The second operand of LAXG must be designated on a doubleword boundary. Otherwise, a specification exception is recognized.

Resulting Condition Code:

0 Result zero
1 Result not zero
2 --
3 --

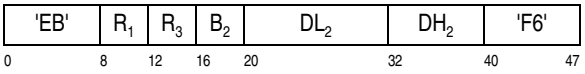
Program Exceptions:

- Access (fetch and store, operand 2)
- Operation (if the interlocked-access facility 1 is not installed)
- Specification
- Transaction constraint

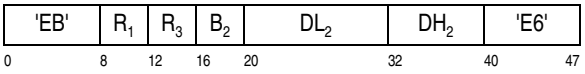
Programming Note: See the programming notes for LOAD AND ADD.

LOAD AND OR

LAO $R_1,R_3,D_2(B_2)$ [RSY-a]



LAOG $R_1,R_3,D_2(B_2)$ [RSY-a]



The OR of the second operand and third operand is placed at the second-operand location. Subsequently, the original contents of the second operand (prior to the OR operation) are placed unchanged at the first-operand location.

For LAO, the operands are 32 bits. For LAOG, the operands are 64 bits.

The connective OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit position in one or both operands contains a one; otherwise, the result bit is set to zero.

All accesses to the second-operand location appear to be a block-concurrent interlocked-update reference as observed by other CPUs and the I/O subsystem. A specific-operand-serialization operation is performed.

The displacement is treated as a 20-bit signed binary integer.

The second operand of LAO must be designated on a word boundary. The second operand of LAOG must be designated on a doubleword boundary. Otherwise, a specification exception is recognized.

Resulting Condition Code:

0 Result zero
1 Result not zero
2 --
3 --

Program Exceptions:

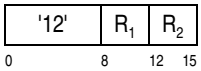
- Access (fetch and store, operand 2)
- Operation (if the interlocked-access facility 1 is not installed)
- Specification
- Transaction constraint

Programming Note: See the programming notes for LOAD AND ADD.

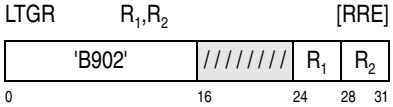
LOAD AND TEST

Register-and-register formats:

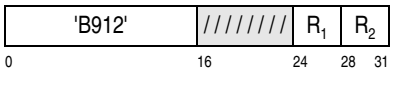
LTR R_1,R_2 [RR]



LTGR R_1,R_2 [RRE]

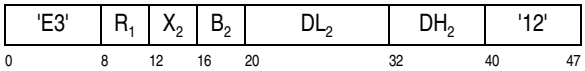


LTGFR R_1,R_2 [RRE]

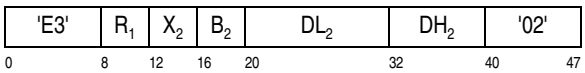


Register-and-storage formats:

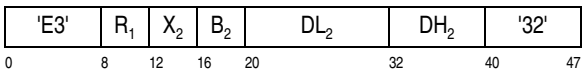
LT $R_1,D_2(X_2,B_2)$ [RXY-a]



LTG $R_1,D_2(X_2,B_2)$ [RXY-a]



LTGF $R_1,D_2(X_2,B_2)$ [RXY-a]



The second operand is placed unchanged at the first-operand location, except that, for LTGF and LTGFR, it is sign extended. The sign and magnitude of the second operand, treated as a signed binary integer, are indicated in the condition code.

For LT and LTR, the operands are 32 bits, and, for LTG and LTGR, the operands are 64 bits. For LTGF and LTGFR, the second operand is 32 bits, and the first operand is treated as a 64-bit signed binary integer.

Resulting Condition Code:

0 Result zero
1 Result less than zero
2 Result greater than zero
3 --

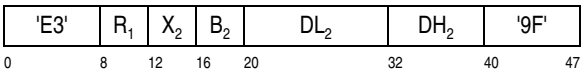
Program Exceptions:

- Access (fetch, operand 2 of LT, LTG, and LTGF only)
- Operation (LT and LTG, if the extended-immediate facility is not installed; LTGF, if the general-instructions-extension facility is not installed)

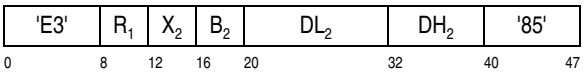
Programming Note: For LOAD AND TEST (LTR and LTGR) when the R_1 and R_2 fields designate the same register, the operation is equivalent to a test without data movement.

LOAD AND TRAP

LAT $R_1,D_2(X_2,B_2)$ [RXY-a]



LGAT $R_1,D_2(X_2,B_2)$ [RXY-a]



The second operand is placed unchanged at the first operand location. If all zeros are placed at the first operand location, a compare-and-trap-instruction data exception is recognized.

For LAT, the second operand is treated as a 32-bit signed integer and placed in bit positions 32-63 of general register R_1, and bit positions 0-31 remain unchanged.

For LGAT, the second operand is treated as a 64-bit signed integer and placed in bit positions 0-63 of general register R_1.

Condition Code: The code remains unchanged.

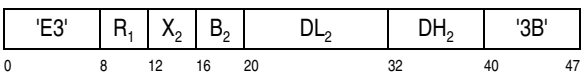
Program Exceptions:

- Data
- Access (fetch, operand 2)
- Operation (if the load-and-trap facility is not installed)

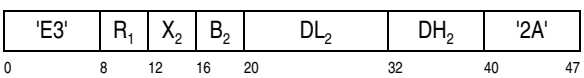
Programming Note: Possible uses of LOAD AND TRAP include checking for pointers containing zero (null pointer).

LOAD AND ZERO RIGHTMOST BYTE

LZRF $R_1,D_2(X_2,B_2)$ [RXY-a]



LZRG $R_1,D_2(X_2,B_2)$ [RXY-a]



The second operand, with the rightmost byte set to zero, is placed at the first-operand location. For LZRF, the first and second operands are 32 bits, and for LZRG, the first and second operands are 64 bits.

The displacement is treated as a 20-bit signed binary integer.

It is unpredictable whether an access exception is recognized for the rightmost byte of the second operand.

Condition Code: The code remains unchanged.

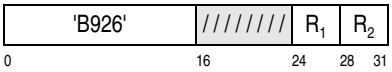
Program Exceptions:

- Access (fetch, operand 2)
- Operation (if the load-and-zero-rightmost-byte facility is not installed)

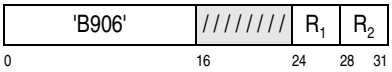
LOAD BYTE

Register-and-register formats:

LBR R_1,R_2 [RRE]

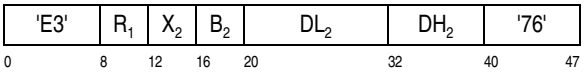


LGBR R_1,R_2 [RRE]

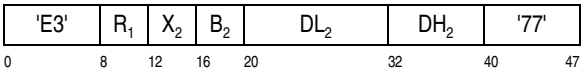


Register-and-storage formats:

LB $R_1,D_2(X_2,B_2)$ [RXY-a]



LGB $R_1,D_2(X_2,B_2)$ [RXY-a]



The second operand is sign extended and placed at the first-operand location. The second operand is one byte in length and is treated as an eight-bit signed binary integer. For LOAD BYTE (LB and LGB), the second operand is a byte in storage. For LOAD BYTE (LBR and LGBR), the second operand is in bits 56-63 of general register R_2.

For LOAD BYTE (LB and LBR), the first operand is treated as a 32-bit signed binary integer. For LOAD BYTE (LGB and LGBR), the first operand is treated as a 64-bit signed binary integer.

The displacement for LB and LGB is treated as a 20-bit signed binary integer.

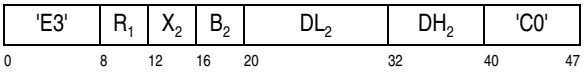
Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of LB and LGB)
- Operation (LB and LGB, if the long-displacement facility is not installed; LBR and LGBR, if the extended-immediate facility is not installed)

LOAD BYTE HIGH

LBH $R_1,D_2(X_2,B_2)$ [RXY-a]



The second operand is sign extended and placed at the first-operand location. The second operand is one byte in length and is treated as an eight-bit signed binary integer.

The first operand is treated as a 32-bit signed binary integer in bits 0-31 of general register R_1; bits 32-63 of the register are unchanged.

The displacement is treated as a 20-bit signed binary integer.

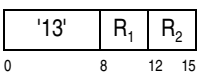
Condition Code: The code remains unchanged.

Program Exceptions:

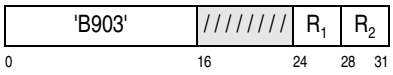
- Access (fetch, operand 2)
- Operation (if the high-word facility is not installed)

LOAD COMPLEMENT

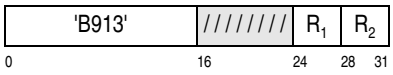
LCR R_1,R_2 [RR]



LCGR R_1,R_2 [RRE]



LCGFR R_1,R_2 [RRE]



The two's complement of the second operand is placed at the first-operand location. For LOAD COMPLEMENT (LCR), the second operand and result are treated as 32-bit signed binary integers. For LOAD COMPLEMENT (LCGR), they are treated as 64-bit signed binary integers. For LOAD COMPLEMENT (LCGFR), the second operand is treated as a 32-bit signed binary integer, and the result is treated as a 64-bit signed binary integer.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

Resulting Condition Code:

0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

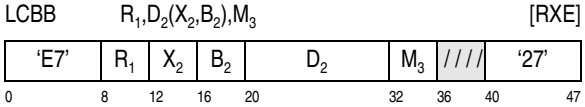
Program Exceptions:

- Fixed-point overflow (LCR and LCGR only)

Programming Note: The operation complements all numbers. Zero remains unchanged. For LCR or LCGR, the maximum negative 32-bit number or 64-bit number, respectively, remains unchanged, and an overflow condition occurs when the number is complemented. LCGFR complements the maximum negative 32-bit number without recognizing overflow.

LOAD COUNT TO BLOCK BOUNDARY

LCBB $R_1,D_2(X_2,B_2),M_3$ [RXE]



A 32-bit unsigned binary integer containing the number of bytes possible to load from the second operand location without crossing a specified block boundary, with a maximum of sixteen is placed in the first operand. If the number of bytes is greater than sixteen, sixteen is placed in the first operand.

The displacement is treated as a 12-bit unsigned integer.

The second operand address is not used to address data.

The M_3 field specifies a code that is used to signal the CPU as to the block boundary size to compute the number of possible bytes loaded. If a reserved value is specified then a specification exception is recognized.

Code	Boundary
0	64 Byte
1	128 Byte
2	256 Byte
3	512 Byte
4	1 K-byte
5	2 K-Byte
6	4 K-Byte
7-15	Reserved

Resulting Condition Code:

0 Operand one is sixteen
1 --
2 --
3 Operand one is less than sixteen

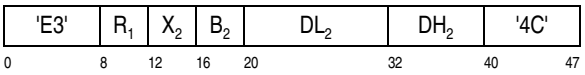
Program Exceptions:

- Operation (if the vector facility for z/Architecture is not installed)
- Specification

Programming Note: It is expected that LOAD COUNT TO BLOCK BOUNDARY will be used in conjunction with VECTOR LOAD TO BLOCK BOUNDARY to determine the number of bytes that were loaded.

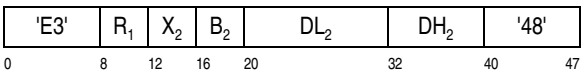
LOAD GUARDED

LGG $R_1,D_2(X_2,B_2)$ [RXY-a]



LOAD LOGICAL AND SHIFT GUARDED

LLGFSG $R_1,D_2(X_2,B_2)$ [RXY-a]



Note: See the section "Guarded-Storage Facility" on page 4-64 for details on the terminology used below.

A 64-bit intermediate result is formed as follows:

- For LOAD GUARDED: The second operand is a doubleword in storage. A specification exception is recognized and the operation is suppressed if the second-operand address is not on a doubleword boundary.

  In the 24-bit addressing mode, the intermediate result is formed from the concatenation of 40 binary zeros with bits 40-63 of the second operand. In the 31-bit addressing mode, the intermediate result is formed from the concatenation of 33 binary zeros with bits 33-63 of the second operand. In the 64-bit addressing mode, the intermediate result is formed from the entire second operand.

- For LOAD LOGICAL AND SHIFT GUARDED: The second operand is a word in storage. A specification exception is recognized and the operation is suppressed if the second-operand address is not on a word boundary.

  When the guarded-storage facility is enabled (by means of bit 59 of control register 2), the intermediate result is formed using the guarded-load-shift value (GLS, in bits 53-55 of the guarded-storage-designation register). When the guarded-storage facility is not enabled, the GLS value is assumed to be zero.

  In the 24-bit addressing mode, the intermediate result is formed from the concatenation of 40 binary zeros, bits (8+GLS) through 31 of the second operand, and GLS binary zeros. In the 31-bit addressing mode, the intermediate result is formed from the concatenation of 33 binary zeros, bits (1+GLS) through 31 of the second operand, and GLS binary zeros. In the 64-bit addressing mode, the intermediate result is formed from the concatenation of (32-GLS) binary zeros, the entire 32-bit second operand, and GLS binary zeros.

When the guarded-storage facility is enabled, the intermediate result is used as described in "Guarded-Storage-Event Detection" on page 4-69. If a guarded-storage event is recognized, then general register R_1 is not modified, and the instruction is completed as described in "Guarded-Storage-Event Processing" on page 4-70.

When either the guarded-storage facility is not enabled, or the facility is enabled but a guarded-storage event is not recognized, then the 64-bit intermediate result is placed into general register R_1, and the instruction is completed.

The displacement is treated as a 20-bit signed binary integer.

The guarded-storage-event parameter list (GSEPL) is only accessed when a guarded-storage event is recognized. Store-type accesses apply to the entire GSEPL.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, second operand; when a guarded-storage-event is recognized, fetch and store, GSEPL fields)
- Operation (guarded-storage facility not installed)

- Specification

1.-7.	Exceptions with the same priority as the priority of program-interruption conditions for the general case.
8.	Access exceptions for the second operand in storage.
9.	Completion with no guarded-storage-event recognized.
10.	Side-effect access exceptions for the guarded-storage-event parameter list.
11.	Completion with a guarded-storage-event recognized.

Figure 7-300. Priority of Execution: LGG and LLGFSG

Programming Notes:

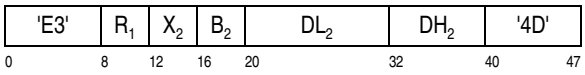
1. LOAD LOGICAL AND SHIFT GUARDED may be useful in loading what are sometimes referred to as *compressed pointers* in which some number of rightmost bits of the pointer address are absent in storage and assumed to be zeros.

2. When the guarded-storage facility is installed in a configuration, the LOAD GUARDED and LOAD LOGICAL AND SHIFT GUARDED instructions can be executed regardless of the contents of the guarded-storage-facility-enablement control (GSFE, bit 59 of control register 2). However, guarded-storage events are only recognized as a result of executing LGG or LLGFSG when (a) the GSFE control is one, and (b) the GSSM is nonzero. The GSSM cannot be loaded without the GSFE control being one.

3. A guarded-storage event is never recognized when all 64 bits of the guarded-storage selection mask (GSSM) are zero. The program can ensure that guarded-storage events are not recognized by either (a) not loading the guarded-storage controls, in which case the GSSM will contain its reset state of zeros, or (b) loading zeros into the GSSM.

LOAD GUARDED STORAGE CONTROLS

LGSC $R_1,D_2(X_2,B_2)$ [RXY-a]



The contents of the guarded-storage control block (GSCB) at the second-operand address are loaded into the three guarded-storage registers. The format of the guarded-storage control block is shown in Figure 4-19 on page 4-66. The R_1 field of the instruction is reserved and should contain zero; otherwise, the program may not operate compatibly in the future.

Access exceptions are recognized for all 32 bytes of the GSCB.

The section "Guarded-Storage-Designation (GSD) Register" on page 4-65 describes the valid values of the GSD register. If either the GLS or GSC fields of the GSD register being loaded contain invalid values, or if reserved bit positions of the register do not contain zeros the results are unpredictable. If the second operand contains either (a) invalid GLS or GSC values, or (b) nonzero values in the reserved bit positions, then it is model dependent whether the CPU replaces the invalid or nonzero values with corrected values. Furthermore, it is unpredictable whether such corrected values are subsequently stored by STORE GUARDED STORAGE CONTROLS.

Special Conditions

A special-operation exception is recognized and the operation is suppressed when the guarded-storage-facility-enablement control, bit 59 of control register 2, is zero.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, second operand)
- Operation (if the guarded-storage facility is not installed)
- Special operation
- Transaction constraint

Programming Notes:

1. If the GSC field of the GSD register contains an invalid value, guarded-storage events may not occur or erroneous guarded-storage events may be detected.

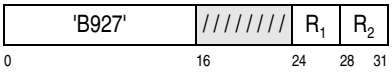
2. If the GLS field of the GSD register contains an invalid value, the intermediate result used by LLGFSG may be formed from an unpredictable

range of bits in the second operand, shifted by an unpredictable number of bits.

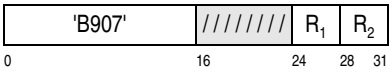
LOAD HALFWORD

Register-and-register formats:

LHR R_1,R_2 [RRE]

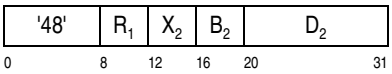


LGHR R_1,R_2 [RRE]

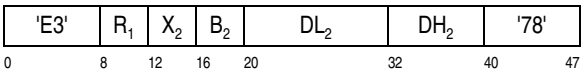


Register-and-storage formats:

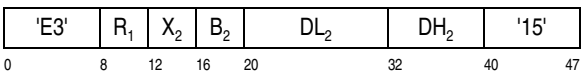
LH $R_1,D_2(X_2,B_2)$ [RX-a]



LHY $R_1,D_2(X_2,B_2)$ [RXY-a]

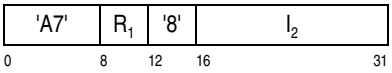


LGH $R_1,D_2(X_2,B_2)$ [RXY-a]

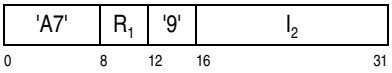


LOAD HALFWORD IMMEDIATE

LHI R_1,I_2 [RI-a]

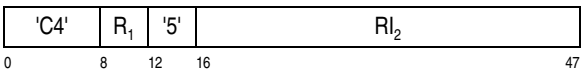


LGHI R_1,I_2 [RI-a]

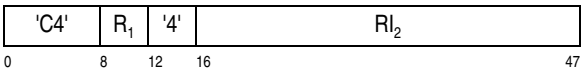


LOAD HALFWORD RELATIVE LONG

LHRL R_1,RI_2 [RIL-b]



LGHRL R_1,RI_2 [RIL-b]



The second operand is sign extended and placed at the first-operand location. The second operand is two bytes in length and is treated as a 16-bit signed binary integer. For LOAD HALFWORD (LH, LHY, and LGH) and LOAD HALFWORD RELATIVE LONG, the second operand is in storage. For LOAD HALFWORD (LHI and LGHI), the second operand is the immediate field of the instruction. For LOAD HALFWORD (LHR and LGHR), the second operand is in bits 48-63 of general register R_2.

For LOAD HALFWORD (LH, LHR, and LHY), LOAD HALFWORD IMMEDIATE (LHI), and LOAD HALFWORD RELATIVE LONG (LHRL), the first operand is treated as a 32-bit signed binary integer. For LOAD HALFWORD (LGH and LGHR), LOAD HALFWORD IMMEDIATE (LGHI), and LOAD HALFWORD RELATIVE LONG (LGHRL), the first operand is treated as a 64-bit signed binary integer.

The displacement for LH is treated as a 12-bit unsigned binary integer. The displacement for LHY and LGH is treated as a 20-bit signed binary integer.

For LOAD HALFWORD RELATIVE LONG, the contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the address of the second operand in storage. When DAT is on, the second operand is accessed using the same addressing-space mode as that used to access the instruction. When DAT is off, the second operand is accessed using a real address.

Condition Code: The code remains unchanged.

Program Exceptions:

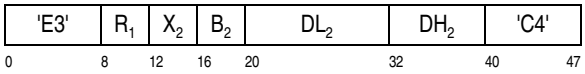
- Access (fetch, operand 2 of LH, LHY, LGH, LHRL and LGHRL)
- Operation (LHY, if the long-displacement facility is not installed; LHR and LGHR, if the extended-immediate facility is not installed; LHRL and LGHRL, if the general-instructions-extension facility is not installed)

Programming Notes:

1. An example of the use of the LOAD HALFWORD instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. For LOAD HALFWORD RELATIVE LONG, the second operand is necessarily aligned on an integral boundary corresponding to the operand’s size.
3. When LOAD HALFWORD RELATIVE LONG is the target of an execute-type instruction, the second-operand address is relative to the target address.

LOAD HALFWORD HIGH

LHH $R_1,D_2(X_2,B_2)$ [RXY-a]



The second operand is sign extended and placed at the first-operand location. The second operand is two bytes in length and is treated as an 16-bit signed binary integer.

The first operand is treated as a 32-bit signed binary integer in bits 0-31 of general register R_1; bits 32-63 of the register are unchanged.

The displacement is treated as a 20-bit signed binary integer.

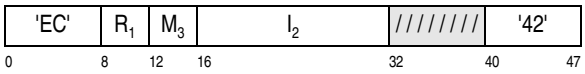
Condition Code: The code remains unchanged.

Program Exceptions:

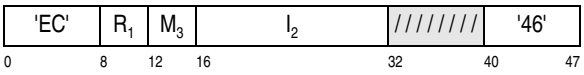
- Access (fetch, operand 2)
- Operation (if the high-word facility is not installed)

LOAD HALFWORD IMMEDIATE ON CONDITION

LOCHI R_1,I_2,M_3 [RIE-g]

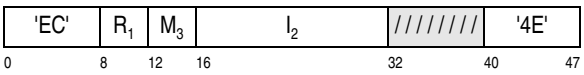


LOCGHI R_1,I_2,M_3 [RIE-g]



LOAD HALFWORD HIGH IMMEDIATE ON CONDITION

LOCHHI R_1,I_2,M_3 [RIE-g]



The second operand is sign extended and placed at the first-operand location if the condition code has one of the values specified by M_3; otherwise, the first operand remains unchanged.

For LOAD HALFWORD IMMEDIATE ON CONDITION (LOCHI), the first operand is treated as a 32-bit signed binary integer in bits 32-63 of general register R_1, and bits 0-31 of the register are unchanged. For LOAD HALFWORD HIGH IMMEDIATE ON CONDITION, the first operand is treated as a 32-bit signed binary integer in bits 0-31 of general register R_1, and bits 32-63 of the register are unchanged. For LOAD HALFWORD IMMEDIATE ON CONDITION (LOCGHI), the first operand is treated as a 64-bit signed binary integer in bits 0-63 of general register R_1.

The second operand is the two byte I_2 field of the instruction and is treated as a 16-bit signed binary integer.

The M_3 field is used as a four-bit mask. The four condition codes (0, 1, 2, and 3) correspond, left to right, with the four bits of the mask, as follows:

Condition Code	0	1	2	3
Instruction Bit Number of Mask	12	13	14	15
Mask Position Value	8	4	2	1

The current condition code is used to select the corresponding mask bit. If the mask bit selected by the condition code is one, the load is performed. If the mask bit selected is zero, the load is not performed.

Condition Code: The code remains unchanged.

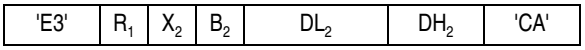
Program Exceptions:

- Operation (if the load/store-on-condition facility 2 is not installed)

Programming Notes: See programming note 4 on page 7-285 for details on extended mnemonics for the instructions of the load/store-on-condition facilities.

LOAD HIGH

LFH $R_1,D_2(X_2,B_2)$ [RXY-a]



The second operand is placed unchanged at the first-operand location. The second operand is 32 bits, and the first operand is in bits 0-31 of general register R_1; bits 32-63 of the register are unchanged.

The displacement is treated as a 20-bit signed binary integer.

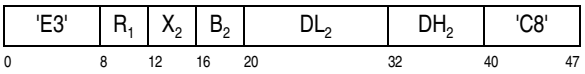
Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2)
- Operation (if the high-word facility is not installed)

LOAD HIGH AND TRAP

LFHAT $R_1,D_2(X_2,B_2)$ [RXY-a]



The second operand is treated as a 32-bit signed integer and placed unchanged in bit positions 0-31 of general register R_1, and bit positions 32-63 remain unchanged. If all zeros are placed at the first operand location, a compare-and-trap-instruction data exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

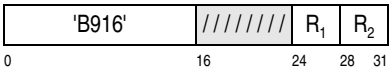
- Data
- Access (fetch, operand 2)
- Operation (if the load-and-trap facility is not installed)

Programming Note: Possible uses of LOAD HIGH AND TRAP include checking for pointers containing zero (null pointer).

LOAD LOGICAL

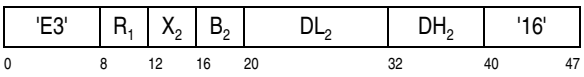
Register-and-register format:

LLGFR R_1,R_2 [RRE]



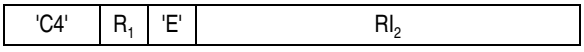
Register-and-storage format:

LLGF $R_1,D_2(X_2,B_2)$ [RXY-a]



LOAD LOGICAL RELATIVE LONG

LLGFRL R_1,RI_2 [RIL-b]



The four-byte second operand is placed in bit positions 32-63 of general register R_1, and zeros are placed in bit positions 0-31 of general register R_1.

For LOAD LOGICAL (LLGFR), the second operand is in bit positions 32-63 of general register R_2.

For LOAD LOGICAL RELATIVE LONG, the contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the address of the second operand in storage. When DAT is on, the second operand is accessed using the same addressing-space mode as that used to access the instruction. When DAT is off, the second operand is accessed using a real address.

For LOAD LOGICAL RELATIVE LONG, the second operand must be aligned on a word boundary; otherwise, a specification exception is recognized.

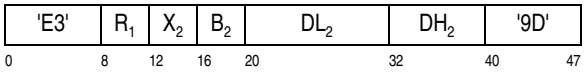
Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of LLGF and LLGFRL only)
- Operation (LLGFRL, if the general-instructions-extension facility is not installed)
- Specification (LLGFRL only)

LOAD LOGICAL AND TRAP

LLGFAT $R_1,D_2(X_2,B_2)$ [RXY-a]



The four-byte second operand is placed unchanged in bit positions 32-63 of general register R_1, and zeros are placed in bit positions 0-31. If all zeros are placed at the first operand location, a compare-and-trap-instruction data exception is recognized.

Condition Code: The code remains unchanged.

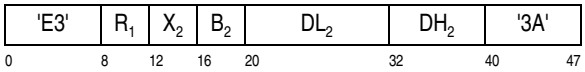
Program Exceptions:

- Data
- Access (fetch, operand 2)
- Operation (if the load-and-trap facility is not installed)

Programming Note: Possible uses of LOAD LOGICAL AND TRAP include checking for pointers containing zero (null pointer).

LOAD LOGICAL AND ZERO RIGHTMOST BYTE

LLZRGF $R_1,D_2(X_2,B_2)$ [RXY-a]



The four-byte second operand, with the rightmost byte set to zero, is placed in bit positions 32-63 of general register R_1, and zeros are placed in bit positions 0-31 of the register.

The displacement is treated as a 20-bit signed binary integer.

It is unpredictable whether an access exception is recognized for the rightmost byte of the second operand.

Condition Code: The code remains unchanged.

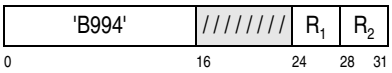
Program Exceptions:

- Access (fetch, operand 2)
- Operation (if the load-and-zero-rightmost-byte facility is not installed)

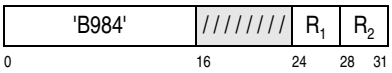
LOAD LOGICAL CHARACTER

Register-and-register formats:

LLCR R_1,R_2 [RRE]

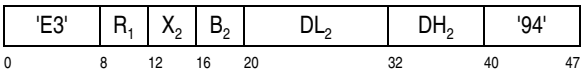


LLGCR R_1,R_2 [RRE]

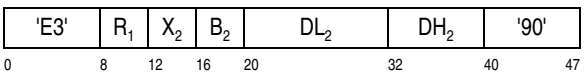


Register-and-storage formats:

LLC $R_1,D_2(X_2,B_2)$ [RXY-a]



LLGC $R_1,D_2(X_2,B_2)$ [RXY-a]



The one-byte second operand is placed in bit positions 56-63 of general register R_1. For LOAD LOGICAL CHARACTER (LLGC, LLGCR), zeros are placed in bit positions 0-55 of general register R_1. For LOAD LOGICAL CHARACTER (LLC, LLCR), zeros are placed in bit positions 32-55 of general register R_1; bit positions 0-31 of general register R_1 are unchanged.

For LOAD LOGICAL CHARACTER (LLC, LLGC), the second operand is in storage. For LOAD LOGICAL CHARACTER (LLCR, LLGCR), the second operand is in bits 56-63 of general register R_2.

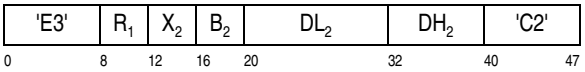
Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of LLC and LLGC)
- Operation (LLC, LLCR, and LLGCR, if the extended-immediate facility is not installed)

LOAD LOGICAL CHARACTER HIGH

LLCH $R_1,D_2(X_2,B_2)$ [RXY-a]



The one-byte second operand is placed in bit positions 24-31 of general register R_1, and zeros are placed in bit positions 0-23 of general register R_1; bit positions 32-63 of general register R_1 are unchanged.

The displacement is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

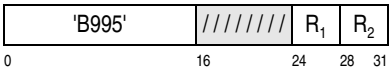
Program Exceptions:

- Access (fetch, operand 2)
- Operation (if the high-word facility is not installed)

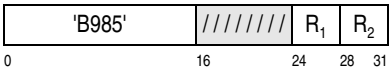
LOAD LOGICAL HALFWORD

Register-and-register formats:

LLHR R_1,R_2 [RRE]

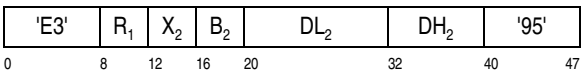


LLGHR R_1,R_2 [RRE]

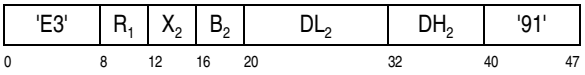


Register-and-storage formats:

LLH $R_1,D_2(X_2,B_2)$ [RXY-a]

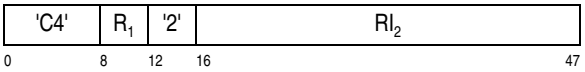


LLGH $R_1,D_2(X_2,B_2)$ [RXY-a]

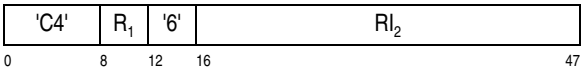


LOAD LOGICAL HALFWORD RELATIVE LONG

LLHRL R_1,RI_2 [RIL-b]



LLGHRL R_1,RI_2 [RIL-b]



The two-byte second operand is placed in bit positions 48-63 of general register R_1. For LOAD LOGICAL HALFWORD (LLGH, LLGHR) and LOAD LOGICAL HALFWORD RELATIVE LONG (LLGHRL), zeros are placed in bit positions 0-47 of general register R_1. For LOAD LOGICAL HALFWORD (LLH, LLHR) and LOAD LOGICAL HALFWORD RELATIVE LONG (LLHRL), zeros are placed in bit positions 32-47 of general register R_1; bit positions bits 0-31 of general register R_1 are unchanged.

For LOAD LOGICAL HALFWORD (LLGH, LLH) and LOAD LOGICAL HALFWORD RELATIVE LONG, the second operand is in storage. For LOAD LOGICAL HALFWORD (LLGHR, LLHR), the second operand is in bits 48-63 of general register R_2.

For LOAD LOGICAL HALFWORD RELATIVE LONG, the contents of the RI_2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the address of the second operand in storage. When DAT is on, the second operand is accessed using the same addressing-space mode as that used to access the instruction. When DAT is off, the second operand is accessed using a real address.

Condition Code: The code remains unchanged.

Program Exceptions:

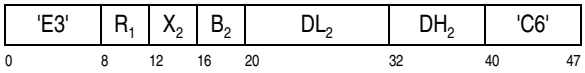
- Access (fetch, operand 2 of LLGH, LLH, LLHRL, and LLGHRL)
- Operation (LLGHR, LLH, and LLHR, if the extended-immediate facility is not installed; LLHRL and LLGHRL, if the general-instructions-extension facility is not installed)

Programming Notes:

1. For LOAD LOGICAL HALFWORD RELATIVE LONG, the second operand is necessarily aligned on an integral boundary corresponding to the operand's size.
2. When LOAD LOGICAL HALFWORD RELATIVE LONG is the target of an execute-type instruction, the second-operand address is relative to the target address.

LOAD LOGICAL HALFWORD HIGH

LLHH $R_1,D_2(X_2,B_2)$ [RXY-a]



The two-byte second operand is placed in bit positions 16-31 of general register R_1, and zeros are placed in bit positions 0-15 of general register R_1; bit positions 32-63 of general register R_1 are unchanged.

The displacement is treated as a 20-bit signed binary integer.

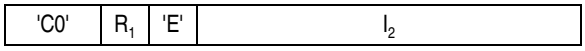
Condition Code: The code remains unchanged.

Program Exceptions:

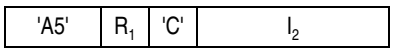
- Access (fetch, operand 2)
- Operation (if the high-word facility is not installed)

LOAD LOGICAL IMMEDIATE

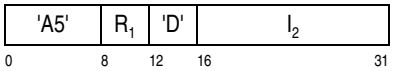
LLIHF R_1,I_2 [RIL-a]



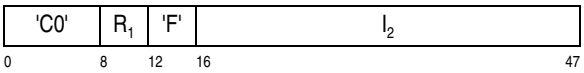
LLIHH R_1,I_2 [RI-a]



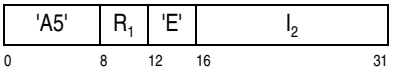
LLIHL R_1,I_2 [RI-a]



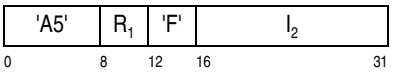
LLILF R_1,I_2 [RIL-a]



LLILH R_1,I_2 [RI-a]



LLILL R_1,I_2 [RI-a]



The second operand is placed in bit positions of the first operand. The remainder of the first operand is set to zeros.

For each instruction, the bit positions of the first operand that are loaded with the second operand are as follows:

Instruction	Bit Positions Loaded
LLIHF	0-31
LLIHH	0-15
LLIHL	16-31
LLILF	32-63
LLILH	32-47
LLILL	48-63

Condition Code: The code remains unchanged.

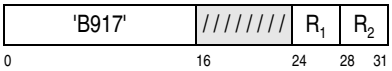
Program Exceptions:

- Operation (LLIHF and LLILF, if the extended-immediate facility is not installed)

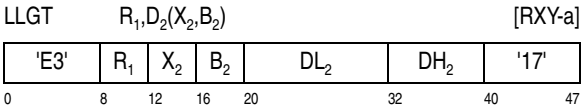
LOAD LOGICAL THIRTY ONE BITS

Register-and-register format:

LLGTR R_1,R_2 [RRE]



Register-and-storage format:



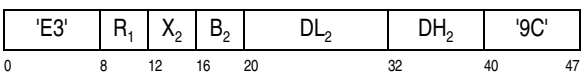
For LLGTR, bits 33-63 of general register R_2, with 33 zeros appended on the left, are placed in general register R_1. For LLGT, bits 1-31 of the four bytes at the second-operand location, with 33 zeros appended on the left, are placed in general register R_1.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of LLGT only)

LOAD LOGICAL THIRTY ONE BITS AND TRAP



Bits 1-31 of the four bytes at the second operand location are placed unchanged in bit positions 33-63 of general register R_1, and zeros are placed in bit positions 0-32. If all zeros are placed at the first operand location, a compare-and-trap-instruction data exception is recognized.

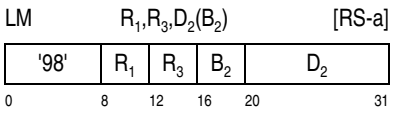
Condition Code: The code remains unchanged.

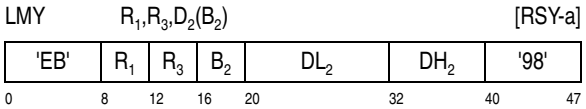
Program Exceptions:

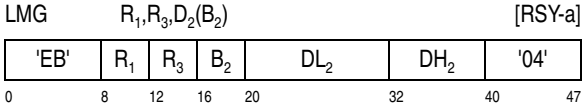
- Data
- Access (fetch, operand 2)
- Operation (if the load-and-trap facility is not installed)

Programming Note: Possible uses of LOAD LOGICAL THIRTY ONE BITS AND TRAP include checking for pointers containing zero (null pointer).

LOAD MULTIPLE







Bit positions of the set of general registers starting with general register R_1 and ending with general register R_3 are loaded from storage beginning at the location designated by the second-operand address and continuing through as many locations as needed.

For LOAD MULTIPLE (LM, LMY), bit positions 32-63 of the general registers are loaded from successive four-byte fields beginning at the second-operand address, and bits 0-31 of the registers remain unchanged. For LOAD MULTIPLE (LMG), bit positions 0-63 of the general registers are loaded from successive eight-byte fields beginning at the second-operand address.

The general registers are loaded in the ascending order of their register numbers, starting with general register R_1 and continuing up to and including general register R_3, with general register 0 following general register 15.

The displacement for LM is treated as a 12-bit unsigned binary integer. The displacement for LMY and LMG is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

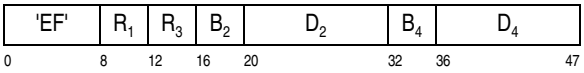
- Access (fetch, operand 2)
- Operation (LMY, if the long-displacement facility is not installed)

Programming Note: All combinations of register numbers specified by R_1 and R_3 are valid. When the register numbers are equal, only four bytes, for LM or

LMY or eight bytes, for LMG, are transmitted. When the number specified by R_3 is less than the number specified by R_1, the register numbers wrap around from 15 to 0.

LOAD MULTIPLE DISJOINT

LMD $R_1,R_3,D_2(B_2),D_4(B_4)$ [SS-e]



Bit positions 0-31 of the set of general registers starting with general register R_1 and ending with general register R_3 are loaded from storage beginning at the location designated by the second-operand address and continuing through as many locations as needed. Bit positions 32-63 of the same registers are similarly loaded from storage beginning at the location designated by the fourth-operand address.

The general registers are loaded in the ascending order of their register numbers, starting with general register R_1 and continuing up to and including general register R_3, with general register 0 following general register 15.

Condition Code: The code remains unchanged.

Program Exceptions:

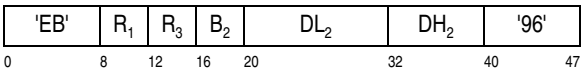
- Access (fetch, operands 2 and 4)
- Transaction constraint

Programming Notes:

1. All combinations of register numbers specified by R_1 and R_3 are valid. When the register numbers are equal, only eight bytes are transmitted. When the number specified by R_3 is less than the number specified by R_1, the register numbers wrap around from 15 to 0.
2. The second-operand and fourth-operand addresses are computed before the contents of any register are changed.
3. The combination of a LOAD MULTIPLE instruction and a LOAD MULTIPLE HIGH instruction provides equal or better performance than a LOAD MULTIPLE DISJOINT instruction for the same register range. LOAD MULTIPLE DISJOINT is for use when the second or fourth operand must be addressed by means of one of the registers loaded.

LOAD MULTIPLE HIGH

LMH $R_1,R_3,D_2(B_2)$ [RSY-a]



The high-order halves, bit positions 0-31, of the set of general registers starting with general register R_1 and ending with general register R_3 are loaded from storage beginning at the location designated by the second-operand address and continuing through as many locations as needed, that is, bit positions 0-31 are loaded from successive four-byte fields beginning at the second-operand address. Bits 32-63 of the registers remain unchanged.

The general registers are loaded in the ascending order of their register numbers, starting with general register R_1 and continuing up to and including general register R_3, with general register 0 following general register 15.

Condition Code: The code remains unchanged.

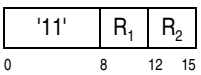
Program Exceptions:

- Access (fetch, operand 2)

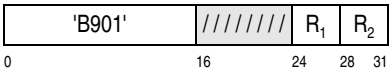
Programming Note: All combinations of register numbers specified by R_1 and R_3 are valid. When the register numbers are equal, only four bytes are transmitted. When the number specified by R_3 is less than the number specified by R_1, the register numbers wrap around from 15 to 0.

LOAD NEGATIVE

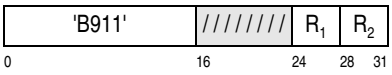
LNR R_1,R_2 [RR]



LNGR R_1,R_2 [RRE]



LNGFR R_1,R_2 [RRE]



The two's complement of the absolute value of the second operand is placed at the first-operand location. For LOAD NEGATIVE (LNR), the second operand and result are treated as 32-bit signed binary integers, and, for LOAD NEGATIVE (LNGR), they are treated as 64-bit signed binary integers. For LOAD NEGATIVE (LNGFR), the second operand is treated as a 32-bit signed binary integer, and the result is treated as a 64-bit signed binary integer.

Resulting Condition Code:

0 Result zero
1 Result less than zero
2 --
3 --

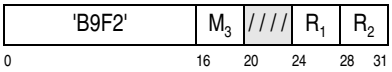
Program Exceptions: None.

Programming Note: The operation complements positive numbers; negative numbers remain unchanged. The number zero remains unchanged.

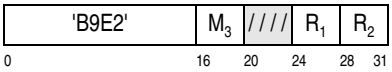
LOAD ON CONDITION

Register-and-register formats:

LOCR R_1,R_2,M_3 [RRF-c]

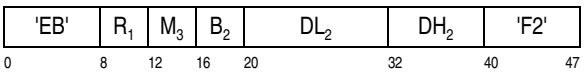


LOCGR R_1,R_2,M_3 [RRF-c]

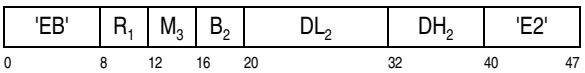


Register-and-storage formats:

LOC $R_1,D_2(B_2),M_3$ [RSY-b]

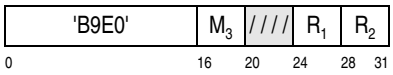


LOCG $R_1,D_2(B_2),M_3$ [RSY-b]

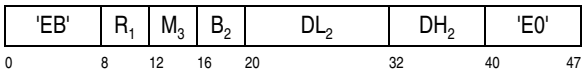


LOAD HIGH ON CONDITION

LOCFHR R_1,R_2,M_3 [RRF-c]



LOCFH $R_1,D_2(B_2),M_3$ [RSY-b]



The second operand is placed unchanged at the first-operand location if the condition code has one of the values specified by M_3; otherwise, the first operand remains unchanged.

For LOC, LOCFH, LOCFHR, and LOCR, the first and second operands are 32 bits, and for LOCG and LOCGR, the first and second operands are 64 bits.

For LOC and LOCR, the first operand is in bits 32-63 of general register R_1, and bits 0-31 of the register are unchanged. For LOCFH and LOCFHR, the first operand is in bits 0-31 of general register R_1, and bits 32-63 of the register are unchanged.

For LOCR, the second operand is in bits 32-63 of general register R_2, and bits 0-31 of the register are ignored. For LOCFHR, the second operand is in bits 0-31 of general register R_2, and bits 32-63 of the register are ignored.

The M_3 field is used as a four-bit mask. The four condition codes (0, 1, 2, and 3) correspond, left to right, with the four bits of the mask, as follows:

Condition Code	0	1	2	3
Mask Position Value	8	4	2	1

The current condition code is used to select the corresponding mask bit. If the mask bit selected by the condition code is one, the load is performed. If the mask bit selected is zero, the load is not performed.

The displacement for LOC, LOCFH, and LOCG is treated as a 20-bit signed binary integer.

For LOC, LOCFH, and LOCG, when the condition specified by the M_3 field is not met (that is, the load operation is not performed), it is model dependent whether an access exception or a PER zero-address

detection event is recognized for the second operand.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of LOC, LOCFH, and LOCG)
- Operation (LOC, LOCG, LOCGR, LOCR, if the load/store-on-condition facility 1 is not installed; LOCFH, LOCFHR, if the load/store-on-condition facility 2 is not installed)

Programming Notes:

1. When the M_3 field contain all zeros and no exception condition exists, the instruction acts as a NOP. When the M_3 field contains all ones and no exception condition exists, the load operation is always performed. However, these are not the preferred means of implementing a NOP or unconditional load, respectively.

2. For LOC, LOCFH, and LOCG, when the condition specified by the M_3 field is not met, it is model dependent whether the second operand is brought into the cache.

3. LOAD ON CONDITION provides a function similar to that of a separate BRANCH ON CONDITION instruction followed by a LOAD instruction, except that LOAD ON CONDITION does not provide an index register. For example, the following two instruction sequences are equivalent.

```
LOCG    15,256(7),8          BC    7,SKIP
                             LG    15,256(0,7)
                       SKIP  DS    0H
```

   On models that implement predictive branching, the combination of the BRANCH ON CONDITION and LOAD instructions may perform somewhat better than the LOAD ON CONDITION instruction when the CPU is able to successfully predict the branch condition. However, on models where the CPU is not able to successfully predict the branch condition, such as when the condition is more random, the LOAD ON CONDITION instruction may provide significant performance improvement.

4. The High-Level Assembler (HLASM) provides the following extended-mnemonic suffixes for the load/store-on-condition facilities' instructions in place of the M_3 field.

Suffix	Meaning	Effective M_3 Value
E	Equal	B'1000'
L	Low	B'0100'
H	High	B'0010'
NE	Not equal	B'0111'
NL	Not low	B'1011'
NH	Not high	B'1101'

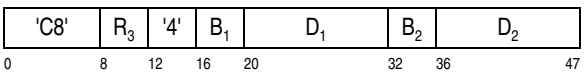
   Later versions of HLASM provide the following extended-mnemonic suffixes for the load/store-on-condition facilities' instructions in place of the M_3 field.

Suffix	Meaning	Effective M_3 Value
Z	Zero	B'1000'
M	Minus or mixed	B'0100'
P	Plus	B'0010'
O	Overflow or ones	B'0001'
NZ	Not zero	B'0111'
NM	Not minus or not mixed	B'1011'
NP	Not plus	B'1101'
NO	Not overflow or not ones	B'1110'

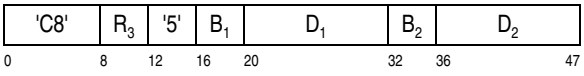
   When the extended mnemonic is coded, the M_3 field must be omitted.

LOAD PAIR DISJOINT

LPD $R_3,D_1(B_1),D_2(B_2)$ [SSF]



LPDG $R_3,D_1(B_1),D_2(B_2)$ [SSF]



General register R_3 designates the even numbered register of an even/odd register pair.

The first operand is placed unchanged into the even-numbered register of the third operand, and the second operand is placed unchanged into odd-numbered register of the third operand. The condition code indicates whether the first and second oper-

ands appear to be fetched by means of block-concurrent interlocked fetch.

For LPD, the first and second operands are words in storage, and the third operand is in bits 32-63 of general registers R_3 and $R_3 + 1$; bits 0-31 of the registers are unchanged. For LPDG, the first and second operands are doublewords in storage, and the third operand is in bits 0-63 of general registers R_3 and $R_3 + 1$.

When, as observed by other CPUs, the first and second operands appear to be fetched by means of block-concurrent interlocked fetch, condition code 0 is set. When the first and second operands do not appear to be fetched by means of block-concurrent interlocked update, condition code 3 is set. The third operand is loaded regardless of the condition code.

The displacement of the first and second operands is treated as a 12-bit unsigned binary integer.

The first and second operands of LPD must be designated on a word boundary. The first and second operands of LPDG must be designated on a doubleword boundary. General register R_3 must designate the even numbered register. Otherwise, a specification exception is recognized.

Resulting Condition Code:

0 Register pair loaded by means of interlocked fetch
1 --
2 --
3 Register pair not loaded by means of interlocked fetch

Program Exceptions:

- Access (fetch, operands 1 and 2)
- Operation (if the interlocked-access facility 1 is not installed)
- Specification
- Transaction constraint

Programming Notes:

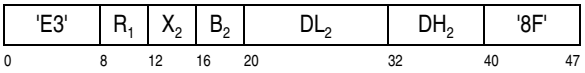
1. The setting of the condition code is dependent upon storage accesses by other CPUs in the configuration.

2. When the resulting condition code is 3, the program may branch back to re-execute the LOAD PAIR DISJOINT instruction. However, after repeated unsuccessful attempts to attain an interlocked fetch, the program should use an alternate means of serializing access to the storage operands. It is recommended that the program re-execute the LOAD PAIR DISJOINT no more than 10 times before branching to the alternate path.

3. The program should be able to accommodate a situation where condition code 0 is never set.

LOAD PAIR FROM QUADWORD

LPQ $R_1,D_2(X_2,B_2)$ [RXY-a]



The quadword second operand is loaded into the first-operand location. The second operand appears to be fetched with quadword concurrency as observed by other CPUs. The left doubleword of the quadword is loaded into general register R_1, and the right doubleword is loaded into general register $R_1 + 1$.

The R_1 field designates an even-odd pair of general registers and must designate an even-numbered register. The second operand must be designated on a quadword boundary. Otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2)
- Specification
- Transaction constraint

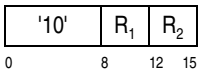
Programming Notes:

1. The LOAD MULTIPLE (LM or LMG) instruction does not necessarily provide quadword-concurrent access.

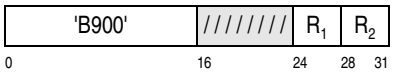
2. The performance of LOAD PAIR FROM QUADWORD on some models may be significantly slower than that of LOAD MULTIPLE (LMG). Unless quadword consistency is required, LMG should be used instead of LPQ.

LOAD POSITIVE

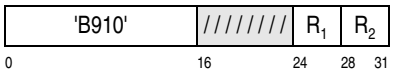
LPR R_1,R_2 [RR]



LPGR R_1,R_2 [RRE]



LPGFR R_1,R_2 [RRE]



The absolute value of the second operand is placed at the first-operand location. For LOAD POSITIVE (LPR), the second operand and result are treated as 32-bit signed binary integers, and, for LOAD POSITIVE (LPGR), they are treated as 64-bit signed binary integers. For LOAD POSITIVE (LPGFR), the second operand is treated as a 32-bit signed binary integer, and the result is treated as a 64-bit signed binary integer.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

Resulting Condition Code:

0 Result zero; no overflow
1 --
2 Result greater than zero; no overflow
3 Overflow

Program Exceptions:

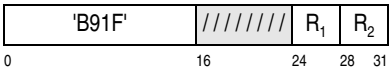
- Fixed-point overflow (LPR and LPGR only)

Programming Note: The operation complements negative numbers; positive numbers and zero remain unchanged. For LPR or LPGR, an overflow condition occurs when the maximum negative 32-bit number or 64-bit number, respectively, is complemented; the number remains unchanged. LPGFR complements the maximum negative 32-bit number without recognizing overflow.

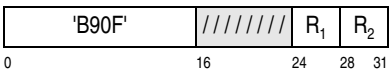
LOAD REVERSED

Register-and-register formats:

LRVR R_1,R_2 [RRE]

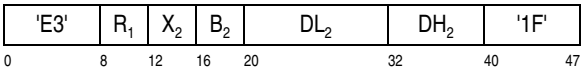


LRVGR R_1,R_2 [RRE]

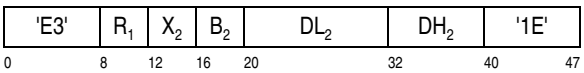


Register-and-storage formats:

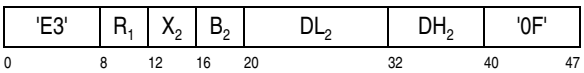
LRVH $R_1,D_2(X_2,B_2)$ [RXY-a]



LRV $R_1,D_2(X_2,B_2)$ [RXY-a]



LRVG $R_1,D_2(X_2,B_2)$ [RXY-a]



The second operand is placed at the first-operand location with the left-to-right sequence of the bytes reversed.

For LOAD REVERSED (LRVH), the second operand is two bytes, the result is placed in bit positions 48-63 of general register R_1, and bits 0-47 of the register remain unchanged.

For LOAD REVERSED (LRVR, LRV), the second operand is four bytes, the result is placed in bit positions 32-63 of general register R_1, and bits 0-31 of the register remain unchanged. For LOAD REVERSED (LRVR), the second operand is in bit positions 32-63 of general register R_2.

For LOAD REVERSED (LRVGR, LRVG), the second operand is eight bytes.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of LRVH, LRV, LRVG only)

Programming Notes:

1. The instruction can be used to convert two, four, or eight bytes from a "little-endian" format to a "big-endian" format, or vice versa. In the big-endian format, the bytes in a left-to-right sequence are in the order most significant to least significant. In the little-endian format, the bytes are in the order least significant to most significant. For example, the bytes ABCD in the big-endian format are DCBA in the little-endian format.

2. LOAD REVERSED (LRVR) can be used with a two-byte value already in a register as shown in the following example. In the example, the two bytes of interest are in bit positions 48-63 of the R1 register.

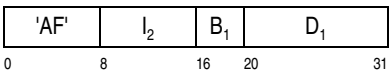
   ```
   LRVR    R1,R1
   SRA     R1,16
   ```

   The LOAD REVERSED instruction places the two bytes of interest in bit positions 32-47 of the register, with the order of the bytes reversed. The SHIFT RIGHT SINGLE (SRA) instruction shifts the two bytes to bit positions 48-63 of the register and extends them on their left, in bit positions 32-47, with their sign bit. The instruction SHIFT RIGHT SINGLE LOGICAL (SRL) should be used, instead, if the two bytes of interest are unsigned.

3. The storage-operand references of LOAD REVERSED may be multiple-access references. (See "Storage-Operand Consistency" on page 5-125.)

MONITOR CALL

MC $D_1(B_1),I_2$ [SI]



Subject to the control of the monitor class and the monitor masks, a monitor event may be recognized. When the enhanced-monitor facility is not installed, a monitor event results in a monitor-event program interruption. When the enhanced-monitor facility is installed, a monitor event results in either a monitor-event program interruption or a monitor-event counting operation.

Bit positions 12-15 of the instruction (bit positions 4-7 of the I_2 field) contain a 4-bit unsigned binary number specifying the monitor class. The monitor-mask bits are in bit positions 48-63 of control register 8, which correspond to monitor classes 0-15, respectively.

When the monitor-mask bit corresponding to the specified monitor class is one, a monitor event occurs. When the monitor-mask bit corresponding to the specified monitor class is zero, no monitor-event occurs, and the instruction is executed as a no-operation.

When the enhanced-monitor facility is installed, the enhanced-monitor-mask bits are in bit positions 16-31 of control register 8, which correspond to monitor classes 0-15 respectively. When a monitor event occurs and either (a) the enhanced-monitor facility is not installed, or (b) the facility is installed but the enhanced-monitor-mask bit corresponding to the monitor class is zero, then a monitor-event program interruption occurs, as described below. When a monitor event occurs and the enhanced-monitor-mask bit corresponding to the monitor class is one, a monitor-event counting operation is performed.

The first-operand address is not used to address data; instead, the address specified by the B_1 and D_1 fields forms the monitor code. Address computation follows the rules of address arithmetic; in the 24-bit addressing mode, bits 0-39 are set to zeros; in the 31-bit addressing mode, bits 0-32 are set to zeros.

Monitor-Event Program Interruption

When a monitor-event program interruption occurs:

- The monitor code formed by the first-operand address is placed in the doubleword at real location 176.
- The contents of the I_2 field are stored at real location 149, with zeros stored at real location 148.
- Bit 9 of the program-interruption code is set to one.

Monitor-Event Counting Operation

The monitor-event counting operation is described in "Monitor-Event Counting" on page 5-109.

Special Conditions

Bit positions 8-11 of the instruction must contain zeros; otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

- Monitor event
- Specification
- Transaction constraint

Programming Notes:

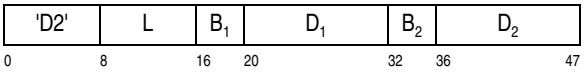
1. The monitor-event program interruption provides the capability for passing control to a monitoring program when selected points are reached in the monitored program. This is accomplished by implanting MONITOR CALL instructions at the desired points in the monitored program. This function may be useful in performing various measurement functions; specifically, tracing information can be generated indicating which programs were executed, counting information can be generated indicating how often particular programs were used, and timing information can be generated indicating the amount of time a particular program required for execution.
2. The monitor masks provide a means of disallowing all monitor events or allowing monitor events for all or selected classes.
3. When used to generate a monitor-event program interruption, the monitor code provides a means of associating descriptive information, in addition to the class number, with each MONITOR CALL.

   Without the use of a base register, up to 4,096 distinct monitor codes can be associated with a monitor event. With the base register designated by a nonzero value in the B_1 field, each monitoring interruption can be identified by a 24-bit, 31-bit, or 64-bit code, depending on the addressing mode.
4. A monitor-event counting operation can never occur in the ESA/390-compatibility mode.

MOVE

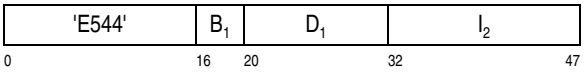
Storage-and-storage format:

MVC $D_1(L,B_1),D_2(B_2)$ [SS-a]

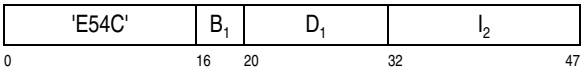


Storage-and-immediate formats:

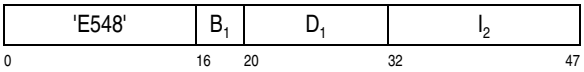
MVHHI $D_1(B_1),I_2$ [SIL]



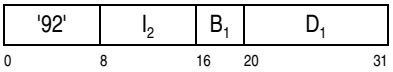
MVHI $D_1(B_1),I_2$ [SIL]



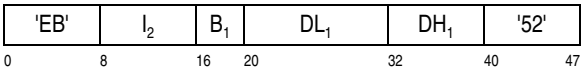
MVGHI $D_1(B_1),I_2$ [SIL]



MVI $D_1(B_1),I_2$ [SI]



MVIY $D_1(B_1),I_2$ [SIY]



The second operand is placed at the first-operand location.

For MOVE (MVC), each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand byte.

For MOVE (MVI, MVIY), the first operand is one byte in length, and only one byte is stored.

For MOVE (MVGHI, MVHHI, and MVHI,), the second operand is treated as a 16-bit signed integer, sign-extended as necessary, and placed in the first-operand location. The first operand is two, four, or eight bytes for MVHHI, MVHI, and MVGHI, respectively.

The displacements for MVGHI, MVHHI, MVHI, MVI and both operands of MVC are treated as 12-bit unsigned binary integers. The displacement for MVIY is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

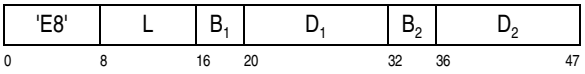
- Access (fetch, operand 2 of MVC; store, operand 1, MVC, MVGHI, MVHHI, MVHI, MVI, and MVIY)
- Operation (MVIY, if the long-displacement facility is not installed; MVGHI, MVHHI, and MVHI, if the general-instructions-extension facility is not installed)
- Transaction constraint (MVC)

Programming Notes:

1. Examples of the use of the MOVE instruction are given in Appendix A, "Number Representation and Instruction-Use Examples."
2. For MOVE (MVC), it is possible to propagate one byte through an entire field by having the first operand start one byte to the right of the second operand.

MOVE INVERSE

MVCIN $D_1(L,B_1),D_2(B_2)$ [SS-a]



The second operand is placed at the first-operand location with the left-to-right sequence of the bytes inverted.

The first-operand address designates the leftmost byte of the first operand. The second-operand address designates the rightmost byte of the second operand. Both operands have the same length.

The result is obtained as if the second operand were processed from right to left and the first operand from left to right. The second operand may wrap around from location 0 to location $2^{24} - 1$ in the 24-bit addressing mode, to location $2^{31} - 1$ in the 31-bit addressing mode, or to location $2^{64} - 1$ in the 64-bit addressing mode. The first operand may wrap around from location $2^{24} - 1$ to location 0 in the 24-bit addressing mode, from location $2^{31} - 1$ to location 0 in the 31-bit addressing mode, or from location $2^{64} - 1$ to location 0 in the 64-bit addressing mode.

When the operands overlap by more than one byte, the contents of the overlapped portion of the result field are unpredictable.

Condition Code: The code remains unchanged.

Program Exceptions:

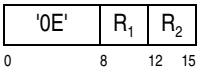
- Access (fetch, operand 2; store, operand 1)
- Transaction constraint

Programming Notes:

1. An example of the use of the MOVE INVERSE instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. The contents of each byte moved remain unchanged.
3. MOVE INVERSE is the only SS-format instruction for which the second-operand address designates the rightmost, instead of the leftmost, byte of the second operand.
4. The storage-operand references for MOVE INVERSE may be multiple-access references. (See "Storage-Operand Consistency" on page 5-125)

MOVE LONG

MVCL R_1,R_2 [RR]



The second operand is placed at the first-operand location, provided overlapping of operand locations would not affect the final contents of the first-operand location. The remaining rightmost byte positions, if any, of the first-operand location are filled with padding bytes.

The R_1 and R_2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and second operand is designated by the contents of

general registers R_1 and R_2, respectively. The number of bytes in the first-operand and second-operand locations is specified by unsigned binary integers in bit positions 40-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively. Bit positions 32-39 of general register $R_2 + 1$ contain the padding byte. The contents of bit positions 0-39 of general register $R_1 + 1$ and of bit positions 0-31 of general register $R_2 + 1$ are ignored.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-301.

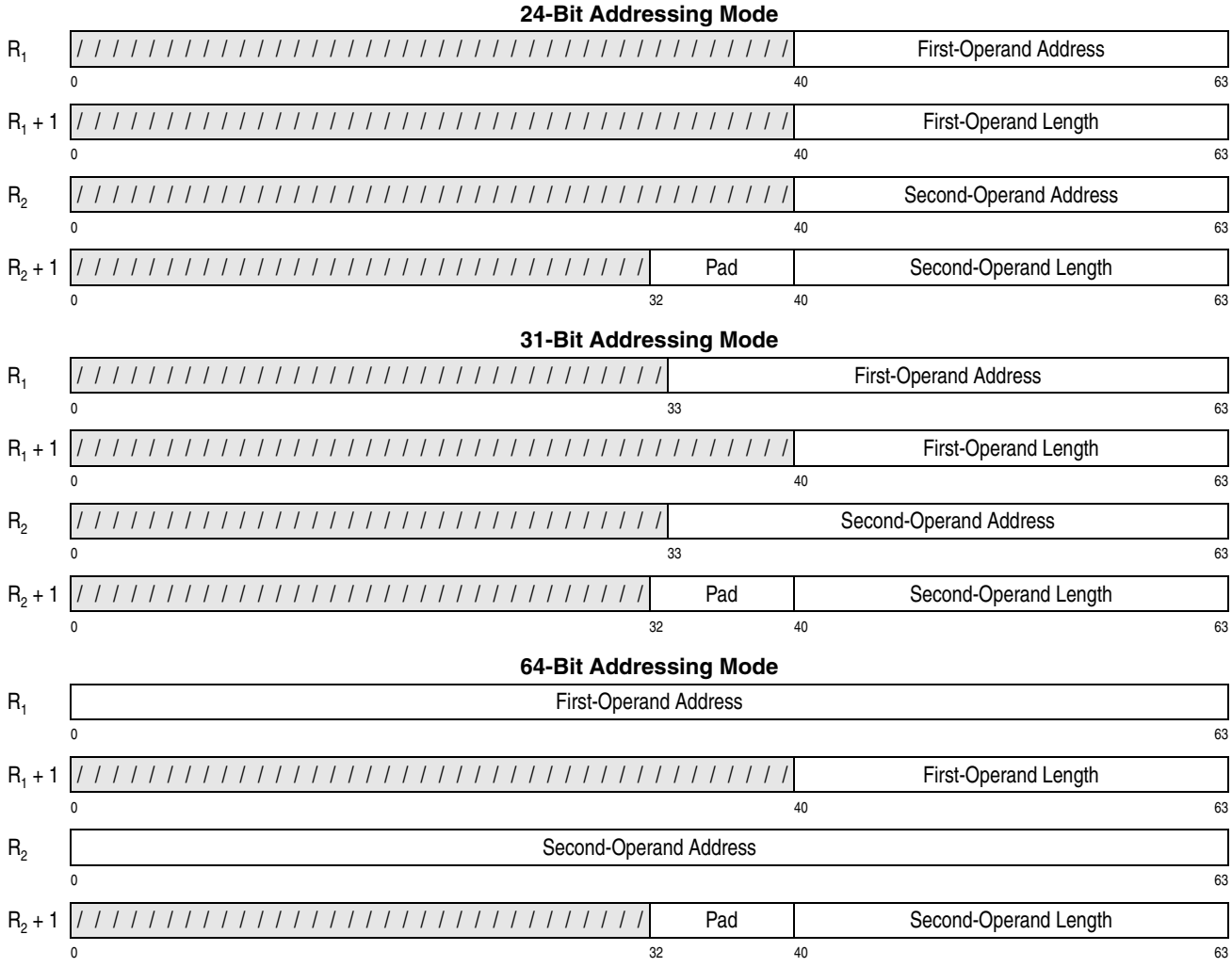


Figure 7-301. Register Contents for MOVE LONG

The result is obtained as if the movement starts at the left end of both fields and proceeds to the right, byte by byte. The operation is ended when the number of bytes specified by bits 40-63 of general register $R_1 + 1$ have been moved into the first-operand location. If the second operand is shorter than the first operand, the remaining rightmost bytes of the first-operand location are filled with the padding byte.

As part of the execution of the instruction, the values of the two length fields are compared for the setting of the condition code, and a check is made for

destructive overlap of the operands. Operands are said to overlap destructively when the first-operand location is used as a source after data has been moved into it, assuming the inspection for overlap is performed by the use of logical operand addresses. When the operands overlap destructively, no movement takes place, and condition code 3 is set.

Operands do not overlap destructively, and movement is performed, if the leftmost byte of the first operand does not coincide with any of the second-operand bytes participating in the operation other than the leftmost byte of the second operand. When an operand wraps around from location $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) to location 0, operand bytes in locations up to and including $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) are considered to be to the left of bytes in locations from 0 up.

In the 24-bit addressing mode, wraparound is from location $2^{24} - 1$ to location 0; in the 31-bit addressing mode, wraparound is from location $2^{31} - 1$ to location 0; in the 64-bit addressing mode, wraparound is from location $2^{64} - 1$ to location 0.

In the access-register mode, the contents of access register R_1 and access register R_2 are compared. If the R_1 or R_2 field is zero, 32 zeros are used rather than the contents of access register 0. If all 32 bits of the compared values are equal, then the destructive overlap test is made. If all 32 bits of the compared values are not equal, destructive overlap is declared not to exist. If, for this case, the operands actually overlap in real storage, it is unpredictable whether the result reflects the overlap condition.

When the length specified by bits 40-63 of general register $R_1 + 1$ is zero, no movement takes place, and condition code 0 or 1 is set to indicate the relative values of the lengths.

The execution of the instruction is interruptible. When an interruption occurs, other than one that follows termination, the lengths in general registers $R_1 + 1$ and $R_2 + 1$ are decremented by the number of bytes moved, and the addresses in general registers R_1 and R_2 are incremented by the same number, so that the instruction, when reexecuted, resumes at the point of interruption. In the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R_1 and R_2 are set to zeros, and the contents of bit positions 0-31 remain unchanged. In any addressing mode, the contents of bit positions 0-39 of general registers $R_1 + 1$ and $R_2 + 1$ remain unchanged; and the condition code is unpredictable. If the operation is interrupted during padding, the length field in general register $R_2 + 1$ is 0, the address in general register R_2 is incremented by the original length in general register $R_2 + 1$, and general registers R_1 and $R_1 + 1$ reflect the extent of the padding operation.

When the first-operand location includes the location of the instruction or of an execute-type instruction, the instruction may be refetched from storage and reinterpreted even in the absence of an interruption during execution. The exact point in the execution at which such a refetch occurs is unpredictable.

Padding byte values of B0 hex and B8 hex may be used during the nonpadding part of the operation by some models, in certain cases, as an indication of whether the movement should be performed bypassing the cache or using the cache, respectively. Thus, a padding byte of B0 hex indicates no intention to reference the destination area after the move, and a padding byte of B8 hex indicates an intention to reference the destination area.

For the nonpadding part of the operation when the padding byte is not B1 hex, accesses to the operands for MOVE LONG may be multiple-access references, and these accesses do not necessarily appear to occur in a left-to-right direction as observed by other CPUs and by the channel subsystem. When the padding byte is B1 hex, accesses to the operands are single-access references, and these accesses appear to occur in a left-to-right direction as observed by other CPUs and by the channel subsystem. During the nonpadding part of the operation, operands appear to be accessed doubleword concurrent as observed by other CPUs, provided that both operands start on doubleword boundaries, are an integral number of doublewords in length, and do not overlap.

As observed by other CPUs and by the channel subsystem, that portion of the first operand which is filled with the padding byte is not necessarily stored into in a left-to-right direction and may appear to be stored into more than once. During the padding operation, stores by other CPUs or by the channel subsystem into that portion of the first operand which is filled with the padding byte may cause unpredictable results.

At the completion of the operation, the length in general register $R_1 + 1$ is decremented by the number of

bytes stored at the first-operand location, and the address in general register R_1 is incremented by the same amount. The length in general register $R_2 + 1$ is decremented by the number of bytes moved out of the second-operand location, and the address in general register R_2 is incremented by the same amount. In the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R_1 and R_2 are set to zeros, even when one or both of the original length values are zeros or when condition code 3 is set. The contents of bit positions 0-31 of the registers remain unchanged. In any addressing mode, the contents of bit positions 0-39 of general registers $R_1 + 1$ and $R_2 + 1$ remain unchanged.

When condition code 3 is set, no exceptions associated with operand access are recognized. When the length of an operand is zero, no access exceptions for that operand are recognized. Similarly, when the second operand is longer than the first operand, access exceptions are not recognized for the part of the second-operand field that is in excess of the first-operand field. For operands longer than 2K bytes, access exceptions are not recognized for locations more than 2K bytes beyond the current location being processed. Access exceptions are not recognized for an operand if the R field associated with that operand is odd. Also, when the R_1 field is odd, PER storage-alteration events are not recognized, and no change bits are set.

Resulting Condition Code:

0 Operand lengths equal; no destructive overlap
1 First-operand length low; no destructive overlap
2 First-operand length high; no destructive overlap
3 No movement performed because of destructive overlap

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Specification
- Transaction constraint

Programming Notes:

1. An example of the use of the MOVE LONG instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. MOVE LONG may be used for clearing storage by setting the padding byte to zero and the second-operand length to zero. However, the stores associated with this clearing may be multiple-access stores and should not be used to clear an area if the possibility exists that another CPU or a channel program will attempt to access and use the area as soon as it appears to be zero. For more details, see “Storage-Operand Consistency” on page 5-125.

3. When the first-operand length is zero, the operation consists in setting the condition code and, in the 24-bit or 31-bit addressing mode, of setting the leftmost bits in bit positions 32-63 of general registers R_1 and R_2 to zero.

4. When the contents of the R_1 and R_2 fields are the same, the contents of the designated registers are incremented or decremented only by the number of bytes moved, not by twice the number of bytes moved. Condition code 0 is set.

5. The following is a detailed description of those cases in which movement takes place, that is, where destructive overlap does not exist.

   In the access-register mode, the contents of the access registers used are called the effective space designations. When the effective space designations are not equal, destructive overlap is declared not to exist and movement occurs. When the effective space designations are the same or when not in the access-register mode, then the following cases apply.

   Depending on whether the second operand wraps around from location $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$, depending on the addressing mode) to location 0, movement takes place in the following cases:

   a. When the second operand does not wrap around, movement is performed if the leftmost byte of the first operand coincides with or is to the left of the leftmost byte of the second operand, *or* if the leftmost byte of the first operand is to the right of the rightmost second-operand byte participating in the operation.

   b. When the second operand wraps around, movement is performed if the leftmost byte of the first operand coincides with or is to the left of the leftmost byte of the second operand, *and* if the leftmost byte of the first operand is to the right of the rightmost second-operand byte participating in the operation.

The rightmost second-operand byte is determined by using the smaller of the first-operand and second-operand lengths.

When the second-operand length is one or zero, destructive overlap cannot exist.

6. Special precautions should be taken if MOVE LONG is made the target of an execute-type instruction. See the programming note concerning interruptible instructions under EXECUTE.

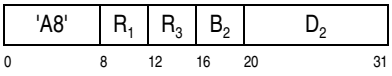
7. Since the execution of MOVE LONG is interruptible, the instruction cannot be used for situations where the program must rely on uninterrupted execution of the instruction. Similarly, the program should normally not let the first operand of MOVE LONG include the location of the instruction or of an execute-type instruction because the new contents of the location may be interpreted for a resumption after an interruption, or the instruction may be refetched without an interruption.

8. Further programming notes concerning interruptible instructions are included in “Interruptible Instructions” in Chapter 5, “Program Execution.”

9. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

MOVE LONG EXTENDED

MVCLE $R_1,R_3,D_2(B_2)$ [RS-a]



All or part of the third operand is placed at the first-operand location. The remaining rightmost byte positions, if any, of the first-operand location are filled with padding bytes. The operation proceeds until the end of the first-operand location is reached or a CPU-determined number of bytes have been placed at the first-operand location, whichever occurs first. The result is indicated in the condition code.

The R_1 and R_3 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and third operand is designated by the contents of general registers R_1 and R_3, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_3 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the entire contents of general registers $R_1 + 1$ and $R_3 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R_1 and R_3 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_3 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The second-operand address is not used to address data; instead, the rightmost eight bits of the second-operand address, bits 56-63, are the padding byte. Bits 0-55 of the second-operand address are ignored.

The contents of the registers and address just described are shown in Figure 7-302.

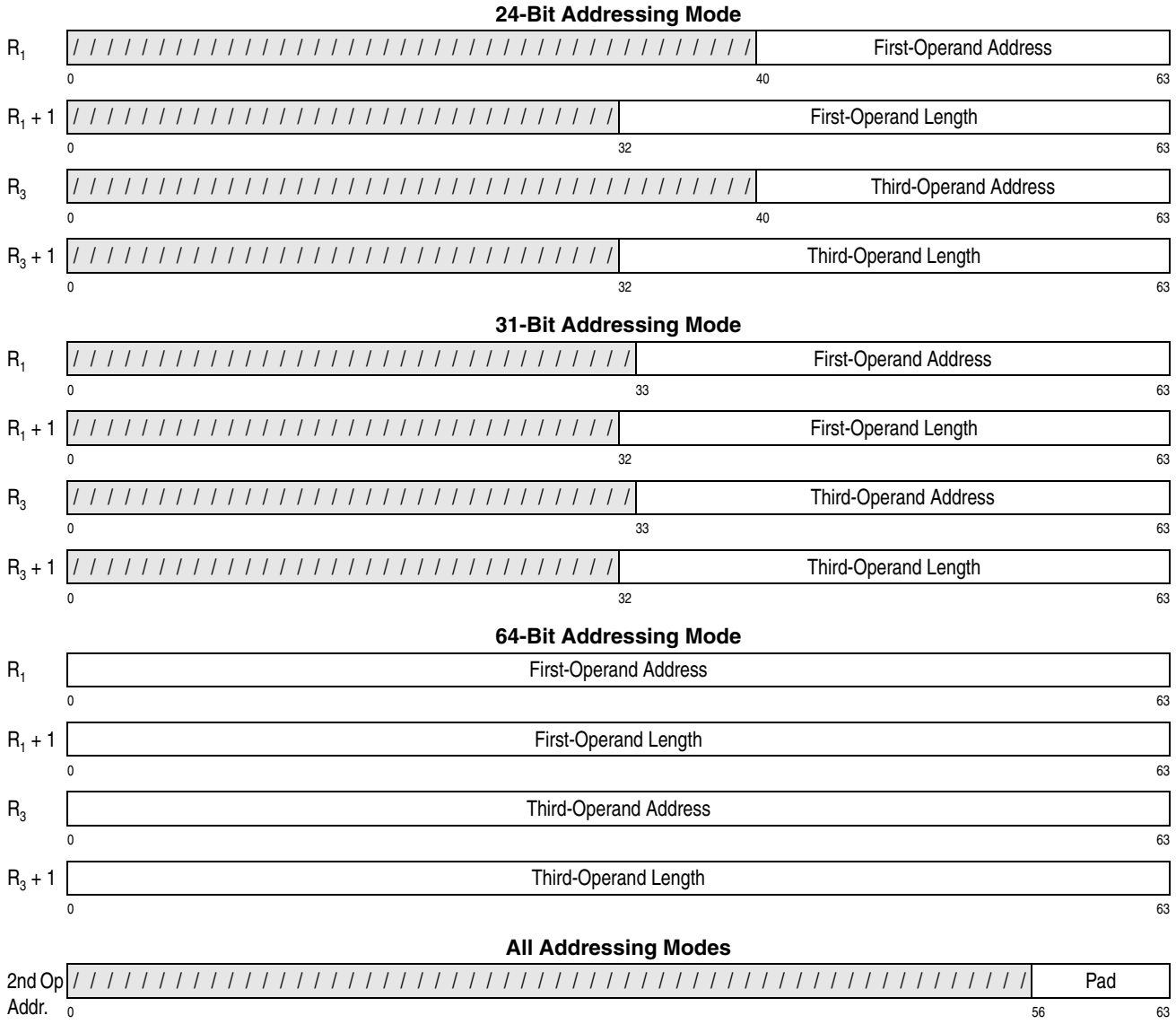


Figure 7-302. Register Contents and Second-Operand Address for MOVE LONG EXTENDED

The result is obtained as if the movement starts at the left end of both fields and proceeds to the right, byte by byte. The operation is ended when the number of bytes specified in general register $R_1 + 1$ have been placed at the first-operand location or when a CPU-determined number of bytes have been placed, whichever occurs first. If the third operand is shorter than the first operand, the remaining rightmost bytes of the first-operand location are filled with the padding byte.

When the operation is completed because the end of the first operand has been reached, the condition code is set to 0 if the two operand lengths are equal, it is set to 1 if the first-operand length is less than the third-operand length, or it is set to 2 if the first-operand length is greater than the third-operand length. When the operation is completed because a CPU-determined number of bytes have been moved without reaching the end of the first operand, condition code 3 is set.

No test is made for destructive overlap, and the results in the first-operand location are unpredictable when destructive overlap exists. Operands are said to overlap destructively when the first-operand location is used as a source after data has been moved into it.

Operands do not overlap destructively if the leftmost byte of the first operand does not coincide with any of the third-operand bytes participating in the operation other than the leftmost byte of the third operand. When an operand wraps around from location $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) to location 0, operand bytes in locations up to and including $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) are considered to be to the left of bytes in locations from 0 up.

In the 24-bit addressing mode, wraparound is from location $2^{24} - 1$ to location 0; in the 31-bit addressing mode, wraparound is from location $2^{31} - 1$ to location 0; and, in the 64-bit addressing mode, wraparound is from location $2^{64} - 1$ to location 0.

When the length specified in general register $R_1 + 1$ is zero, no movement takes place, and condition code 0 or 1 is set to indicate the relative values of the lengths.

Padding byte values of B0 hex and B8 hex may be used during the nonpadding part of the operation by some models, in certain cases, as an indication of whether the movement should be performed bypassing the cache or using the cache, respectively. Thus, a padding byte of B0 hex indicates no intention to reference the destination area after the move, and a padding byte of B8 hex indicates an intention to reference the destination area.

For the nonpadding part of the operation when the padding byte is not B1 hex, accesses to the operands for MOVE LONG EXTENDED may be multiple-access references, and these accesses do not necessarily appear to occur in a left-to-right direction as observed by other CPUs and by the channel subsystem. When the padding byte is B1 hex, accesses to the operands are single-access references, and these accesses appear to occur in a left-to-right direction as observed by other CPUs and by the channel subsystem. During the nonpadding part of the operation, operands appear to be accessed doubleword concurrent as observed by other CPUs, provided that both operands start on doubleword boundaries, are an integral number of doublewords in length, and do not overlap.

As observed by other CPUs and by the channel subsystem, that portion of the first operand which is filled with the padding byte is not necessarily stored into in a left-to-right direction and may appear to be stored into more than once. During the padding operation, stores by other CPUs or by the channel subsystem into that portion of the first operand which is filled with the padding byte may cause unpredictable results.

At the completion of the operation, the length in general register $R_1 + 1$ is decremented by the number of bytes stored at the first-operand location, and the address in general register R_1 is incremented by the same amount. The length in general register $R_3 + 1$ is decremented by the number of bytes moved out of the third-operand location, and the address in general register R_3 is incremented by the same amount.

If the operation is completed because a CPU-determined number of bytes have been moved without reaching the end of the first operand, the lengths in general registers $R_1 + 1$ and $R_3 + 1$ are decremented by the number of bytes moved, and the addresses in general registers R_1 and R_3 are incremented by the same number, so that the instruction, when reexecuted, resumes at the next byte to be moved. If the operation is completed during padding, the length field in general register $R_3 + 1$ is zero, the address in general register R_3 is incremented by the original length in general register $R_3 + 1$, and general registers R_1 and $R_1 + 1$ reflect the extent of the padding operation.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_3, and $R_3 + 1$, always remain unchanged.

The padding byte may be formed from $D_2(B_2)$ multiple times during the execution of the instruction, and the registers designated by R_1 and R_3 may be updated multiple times. Therefore, if B_2 equals R_1, $R_1 + 1$, R_3, or $R_3 + 1$ and is subject to change during the execution of the instruction, the results are unpredictable.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed. The maximum amount is approximately 4K bytes of either operand.

At the completion of the operation in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R_1 and R_3 may be set to zeros or may remain unchanged from their original values, even when one or both of the original length values are zeros.

When the length of an operand is zero, no access exceptions for that operand are recognized. Similarly, when the third operand is longer than the first operand, access exceptions are not recognized for the part of the third-operand field that is in excess of the first-operand field. For operands longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the current location being processed. Access exceptions are not recognized for an operand if the R field associated with that operand is odd. Also, when the R_1 field is odd, PER storage-alteration events are not recognized, and no change bits are set.

Resulting Condition Code:

0 All bytes moved, operand lengths equal
1 All bytes moved, first-operand length low
2 All bytes moved, first-operand length high
3 CPU-determined number of bytes moved without reaching end of first operand

Program Exceptions:

- Access (fetch, operand 3; store, operand 1)
- Specification
- Transaction constraint

Programming Notes:

1. MOVE LONG EXTENDED is intended for use in place of MOVE LONG when the operand lengths are specified as 32-bit or 64-bit binary integers and a test for destructive overlap is not required. MOVE LONG EXTENDED sets condition code 3 in cases in which MOVE LONG would be interrupted.

2. When condition code 3 is set, the program can simply branch back to the instruction to continue the movement. The program need not determine the number of bytes that were moved.

3. The function of not processing more than approximately 4K bytes of either operand is intended to permit software polling of a flag that may be set by a program on another CPU during long operations.

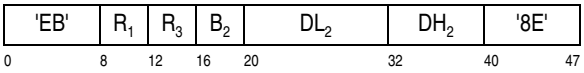
4. MOVE LONG EXTENDED may be used for clearing storage by setting the padding byte to zero and the third-operand length to zero. However, the stores associated with this clearing may be multiple-access stores and should not be used to clear an area if the possibility exists that another CPU or a channel program will attempt to access and use the area as soon as it appears to be zero. For more details, see “Storage-Operand Consistency” on page 5-125.

5. When the contents of the R_1 and R_3 fields are the same, the contents of the designated registers are incremented or decremented only by the number of bytes moved, not by twice the number of bytes moved. The condition code is finally set to 0 after possible settings to 3.

6. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

MOVE LONG UNICODE

MVCLU $R_1,R_3,D_2(B_2)$ [RSY-a]



All or part of the third operand is placed at the first-operand location. The remaining rightmost positions, if any, of the first-operand location are filled with one or more padding bytes, alternating between the even padding byte and the odd padding byte. The operation proceeds until the end of the first-operand location is reached or a CPU-determined number of bytes have been placed at the first-operand location, whichever occurs first. The result is indicated in the condition code.

The R_1 and R_3 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and third operand is designated by the contents of general registers R_1 and R_3, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 32-63 of

general registers $R_1 + 1$ and $R_3 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 0-63 of general registers $R_1 + 1$ and $R_3 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

On some models, the contents of general registers $R_1 + 1$ and $R_3 + 1$ must specify an even number of bytes; otherwise, a specification exception is recognized.

The handling of the addresses in general registers R_1 and R_3 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_3 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of the registers constitute the address.

The second-operand address is not used to address data; instead, the rightmost 16 bits of the second-operand address, bits 48-63, are the two padding bytes; bits 48-55 are the even padding byte, and bits 56-63 are the odd padding byte. Bits 0-47 of the second-operand address are ignored.

The contents of the registers and address just described are shown in Figure 7-303.

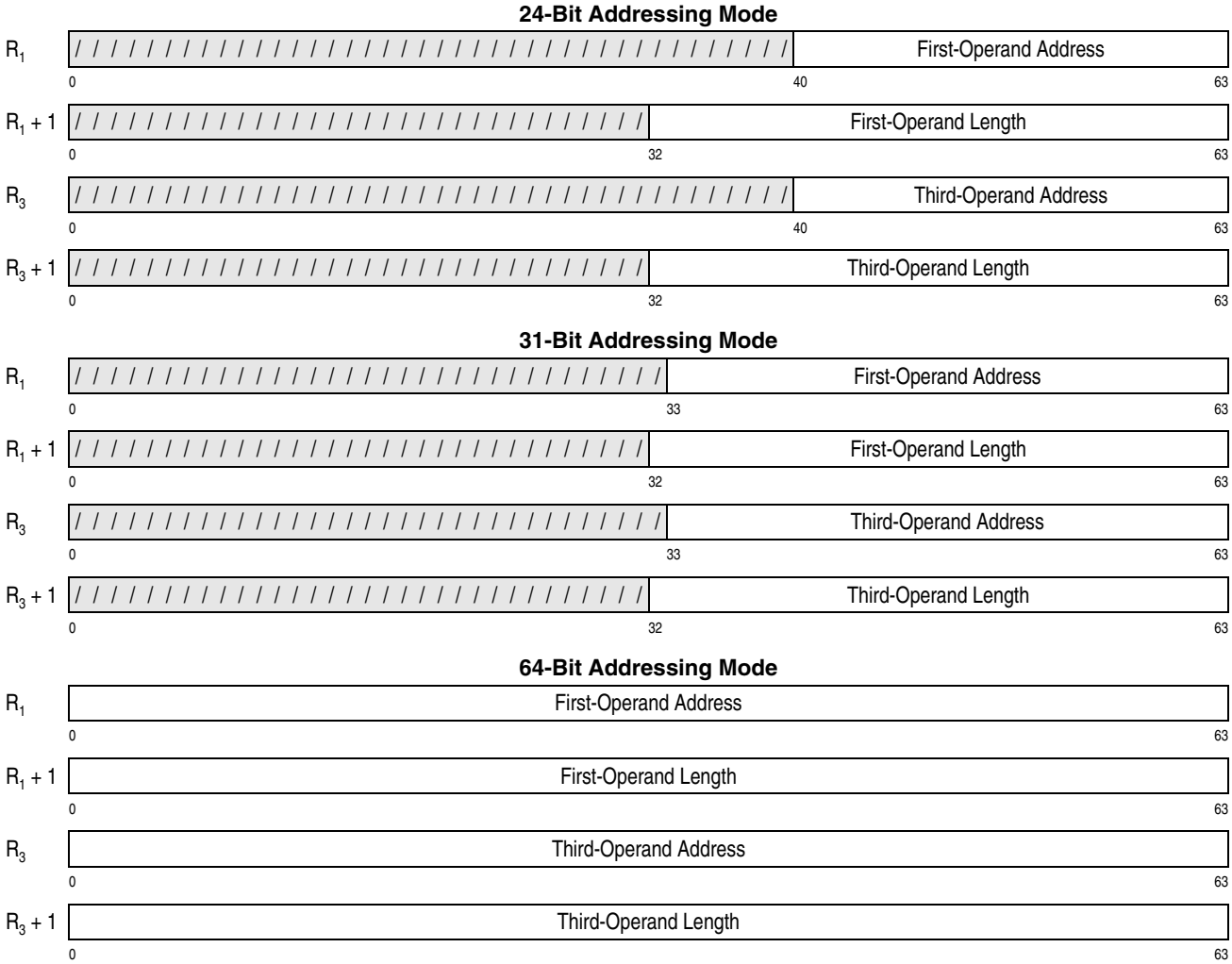


Figure 7-303. Register Contents and Second-Operand Address for MOVE LONG UNICODE (Part 1 of 2)

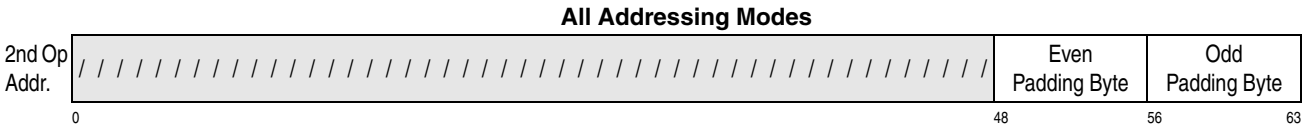


Figure 7-303. Register Contents and Second-Operand Address for MOVE LONG UNICODE (Part 2 of 2)

The result is obtained as if the movement starts at the left end of both fields and proceeds to the right, byte by byte. The operation is ended when the number of bytes specified by the contents of general register $R_1 + 1$ have been placed at the first-operand location or when a CPU-determined number of bytes have been placed, whichever occurs first.

If the third operand is shorter than the first operand, the remaining rightmost positions of the first-operand location are filled with alternating padding bytes. When the number of bytes remaining in the first-operand location is an even number, the even padding byte is placed first; when the number of bytes remaining in the first-operand location is odd, the odd padding byte is placed first.

When the operation is completed because the end of the first operand has been reached, the condition code is set to 0 if the two operand lengths are equal, it is set to 1 if the first-operand length is less than the third-operand length, or it is set to 2 if the first-operand length is greater than the third-operand length. When the operation is completed because a CPU-determined number of bytes have been moved without reaching the end of the first operand, condition code 3 is set.

No test is made for destructive overlap, and the results in the first-operand location are unpredictable when destructive overlap exists. Operands are said to overlap destructively when the first-operand location is used as a source after data has been moved into it.

Operands do not overlap destructively if the leftmost byte of the first operand does not coincide with any of the third-operand bytes participating in the operation other than the leftmost byte of the third operand. When an operand wraps around from location $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) to location 0, operand bytes in locations up to and including $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) are considered to be to the left of bytes in locations from 0 up.

In the 24-bit addressing mode, wraparound is from location $2^{24} - 1$ to location 0; in the 31-bit addressing mode, wraparound is from location $2^{31} - 1$ to location 0; and, in the 64-bit addressing mode, wraparound is from location $2^{64} - 1$ to location 0.

When the length specified in general register $R_1 + 1$ is zero, no movement takes place, and condition code 0 or 1 is set to indicate the relative values of the lengths.

For the nonpadding part of the operation, accesses to the operands for MOVE LONG UNICODE may be multiple-access references. These accesses do not necessarily appear to occur in a left-to-right direction as observed by other CPUs and by channel programs. During the nonpadding part of the operation, operands appear to be accessed doubleword concurrent as observed by other CPUs, provided that both operands start on doubleword boundaries, are an integral number of doublewords in length, and do not overlap.

As observed by other CPUs and by he channel subsystem, that portion of the first operand which is filled with the padding bytes is not necessarily stored into in a left-to-right direction and may appear to be stored into more than once. During the padding operation, stores by other CPUs or by the channel subsystem into that portion of the first operand which is filled with the padding bytes may cause unpredictable results.

At the completion of the operation, the length in general register $R_1 + 1$ is decremented by the number of bytes stored at the first-operand location, and the address in general register R_1 is incremented by the same amount. The length in general register $R_3 + 1$ is decremented by the number of bytes moved out of the third-operand location, and the address in general register R_3 is incremented by the same amount.

If the operation is completed because a CPU-determined number of bytes have been moved without reaching the end of the first operand, the lengths in general registers $R_1 + 1$ and $R_3 + 1$ are decremented by the number of bytes moved, and the addresses in general registers R_1 and R_3 are incremented by the same number, so that the instruction, when reexe-

cuted, resumes at the next byte to be moved. If the operation is completed during padding, the length field in general register $R_3 + 1$ is zero, the address in general register R_3 is incremented by the number of bytes moved from operand 3, and general registers R_1 and $R_1 + 1$ reflect the extent of the padding operation.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R_1, $R_1 + 1$, R_2, and $R_2 + 1$, always remain unchanged.

The padding bytes may be formed from $D_2(B_2)$ multiple times during the execution of the instruction, and the registers designated by R_1 and R_3 may be updated multiple times. Therefore, if B_2 equals R_1, $R_1 + 1$, R_3, or $R_3 + 1$ and is subject to change during the execution of the instruction, the results are unpredictable.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

At the completion of the operation in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R_1 and R_3 may be set to zeros or may remain unchanged from their original values, including the case when one or both of the original length values are zeros.

When the length of an operand is zero, no access exceptions for that operand are recognized. Similarly, when the third operand is longer than the first operand, access exceptions are not recognized for the part of the third-operand field that is in excess of the first-operand field. For operands longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the current location being processed. Access exceptions are not recognized for an operand if the R field associated with that operand is odd. Also, when the R_1 field is odd, PER storage-alteration events are not recognized, and no change bits are set. On models that recognize a specification exception if the length associated with an operand is odd, access exceptions and PER storage-alteration events are not recognized and no change bits are set if the length is odd.

Resulting Condition Code:

0 All bytes moved, operand lengths equal
1 All bytes moved, first-operand length low
2 All bytes moved, first-operand length high
3 CPU-determined number of bytes moved without reaching end of first operand

Program Exceptions:

- Access (fetch, operand 3; store, operand 1)
- Operation (if the extended-translation facility 2 is not installed)
- Specification
- Transaction constraint

Programming Notes:

1. MOVE LONG UNICODE is intended for use in place of MOVE LONG or MOVE LONG EXTENDED when padding with double-byte characters. The character may be a Unicode character or any other double-byte character. MOVE LONG UNICODE may set condition code 3 in cases in which MOVE LONG would be interrupted.

2. When condition code 3 is set, the program can simply branch back to the instruction to continue the movement. The program need not determine the number of bytes that were moved.

3. MOVE LONG UNICODE may be used for filling storage with padding bytes by placing the padding bytes in the second-operand address and setting the third-operand length to zero. However, the stores associated with this clearing may be multiple-access stores and should not be used to clear an area if the possibility exists that another CPU or a channel program will attempt to access and use the area as soon as it appears to be zero. For more details, see “Storage-Operand Consistency” on page 5-125.

4. When the contents of the R_1 and R_3 fields are the same, the contents of the designated registers are incremented or decremented only by the number of bytes moved, not by two times the number of bytes moved. The condition code is finally set to 0 after possible settings to 3.

5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

6. The padding bytes may be represented in the displacement field of the instruction. The following example illustrates padding with a Unicode space character.

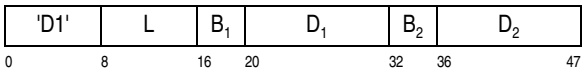
   MVCLU 6,8,X'0020'

   When the B_2 field of the instruction designates general register 0, and the long-displacement facility is not installed, the padding bytes are limited to a character whose representation is less than or equal to 0FFF hex.

7. MOVE LONG UNICODE is intended to be used to move and pad pairs of bytes. Although some models allow the first- and third-operand length registers to designate an odd length, without recognizing a specification exception, the program should always be coded to provide an even length. This ensures that the program will operate compatibly in configurations that recognize a specification exception for an odd length value.

MOVE NUMERICS

MVN $D_1(L,B_1),D_2(B_2)$ [SS-a]



The rightmost four bits of each byte in the second operand are placed in the rightmost bit positions of the corresponding bytes in the first operand. The leftmost four bits of each byte in the first operand remain unchanged.

Each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2; fetch and store, operand 1)
- Transaction constraint

Programming Notes:

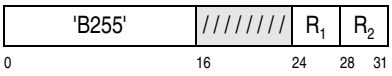
1. An example of the use of the MOVE NUMERICS instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. MOVE NUMERICS moves the numeric portion of a decimal-data field that is in the zoned format. The zoned-decimal format is described in Chapter 8, “Decimal Instructions.” The operands are not checked for valid sign and digit codes.

3. Accesses to the first operand of MOVE NUMERICS consist in fetching the leftmost four bits of each byte in the first operand and subsequently storing the updated value of the byte. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, this instruction cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for OR (OI) in “Multiprogramming and Multiprocessing Examples” in Appendix A, “Number Representation and Instruction-Use Examples.”

MOVE STRING

MVST R_1,R_2 [RRE]



All or part of the second operand is placed in the first-operand location. The operation proceeds until the end of the second operand is reached or a CPU-determined number of bytes have been moved, whichever occurs first. The CPU-determined number is at least one. The result is indicated in the condition code.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers R_1 and R_2, respectively.

The handling of the addresses in general registers R_1 and R_2 is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R_1 and R_2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are

ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The end of the second operand is indicated by an ending character in the last byte position of the operand. The ending character to be used to determine the end of the second operand is specified in bit positions 56-63 of general register 0. Bit positions 32-55 of general register 0 are reserved for possible future extensions and must contain all zeros; otherwise, a specification exception is recognized.

The operation proceeds left to right and ends as soon as the second-operand ending character has been moved or a CPU-determined number of second-operand bytes have been moved, whichever occurs first. The CPU-determined number is at least one. When the ending character is in the first byte position of the second operand, only the ending character is moved. When the ending character has been moved, condition code 1 is set. When a CPU-determined number of second-operand bytes not including an ending character have been moved, condition code 3 is set. Destructive overlap is not recognized. If the second operand is used as a source after it has been used as a destination, the results are unpredictable.

When condition code 1 is set, the address of the ending character in the first operand is placed in general register R_1, and the contents of general register R_2 remain unchanged. When condition code 3 is set, the address of the next byte to be processed in the first and second operands is placed in general registers R_1 and R_2, respectively. Whenever an address is placed in a general register, bits 32-39 of the register, in the 24-bit addressing mode, or bit 32, in the 31-bit addressing mode, are set to zeros. Bits 0-31 of the R_1 and R_2 registers always remain unchanged in the 24-bit or 31-bit mode.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

Access exceptions for the first and second operands are recognized only for that portion of the operand that is necessarily used in the operation.

The storage-operand-consistency rules are the same as for the MOVE (MVC) instruction, except that destructive overlap is not recognized.

Resulting Condition Code:

0 --
1 Entire second operand moved; general register R_1 updated with address of ending character in first operand; general register R_2 unchanged
2 --
3 CPU-determined number of bytes moved; general registers R_1 and R_2 updated with addresses of next bytes

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Specification
- Transaction constraint

Programming Notes:

1. An example of the use of the MOVE STRING instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. When condition code 3 is set, the program can simply branch back to the instruction to continue the data movement. The program need not determine the number of bytes that were moved.

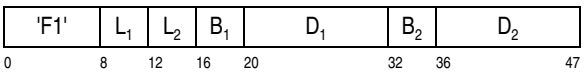
3. R_1 or R_2 may be zero, in which case general register 0 is treated as containing an address and also the ending character.

4. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

5. If the second operand is used as a source after it has been used as a destination, an ending character in the second operand may not be recognized.

MOVE WITH OFFSET

MVO $D_1(L_1,B_1),D_2(L_2,B_2)$ [SS-b]



The second operand is placed to the left of and adjacent to the rightmost four bits of the first operand.

The rightmost four bits of the first operand are attached as the rightmost bits to the second operand, the second-operand bits are offset by four bit posi-

tions, and the result is placed at the first-operand location.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros. If the first operand is too short to contain all of the second operand, the remaining leftmost portion of the second operand is ignored. Access exceptions for the unused portion of the second operand may or may not be indicated.

When the operands overlap, the result is obtained as if the operands were processed one byte at a time, as if each result byte were stored immediately after fetching the necessary operand bytes, and as if the left digit of each second-operand byte were to remain available for the next result byte and need not be refetched.

Condition Code: The code remains unchanged.

Program Exceptions:

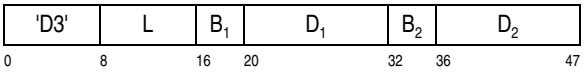
- Access (fetch, operand 2; fetch and store, operand 1)
- Transaction constraint

Programming Notes:

1. An example of the use of the MOVE WITH OFFSET instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. MOVE WITH OFFSET may be used to shift packed decimal data by an odd number of digit positions. The packed-decimal format is described in Chapter 8, “Decimal Instructions.” The operands are not checked for valid sign and digit codes. In many cases, however, SHIFT AND ROUND DECIMAL may be more convenient to use.
3. Access to the rightmost byte of the first operand of MOVE WITH OFFSET consists in fetching the rightmost four bits and subsequently storing the updated value of this byte. These fetch and store accesses to the rightmost byte of the first operand do not necessarily occur one immediately after the other. Thus, this instruction cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for OR (OI) in “Multiprogramming and Multiprocessing Examples” in Appendix A, “Number Representation and Instruction-Use Examples.”
4. The storage-operand references for MOVE WITH OFFSET may be multiple-access references. (See “Storage-Operand Consistency” on page 5-125.)

MOVE ZONES

MVZ $D_1(L,B_1),D_2(B_2)$ [SS-a]



The leftmost four bits of each byte in the second operand are placed in the leftmost four bit positions of the corresponding bytes in the first operand. The rightmost four bits of each byte in the first operand remain unchanged.

Each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after the necessary operand byte is fetched.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2; fetch and store, operand 1)
- Transaction constraint

Programming Notes:

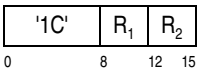
1. An example of the use of the MOVE ZONES instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. MOVE ZONES moves the zoned portion of a decimal field in the zoned format. The zoned format is described in Chapter 8, “Decimal Instructions.” The operands are not checked for valid sign and digit codes.
3. Accesses to the first operand of MOVE ZONES consist in fetching the rightmost four bits of each byte in the first operand and subsequently storing the updated value of the byte. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the

other. Thus, this instruction cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for the OR (OI) instruction in "Multiprogramming and Multiprocessing Examples" in Appendix A, "Number Representation and Instruction-Use Examples."

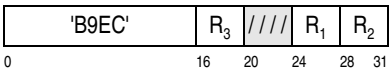
MULTIPLY

Register-and-register format:

MR R_1,R_2 [RR]

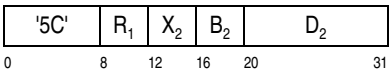


MGRK R_1,R_2,R_3 [RRF-a]

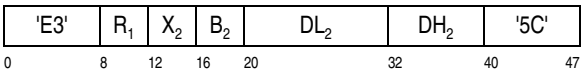


Register-and-storage formats:

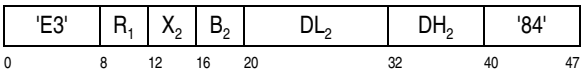
M $R_1,D_2(X_2,B_2)$ [RX-a]



MFY $R_1,D_2(X_2,B_2)$ [RXY-a]



MG $R_1,D_2(X_2,B_2)$ [RXY-a]



For M, MFY, and MR, the 32-bit first operand (the multiplicand) is multiplied by the 32-bit second-operand (the multiplier), and the 64-bit product is placed at the first-operand location. For MG, the 64-bit first operand (the multiplicand) is multiplied by the 64-bit second-operand (the multiplier), and the 128-bit product is placed at the first-operand location. For MGRK, the 64-bit third operand (the multiplicand) is multiplied by the 64-bit second-operand (the multiplier), and the 128-bit product is placed at the first-operand location.

The R_1 field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The following applies to M, MFY, and MR:

- Both the multiplicand and multiplier are treated as 32-bit signed binary integers.
- The multiplicand is in bit positions 32-63 of general register $R_1 + 1$.
- For MULTIPLY (MR), the multiplier is in bit positions 32-63 of general register R_2. The contents of general register R_1 and of bit positions 0-31 of general register $R_1 + 1$ and, for MR, of general register R_2 are ignored.

  For M and MY, the multiplier is at the second-operand location in storage.
- The product is a 64-bit signed binary integer. Bits 0-31 of the product replace bits 32-63 of general register R_1. Bits 32-63 of the product replace bits 32-63 of general register $R_1 + 1$. Bits 0-31 of general registers R_1 and $R_1 + 1$ remain unchanged.

The following applies to MG and MGRK:

- Both the multiplicand and multiplier are treated as 64-bit signed binary integers.
- For MG, the multiplicand is in bit positions 0-63 of general register $R_1 + 1$, and the contents of general register R_1 are ignored. For MGRK, the multiplicand is in bit positions 0-63 of general register R_3, and the contents of general registers R_1 and $R_1 + 1$ are ignored.
- For MG, the multiplier is at the second-operand location in storage. For MGRK, the multiplier is in bit positions 0-63 of general register R_2.
- For both MG and MGRK, the product is a 128-bit signed binary integer. Bits 0-63 of the product replace bits 0-63 of general register R_1. Bits 64-127 of the product replace bits 0-63 of general register $R_1 + 1$.

The sign of the product is determined by the rules of algebra from the multiplier and multiplicand sign, except that a zero result is always positive. An overflow cannot occur.

The displacement for M is treated as a 12-bit unsigned binary integer. The displacement for MFY and MG is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of M, MFY, and MG only)
- Operation (MFY, if the general-instructions-extension facility is not installed; MG, MGRK, if the miscellaneous-instruction-extensions facility 2 is not installed)
- Specification

Programming Notes:

1. An example of the use of the MULTIPLY instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. For M, MFY, and MR, the significant part of the product usually occupies 62 bit positions or fewer. Only when two maximum 32-bit negative numbers are multiplied are 63 significant product bits formed.

   Similarly, for MG and MGRK, the significant part of the product usually occupies 126 bit positions or fewer. Only when two maximum 64-bit negative numbers are multiplied are 127 significant product bits formed.
3. Care should be taken not to confuse the mnemonic for MULTIPLY UNNORMALIZED (MY) with the mnemonic for MULTIPLY (MFY).

MULTIPLY HALFWORD

MH $R_1,D_2(X_2,B_2)$ [RX-a]

'4C'	R_1	X_2	B_2	D_2
0	8	12	16	20 31

MHY $R_1,D_2(X_2,B_2)$ [RXY-a]

'E3'	R_1	X_2	B_2	DL_2	DH_2	'7C'
0	8	12	16	20	32	40 47

MGH $R_1,D_2(X_2,B_2)$ [RXY-a]

'E3'	R_1	X_2	B_2	DL_2	DH_2	'3C'
0	8	12	16	20	32	40 47

MULTIPLY HALFWORD IMMEDIATE

MHI R_1,I_2 [RI-a]

'A7'	R_1	'C'	I_2
0	8	12	16 31

MGHI R_1,I_2 [RI-a]

'A7'	R_1	'D'	I_2
0	8	12	16 31

The 32-bit or 64-bit first operand (the multiplicand) is multiplied by the 16-bit second operand (the multiplier), and the rightmost 32 or 64 bits of the product are placed at the first-operand location. The second operand is two bytes in length and is treated as a 16-bit signed binary integer.

For MULTIPLY HALFWORD (MH, MHY) and MULTIPLY HALFWORD IMMEDIATE (MHI), the multiplicand is treated as a 32-bit signed binary integer in bit positions 32-63 of general register R_1, and it is replaced by the rightmost 32 bits of the signed-binary-integer product. The bits to the left of the 32 rightmost bits of the product are not tested for significance; no overflow indication is given. Bits 0-31 of general register R_1 are ignored and remain unchanged.

For MULTIPLY HALFWORD (MGH) and MULTIPLY HALFWORD IMMEDIATE (MGHI), the multiplicand is treated as a 64-bit signed binary integer in bit positions 0-63 of general register R_1, and it is replaced by the rightmost 64 bits of the signed-binary-integer product. The bits to the left of the 64 rightmost bits of the product are not tested for significance; no overflow indication is given.

The sign of the product is determined by the rules of algebra from the multiplier and multiplicand sign, except that a zero result is always positive.

The displacement for MGH and MH is treated as a 12-bit unsigned binary integer. The displacement for MHY is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2 of MH, MHY, and MGH only)

- Operation (MHY; if the general-instructions-extension facility is not installed; MGH, if the miscellaneous-instruction-extensions facility 2 is not installed)

Programming Notes:

1. An example of the use of the MULTIPLY HALFWORD instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

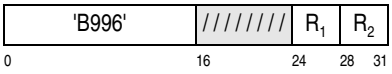
2. For MH, MHY, and MHI, the significant part of the product usually occupies 46 bit positions or fewer. Only when two maximum negative numbers are multiplied are 47 significant product bits formed. Since the rightmost 32 bits of the product are placed unchanged at the first-operand location, ignoring all bits to the left, the sign bit of the result may differ from the true sign of the product in the case of overflow. For a negative product, the 32 bits placed in register R_1 are the rightmost part of the product in two's-complement notation.

   Similarly, for MGH and MGHI, the significant part of the product usually occupies 78 bit positions or fewer, but may occupy 79 bits when two maximum negative numbers are multiplied. Since the rightmost 64 bits of the product are placed unchanged at the first-operand location, the sign of the result may differ from the true sign of the 80-bit product in the case of overflow.

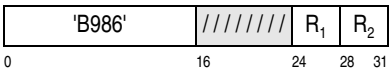
MULTIPLY LOGICAL

Register-and-register formats:

MLR R_1,R_2 [RRE]

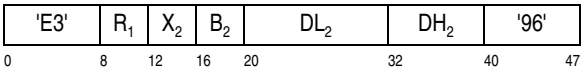


MLGR R_1,R_2 [RRE]

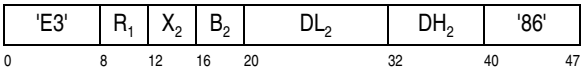


Register-and-storage formats:

ML $R_1,D_2(X_2,B_2)$ [RXY-a]



MLG $R_1,D_2(X_2,B_2)$ [RXY-a]



The 32-bit or 64-bit first operand (the multiplicand) is multiplied by the 32-bit or 64-bit second operand (the multiplier), and the 64-bit or 128-bit product is placed at the first-operand location.

The R_1 field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

For MULTIPLY LOGICAL (MLR, ML), both the multiplicand and the multiplier are treated as 32-bit unsigned binary integers. The multiplicand is in bit positions 32-63 of general register $R_1 + 1$. For MULTIPLY LOGICAL (MLR), the multiplier is in bit positions 32-63 of general register R_2. The contents of general register R_1 and of bit positions 0-31 of general register $R_1 + 1$ and, for MLR, of general register R_2 are ignored. The product is a 64-bit unsigned binary integer. Bits 0-31 of the product replace bits 32-63 of general register R_1, and bits 32-63 of the product replace bits 32-63 of general register $R_1 + 1$. Bits 0-31 of general registers R_1 and $R_1 + 1$ remain unchanged. An overflow cannot occur.

For MULTIPLY LOGICAL (MLGR, MLG), the multiplicand and the multiplier are treated as 64-bit unsigned binary integers. The multiplicand is in general register $R_1 + 1$. The contents of general register R_1 are ignored. The product is a 128-bit unsigned binary integer. Bits 0-63 of the product replace the contents of general register R_1, and bits 64-127 of the product replace the contents of general register $R_1 + 1$. An overflow cannot occur.

Condition Code: The code remains unchanged.

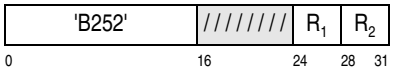
Program Exceptions:

- Access (fetch, operand 2 of ML and MLG only)
- Specification

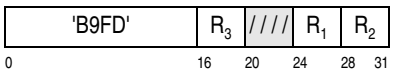
MULTIPLY SINGLE

Register-and-register formats:

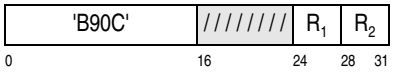
MSR R_1,R_2 [RRE]



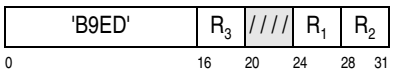
MSRKC R_1,R_2,R_3 [RRF-a]



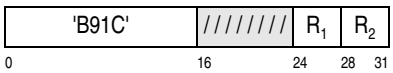
MSGR R_1,R_2 [RRE]



MSGRKC R_1,R_2,R_3 [RRF-a]

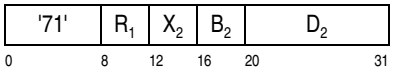


MSGFR R_1,R_2 [RRE]

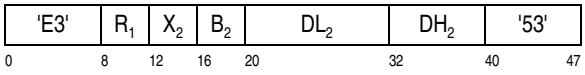


Register-and-storage formats:

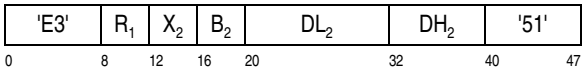
MS $R_1,D_2(X_2,B_2)$ [RX-a]



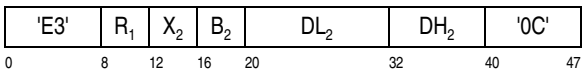
MSC $R_1,D_2(X_2,B_2)$ [RXY-a]



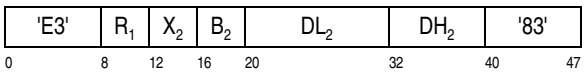
MSY $R_1,D_2(X_2,B_2)$ [RXY-a]



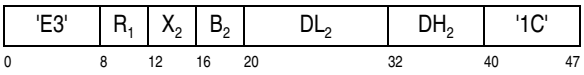
MSG $R_1,D_2(X_2,B_2)$ [RXY-a]



MSGC $R_1,D_2(X_2,B_2)$ [RXY-a]

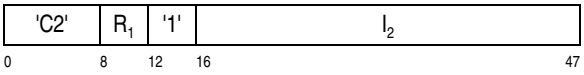


MSGF $R_1,D_2(X_2,B_2)$ [RXY-a]

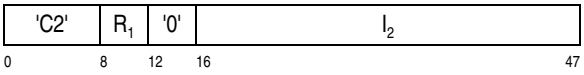


MULTIPLY SINGLE IMMEDIATE

MSFI R_1,I_2 [RIL-a]



MSGFI R_1,I_2 [RIL-a]



The multiplicand is multiplied by the second operand (multiplier), and the rightmost 32 or 64 bits of the product are placed at the first-operand location. Depending on the instruction format, the multiplicand is in either general register R_1 or R_3.

The following applies to MULTIPLY (MS, MSC, MSR, MSRKC, MSY) and to MULTIPLY SINGLE IMMEDIATE (MSFI) – that is, to those operations which produce a 32-bit result:

- The multiplicand is a 32-bit signed binary integer. For MS, MSC, MSFI, MSR, and MSY, the multiplicand is in bit positions 32-63 of general register R_1. For MSRKC, the multiplicand is in bit positions 32-63 of general register R_3.
- The multiplier is a 32-bit signed binary integer. For MSR and MSRKC, the multiplier is in bit positions 32-63 of general register R_2. For MSFI, the multiplier is in the I_2 field of the instruction. For MS, MSC, and MSY, the multiplier is at the second-operand location in storage.
- The rightmost 32 bits of the signed-binary-integer product replace bits 32-63 of general register R_1, and bits 0-31 of the register remain unchanged. For MS, MSFI, MSR, and MSY, the bits to the left of the 32 rightmost bits of the product are not tested for significance.

  For MSC and MSRKC, an overflow condition exists when the leftmost 33 bits of an intermediate 64-bit product are neither all zeros nor all ones. Condition code 3 is set. If the fixed-point-overflow mask in the PSW is one, a program interruption for fixed-point overflow occurs.

The following applies to MULTIPLY SINGLE (MSG, MSGC, MSGF, MSGFR, MSGR, MSGRKC) and to MULTIPLY SINGLE IMMEDIATE (MSGFI) – that is, to those operations which produce a 64-bit result:

- The multiplicand is a 64-bit signed binary integer. For MSG, MSGC, MSGF, MSGFI, and MSGFR, the multiplicand is in bit positions 0-63 of general register R_1. For MSGRKC, the multiplicand is in bit positions 0-63 of general register R_3.
- For MSGR and MSGRKC, the multiplier is a 64-bit signed binary integer in bit positions 0-63 of general register R_2. For MSGFR, the multiplier is a 32-bit signed-binary integer in bit positions 32-63 of general register R_2. For MSGFI, the multiplier is a 32-bit signed-binary integer in the I_2 field of the instruction. For MSG and MSGC, the multiplier is a 64-bit signed binary integer, respectively, at the second-operand location in storage. For MSGFI, the multiplier is a 32-bit signed binary integer at the second-operand location in storage.
- The rightmost 64 bits of the signed-binary-integer product replace bits 0-63 of general register R_1. For MSG, MSGF, MSGFR, MSGR, MSGFI, the bits to the left of the 64 rightmost bits of the product are not tested for significance.

  For MSGC and MSGRKC, an overflow condition exists when the leftmost 65 bits of an intermediate 128-bit product are neither all zeros or all ones. Condition code 3 is set. If the fixed-point-overflow mask in the PSW is one, a program interruption for fixed-point overflow occurs.

The sign of the product is determined by the rules of algebra from the multiplier and multiplicand sign, except that a zero result is always positive.

The displacement for MS is treated as a 12-bit unsigned binary integer. The displacement for MSC, MSG, MSGF, and MSY is treated as a 20-bit signed binary integer.

Resulting Condition Code: For MSC, MSGC, MSGRKC, and MSRKC, the condition code is set as follows:

0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

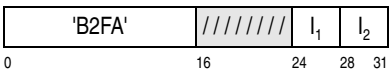
For all other operations, the condition code remains unchanged, and no overflow indication is provided.

Program Exceptions:

- Access (fetch, operand 2 of MS, MSC, MSG, MSGC, MSGF, MSY only)
- Fixed-point overflow (MSC, MSGC, MSGRKC, MSRKC only)
- Operation (MSY, if the long-displacement facility is not installed; MSFI, MSGFI, if the general-instructions-extension facility is not installed; MSC, MSGC, MSGRKC, MSRKC, if the miscellaneous-instruction-extensions facility 2 is not installed.))

NEXT INSTRUCTION ACCESS INTENT

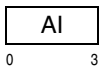
NIAI I_1,I_2 [IE]



Note: In this instruction definition, the term primary-access operand means the lowest numbered storage operand of an instruction. Similarly, the term secondary-access operand means the next-lowest numbered storage operand of the instruction. These terms do not have the same meaning as the terms first operand and second operand, even though there may be some correlation. For example, for the MOVE instruction the first operand is the primary-access operand and the second operand is the secondary-access operand. However, for the LOAD MULTIPLE DISJOINT instruction, the second operand is the primary-access operand and the fourth operand is the secondary-access operand.

Subject to the controls in the I_1 and I_2 fields, the CPU is signaled the future access intent for either or both the primary-access and secondary-access operands of the next-sequential instruction. The I_1 field contains a code to signal the CPU the access intent for the primary-access operand of the next-sequential instruction. The I_2 field contains a code to signal the CPU the access intent for the secondary-access operand of the next-sequential instruction. When the next-sequential instruction has only a single storage operand, the I_2 field is ignored.

The I_1 and the I_2 fields have the following format:



Access Intent (AI): Bits 0-3 of the I_1 and I_2 fields contain an unsigned integer that is used as a code to signal the CPU the access intent for the corresponding operand of the next-sequential instruction as follows:

Code	Meaning
0	The corresponding operand of the next-sequential instruction may or may not be accessed as an instruction operand by subsequent instructions.
1	The corresponding operand of the next-sequential instruction will be accessed by subsequent instructions for operand store access, and may also be accessed for operand fetch access.
2	The corresponding operand of the next-sequential instruction will be accessed by subsequent instructions for operand fetch access.
3	The corresponding operand of the next-sequential instruction will not be accessed as an instruction operand by subsequent instructions.
4-15	Reserved.

Reserved access-intent codes should not be specified; otherwise, the program may not operate compatibly in the future.

Depending on the model, the CPU may not recognize all of the access intents for an operand. For access intents that are not recognized by the CPU, the instruction acts as a no-operation. If the instruction has more than two storage operands, no access intent is specified for the additional operands.

Depending on the model, the CPU may not recognize access intents for a particular instruction. For such cases, the instruction acts as a no-operation.

The NEXT INSTRUCTION ACCESS INTENT instruction only affects subsequent instruction's operand accesses; it does not affect subsequent instruction fetches.

No references to storage are made.

Condition Code: The code remains unchanged.

Program Exceptions:

- Operation (if the execution-hint facility is not installed)

Programming Notes:

1. NEXT INSTRUCTION ACCESS INTENT signals the CPU of the future access intent for the storage operands of the next-sequential instruction, but it does not guarantee that the CPU will necessarily retain or use this information.
2. Specifying an incorrect access intent or an access intent for a non-existent operand may lead to performance degradation.
3. Caches are managed on a cache-line-size granularity. Therefore, when a corresponding storage operand specifies any portion of a cache line, the access intent is signaled for the entire cache line. The cache line size can be determined by executing the EXTRACT CPU ATTRIBUTE instruction.
4. When a corresponding storage operand specifies crossing one or more cache-line boundaries, it is model dependent whether the access intent is signaled for one or multiple cache lines.

   It is anticipated a program may use access-intent values of one, two, or three, with corresponding storage operands that may cross one or more cache-line boundaries.
5. Specifying an access-intent value of three does not signal the CPU to actively release the cache line from the cache for the specified operand. Instead, it signals the CPU that it should not actively try to retain that line in the cache.
6. Specifying an access-intent value of zero to a single operand instruction is not encouraged and not necessary. However, in the case where an instruction has two operands and access intent is known for one operand, but not the other, the access intent value of zero can be used for the operand of unknown intent.
7. If an interruption occurs after the execution of NEXT INSTRUCTION ACCESS INTENT and before the execution of the next-sequential instruction, NEXT INSTRUCTION ACCESS INTENT acts as a no-operation and the specified access intent is ignored.

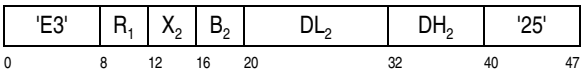
8. If the next sequential instruction after NEXT INSTRUCTION ACCESS INTENT is a PREFETCH DATA (RELATIVE LONG) instruction, it is unpredictable what the CPU will do with data prefetching.

9. When the next-sequential instruction after NEXT INSTRUCTION ACCESS INTENT is an execute-type instruction (EX or EXRL), the following applies:

   - Access-intent codes are not associated with the execute-type instruction. The second operand of an execute-type instruction corresponds to an instruction fetch.
   - Depending on the model, access-intent codes may be associated with the target instruction of the execute-type instruction.

NONTRANSACTIONAL STORE

NTSTG $R_1,D_2(X_2,B_2)$ [RXY-a]



The 64-bit first operand is nontransactionally placed unchanged at the second-operand location.

The displacement is treated as a 20-bit signed binary integer.

The second operand must be aligned on a doubleword boundary; otherwise, a specification exception is recognized and the operation is suppressed.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (store, second operand)
- Operation (transactional-execution facility not installed)
- Specification
- Transaction constraint

Programming Notes:

1. NONTRANSACTIONAL STORE provides a means by which a program executing in the nonconstrained transactional-execution mode can perform stores that will be retained, even if the transaction aborts. This may facilitate debugging of the transaction if it is aborted.

2. When the CPU is not in the transactional-execution mode, the operation of NONTRANSACTIONAL STORE is identical to that of STORE (STG), except that the second operand of NONTRANSACTIONAL STORE is required to be on a doubleword boundary (regardless of transactional-execution mode) whereas the second operand of STG has no alignment requirements.

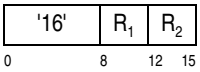
3. Depending on the model, the performance of NONTRANSACTIONAL STORE may be slower than that of STORE (STG).

4. If a CPU makes transactional and non-transactional stores to the same storage location within a transaction, and the transaction then aborts, the content of all storage locations altered by either the transactional or nontransactional store are unpredictable.

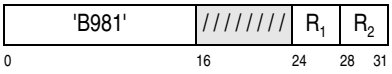
OR

Register-and-register formats:

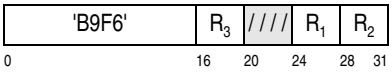
OR R_1,R_2 [RR]



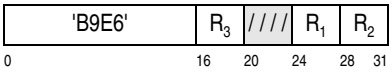
OGR R_1,R_2 [RRE]



ORK R_1,R_2,R_3 [RRF-a]

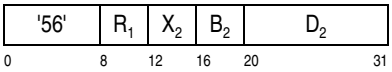


OGRK R_1,R_2,R_3 [RRF-a]

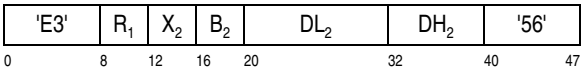


Register-and-storage formats:

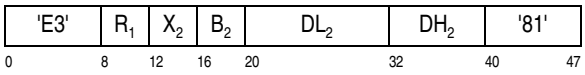
O $R_1,D_2(X_2,B_2)$ [RX-a]



OY $R_1,D_2(X_2,B_2)$ [RXY-a]

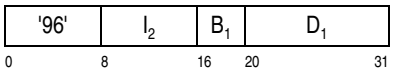


OG $R_1,D_2(X_2,B_2)$ [RXY-a]

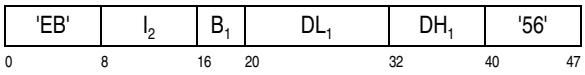


Storage-and-immediate formats:

OI $D_1(B_1),I_2$ [SI]

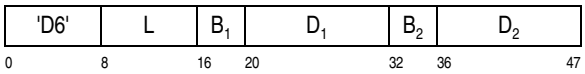


OIY $D_1(B_1),I_2$ [SIY]



Storage-and-storage format:

OC $D_1(L,B_1),D_2(B_2)$ [SS-a]



For O, OC, OG, OGR, OI, OIY, OR, and OY, the OR of the first and second operands is placed at the first-operand location. For OGRK and ORK, the OR of the second and third operands is placed at the first-operand location.

The connective OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit position in one or both operands contains a one; otherwise, the result bit is set to zero.

For OR (OC), each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes.

For OR (OI, OIY), the first operand is one byte in length, and only one byte is stored. When the interlocked-access facility 2 is installed, the update of the first operand appears to be an interlocked-update reference as observed by other CPUs and channel programs, and a specific-operand-serialization operation is performed.

For OR (O, OR, ORK, and OY), the operands are 32 bits, and for OR (OG, OGR, and OGRK), they are 64 bits.

The displacements for O, OI, and both operands of OC are treated as 12-bit unsigned binary integers. The displacement for OY, OIY, and OG is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0 Result zero
1 Result not zero
2 --
3 --

Program Exceptions:

- Access (fetch, operand 2, O, OY, OG, and OC; fetch and store, operand 1, OI, OIY, and OC)
- Operation (OY and OIY, if the long-displacement facility is not installed; OGRK and ORK, if the distinct-operands facility is not installed)
- Transaction constraint (OC)

Programming Notes:

1. Examples of the use of the OR instruction are given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. OR may be used to set a bit to one.

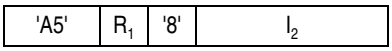
3. Accesses to the first operand of OR (OC) – and, when the interlocked-access facility 2 is not installed, accesses to the first operand of OR (OI, OIY) – consist in fetching a first-operand byte from storage and subsequently storing the updated value. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, these instructions cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown in “Multiprogramming and Multiprocessing Examples” on page A-45.

   When the interlocked-access facility 2 is installed, OR (OI, OIY) can be safely used to update a location in storage, even if the possibility exists that another CPU or a channel program may also be updating the location.

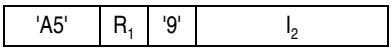
OR IMMEDIATE

OIHF R_1,I_2 [RIL-a]

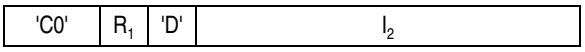
OIHH R_1,I_2 [RI-a]



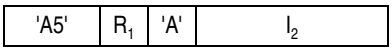
OIHL R_1,I_2 [RI-a]



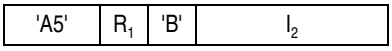
OILF R_1,I_2 [RIL-a]



OILH R_1,I_2 [RI-a]



OILL R_1,I_2 [RI-a]



The second operand is ORed with bits of the first operand, and the result replaces those bits of the first operand. The remainder of the first operand remains unchanged.

For each instruction, the bits of the first operand that are ORed with the second operand and then replaced are as follows:

Instruction	Bits ORed and Replaced
OIHF	0-31
OIHH	0-15
OIHL	16-31
OILF	32-63
OILH	32-47
OILL	48-63

The connective OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit position in one or both operands contains a one; otherwise, the result bit is set to zero.

Resulting Condition Code:

0 Result is zero
1 Result is not zero
2 --
3 --

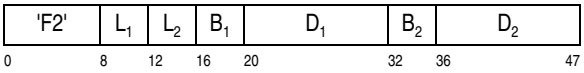
Program Exceptions:

- Operation (OIHF and OILF, if the extended-immediate facility is not installed)

Programming Note: The setting of the condition code is based only on the bits that are ORed and replaced.

PACK

PACK $D_1(L_1,B_1),D_2(L_2,B_2)$ [SS-b]



The format of the second operand is changed from zoned to signed-packed-decimal, and the result is placed at the first-operand location. The zoned and signed-packed-decimal formats are described in Chapter 8, “Decimal Instructions.”

The second operand is treated as having the zoned format. The numeric bits of each byte are treated as a digit. The zone bits are ignored, except the zone bits in the rightmost byte, which are treated as a sign.

The sign and digits are moved unchanged to the first operand and are not checked for valid codes. The sign is placed in the rightmost four bit positions of the rightmost byte of the result field, and the digits are placed adjacent to the sign and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros. If the first operand is too short to contain all digits of the second operand, the remaining leftmost portion of the second operand is ignored. Access exceptions for the unused portion of the second operand may or may not be indicated.

When the operands overlap, the result is obtained as if each result byte were stored immediately after fetching the necessary operand bytes. Two second-operand bytes are needed for each result byte, except for the rightmost byte of the result field, which requires only the rightmost second-operand byte.

Condition Code: The code remains unchanged.

Program Exceptions:

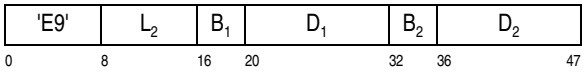
- Access (fetch, operand 2; store, operand 1)
- Transaction constraint

Programming Notes:

1. An example of the use of the PACK instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. PACK may be used to interchange the two hexadecimal digits in one byte by specifying a zero in the L_1 and L_2 fields and the same address for both operands.
3. To remove the zone bits of all bytes of a field, including the rightmost byte, both operands should be extended on the right with a dummy byte, which subsequently should be ignored in the result field.
4. The storage-operand references for PACK may be multiple-access references. (See "Storage-Operand Consistency" on page 5-125.)

PACK ASCII

PKA $D_1(B_1),D_2(L_2,B_2)$ [SS-f]



The format of the second operand is changed from ASCII to signed-packed-decimal, and the result is placed at the first-operand location. The signed-packed-decimal format is described in Chapter 8, "Decimal Instructions."

The second-operand bytes are treated as containing decimal digits, having the binary encoding 0000-1001 for 0-9, in their rightmost four bit positions. The leftmost four bit positions of a byte are ignored. The second operand is considered to be positive.

The implied positive sign (1100 binary) and the source digits are placed at the first-operand location. The source digits are moved unchanged and are not checked for valid codes. The sign is placed in the rightmost four bit positions of the rightmost byte of the result field, and the digits are placed adjacent to the sign and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros.

The length of the first operand is 16 bytes.

The length of the second operand is designated by the contents of the L_2 field. The second-operand length must not exceed 32 bytes (L_2 must be less than or equal to 31); otherwise, a specification exception is recognized.

When the length of the second operand is 32 bytes, the leftmost byte is ignored.

The results are unpredictable if the first and second operands overlap in any way.

As observed by other CPUs and by channel programs, the first-operand location is not necessarily stored into in any particular order.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 2 is not installed)
- Specification
- Transaction constraint

Programming Notes:

1. Although PACK ASCII is primarily intended to change the format of ASCII decimal digits, its use is not restricted to ASCII since the leftmost four bits of each byte are ignored.

2. The following example illustrates the use of the instruction to pack ASCII digits:

```
ASDIGITS  DS    0CL31
          DC    X'3132333435'
          DC    X'3637383930'
          DC    X'3132333435'
          DC    X'3637383930'
          DC    X'3132333435'
          DC    X'3637383930'
          DC    X'31'
PKDIGITS  DS    PL16
          ⋮
          PKA   PKDIGITS,ASDIGITS(31)
```

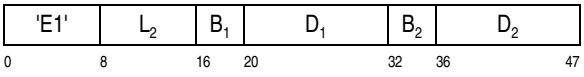
3. The instruction can also be used to pack EBCDIC digits, which is especially useful when the length of the second operand is greater than the 16-byte second-operand limit of PACK.

```
EBDIGITS  DS    0CL31
          DC    X'F1F2F3F4F5'
          DC    X'F6F7F8F9F0'
          DC    X'F1F2F3F4F5'
          DC    X'F6F7F8F9F0'
          DC    X'F1F2F3F4F5'
          DC    X'F6F7F8F9F0'
          DC    X'F1'
PKDIGITS  DS    PL16
          ⋮
          PKA   PKDIGITS,EBDIGITS(31)
```

4. The storage-operand references for PACK ASCII may be multiple-access references. (See "Storage-Operand Consistency" on page 5-125.)

PACK UNICODE

PKU $D_1(B_1),D_2(L_2,B_2)$ [SS-f]



The format of the second operand is changed from Unicode to signed-packed-decimal, and the result is placed at the first-operand location. The signed-packed-decimal format is described in Chapter 8, "Decimal Instructions."

The two-byte second-operand characters are treated as Unicode Basic Latin characters containing decimal digits, having the binary encoding 0000-1001 for 0-9, in their rightmost four bit positions. The leftmost 12 bit positions of a character are ignored. The second operand is considered to be positive.

The implied positive sign (1100 binary) and the source digits are placed at the first-operand location. The source digits are moved unchanged and are not checked for valid codes. The sign is placed in the rightmost four bit positions of the rightmost byte of the result field, and the digits are placed adjacent to the sign and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros.

The length of the first operand is 16 bytes.

The byte length of the second operand is designated by the contents of the L_2 field. The second-operand length must not exceed 32 characters or 64 bytes, and the byte length must be even (L_2 must be less than or equal to 63 and must be odd); otherwise, a specification exception is recognized.

When the length of the second operand is 32 characters (64 bytes), the leftmost character is ignored.

The results are unpredictable if the first and second operands overlap in any way.

As observed by other CPUs and by channel programs, the first-operand location is not necessarily stored into in any particular order.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 2 is not installed)
- Specification
- Transaction constraint

Programming Notes:

1. The following example illustrates the use of PACK UNICODE to pack European numbers:

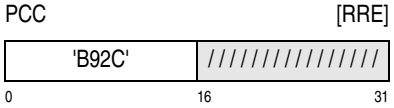
```
UNDIGITS  DS    0CL62
          DC    X'0031003200330034003 5'
          DC    X'0036003700380039003 0'
          DC    X'0031003200330034003 5'
          DC    X'0036003700380039003 0'
          DC    X'0031003200330034003 5'
          DC    X'0036003700380039003 0'
          DC    X'0031'
PKDIGITS  DS    PL16
          ⋮
          PKU   PKDIGITS,UNDIGITS(62)
```

2. Because the leftmost 12 bits of each character are ignored, those Unicode decimal digits where the digit zero has four rightmost zero bits can also be packed by the instruction. For example, for Thai digits:

```
UNDIGITS  DS    0CL62
          DC    X'0E510E520E530E540E55'
          DC    X'0E560E570E580E590E50'
          DC    X'0E510E520E530E540E55'
          DC    X'0E560E570E580E590E50'
          DC    X'0E510E520E530E540E55'
          DC    X'0E560E570E580E590E50'
          DC    X'0E51'
PKDIGITS  DS    PL16
          ⋮
          PKU   PKDIGITS,UNDIGITS(62)
```

3. The storage-operand references for PACK UNICODE may be multiple-access references. (See “Storage-Operand Consistency” on page 5-125.)

PERFORM CRYPTOGRAPHIC COMPUTATION



A function specified by the function code in general register 0 is performed.

Bits 16-31 of the instruction are ignored.

Bit positions 57-63 of general register 0 contain the function code. Figure 7-304 on page 7-315 shows the assigned function codes. All other function codes are unassigned. Bit 56 of general register 0 must be zero; otherwise, a specification exception is recognized. All other bits of general register 0 are ignored.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register 1 constitute the address.

The function codes for PERFORM CRYPTOGRAPHIC COMPUTATION are as follows.

Code	Function	Parm. Block Size (bytes)	Data Block Size (bytes)
0	PCC-Query	16	—
1	PCC-Compute-Last-Block-CMAC-Using-DEA	32	8
2	PCC-Compute-Last-Block-CMAC-Using-TDEA-128	40	8
3	PCC-Compute-Last-Block-CMAC-Using-TDEA-192	48	8
9	PCC-Compute-Last-Block-CMAC-Using-Encrypted-DEA	56	8
10	PCC-Compute-Last-Block-CMAC-Using-Encrypted-TDEA-128	64	8
11	PCC-Compute-Last-Block-CMAC-Using-Encrypted-TDEA-192	72	8
18	PCC-Compute-Last-Block-CMAC-Using-AES-128	56	16
19	PCC-Compute-Last-Block-CMAC-Using-AES-192	64	16

Figure 7-304. Function Codes for PERFORM CRYPTOGRAPHIC COMPUTATION

All other function codes are unassigned.

The query function provides the means of indicating the availability of the other functions.

Figure 7-305 shows the contents of general registers 0 and 1.

In the access-register mode, access register 1 specifies the address space containing the parameter block.

As observed by this CPU, other CPUs, and channel programs, reference to the parameter block may be

Machine Format

Op Code	L_1	L_2	B_1	D_1	B_2	D_2
F1	3	2	C	000	F	000

Assembler Format

```
Op Code   D1(L1,B1),D2(L2,B2)
MVO       0(4,12),0(3,15)
```

is executed, the storage locations 5600-5603 contain 01 23 45 6C. Note that the second operand is extended on the left with one zero to fill out the first-operand field.

MOVE ZONES (MVZ)

The MOVE ZONES instruction can operate on overlapping or nonoverlapping fields, as can the instructions MOVE (MVC) and MOVE NUMERICS. When operating on nonoverlapping fields, MOVE ZONES works like the MOVE NUMERICS instruction (see its example), except that MOVE ZONES moves only the zone bits of each byte. To illustrate the use of MOVE ZONES with overlapping fields, assume that the following data field is in storage:

800					805
F1	C2	F3	C4	F5	C6

Also assume that register 15 contains 00 00 08 00. The instruction:

Machine Format

Op Code	L	B_1	D_1	B_2	D_2
D3	04	F	001	F	000

Assembler Format

```
Op Code   D1(L,B1),D2(B2)
MVZ       1(5,15),0(15)
```

propagates the zone bits from the byte at address 800 through the entire field, so that the field becomes:

800					805
F1	F2	F3	F4	F5	F6

MULTIPLY (M, MR)

Assume that a number in register 5 is to be multiplied by the contents of a four-byte field at address 3750. Initially:

The contents of register 4 are not significant.

Register 5 contains 00 00 00 9A = 154_{10} = the multiplicand.

Register 11 contains 00 00 06 00.

Register 12 contains 00 00 30 00.

Storage locations 3750-3753 contain 00 00 00 83 = 131_{10} = the multiplier.

The instruction required for performing the multiplication is:

Machine Format

Op Code	R_1	X_2	B_2	D_2
5C	4	B	C	150

Assembler Format

```
Op Code   R1,D2(X2,B2)
M         4,X'150'(11,12)
```

After the instruction is executed, the product is in the register pair 4 and 5:

Register 4 contains 00 00 00 00.

Register 5 contains 00 00 4E CE = $20{,}174_{10}$.

Storage locations 3750-3753 are unchanged.

The RR format of the instruction can be used to square the number in a register. Assume that register 7 contains 00 01 00 05. The contents of register 6 are not significant. The instruction:

Machine Format

Op Code	R_1	R_2
1C	6	7

Assembler Format

```
Op Code   R1,R2
MR        6,7
```

multiplies the number in register 7 by itself and places the result in the pair of registers 6 and 7:

Register 6 contains 00 00 00 01.

Register 7 contains 00 0A 00 19.

MULTIPLY HALFWORD (MH)

The MULTIPLY HALFWORD instruction is used to multiply the contents of a register by a two-byte field in storage. For example, assume that:

Register 11 contains 00 00 00 15 =21_{10} = the multiplicand.

Register 14 contains 00 00 01 00.

Register 15 contains 00 00 20 00.

Storage locations 2102-2103 contain FF D9 = -39_{10} = the multiplier.

The instruction:

Machine Format

Op Code	R_1	X_2	B_2	D_2
4C	B	E	F	002

Assembler Format

```
Op Code  R1,D2(X2,B2)
-------------------------
   MH    11,2(14,15)
```

multiplies the two numbers. The product, FF FF FC CD = -819_{10}, replaces the original contents of register 11.

Only the rightmost 32 bits of a product are stored in a register; any significant bits on the left are lost. No program interruption occurs on overflow.

OR (O, OC, OI, OR)

When the Boolean operator OR is applied to two bits, the result is one when either bit is one; otherwise, the result is zero. When two bytes are ORed, each pair of bits is handled separately; there is no connection from one bit position to another. The following is an example of ORing two bytes:

```
First-operand byte:    0011 0101₂
Second-operand byte:   0101 1100₂
-----------------------------------
Result byte:           0111 1101₂
```

OI Example

A frequent use of the OR instruction is to set a particular bit to one. For example, assume that storage location 4891 contains 0100 0010_2. To set the rightmost bit of this byte to one without affecting the other bits, the following instruction can be used (assume that register 8 contains 00 00 48 90):

Machine Format

Op Code	I_2	B_1	D_1
96	01	8	001

Assembler Format

```
Op Code  D1(B1),I2
-------------------
   OI    1(8),X'01'
```

When this instruction is executed, the byte in storage is ORed with the immediate byte (the I_2 field of the instruction):

```
Location 4891:     0100 0010₂
Immediate byte:    0000 0001₂
-----------------------------------
Result:            0100 0011₂
```

The resulting byte with bit 7 set to one is stored back in location 4891. Condition code 1 is set.

PACK (PACK)

Assume that storage locations 1000-1003 contain the following zoned-decimal number that is to be converted to a signed-packed-decimal number and left in the same location:

	1000			1003
Zoned number	F1	F2	F3	C4

Also assume that register 12 contains 00 00 10 00. After the instruction:

Machine Format

Op Code	L_1	L_2	B_1	D_1	B_2	D_2
F2	3	3	C	000	C	000

Assembler Format

Op Code	$D_1(L_1,B_1),D_2(L_2,B_2)$
PACK	0(4,12),0(4,12)

is executed, the result in locations 1000-1003 is in the signed-packed-decimal format:

	1000			1003
Signed-packed number	00	01	23	4C

Notes:

1. This example illustrates the operation of PACK when the first- and second-operand fields overlap completely.
2. During the operation, the second operand was extended on the left with zeros.

ROTATE THEN EXCLUSIVE OR SELECTED BITS

The following example illustrates the use of ROTATE THEN EXCLUSIVE OR SELECTED BITS to determine if the access-control bits of a page's storage-protection key match the key in the PSW. In this example, only the condition code is set; the result is not placed in the first-operand register.

```
LAY   9,BLOCK   Point to storage block.
IVSK  14,9      Insert key into bits 56-63.
EPSW  4,0       Extract PSW key (bits 40-43).
RXSBG 4,14,128+40,43,16 Compare keys.
```

On completion, the condition code will be set to zero if the PSW key matches the access-control bits in the storage key of page addressed by general register 9. Condition code 1 will be set if the keys do not match.

ROTATE THEN INSERT SELECTED BITS

1. The following example illustrates the use of ROTATE THEN INSERT SELECTED BITS to extract the various DAT-table indices of a virtual address contained in general register 8. This example shows the use of the zero-remaining-bits control to set the remaining bits of the result registers to zero.

   On completion, condition code 1 is set, and general register 6 is as follows:

   The virtual address in general register 8 is assumed to be 123456789ABCDEF0 hex.

   ```
   RISBG 1,8,53,128+63,11 Get reg. 1st index.
   RISBG 2,8,53,128+63,22 Get reg. 2nd index.
   RISBG 3,8,53,128+63,33 Get reg. 3rd index.
   RISBG 4,8,53,128+63,44 Get segment index.
   RISBG 5,8,56,128+63,52 Get page index.
   RISBG 6,8,52,128+63,0  Get byte index.
   ```

   The condition code following each RISBG is 2. On completion, general registers 1-6 contain the region-first index, region-second index, region-third index, segment index, page index, and byte index, respectively, as follows:

   ```
   GR1: 0000000000000091 (RFX)
   GR2: 0000000000000515 (RSX)
   GR3: 00000000000004F1 (RTX)
   GR4: 00000000000001AB (SX)
   GR5: 00000000000000CD (PX)
   GR6: 0000000000000EF0 (BX)
   ```

2. The following example illustrates shifting a 64-bit value to the left by *L* bits:

   ```
   RISBG R1,R2,0,X'80'+63-L,L
   ```

3. The following example illustrates shifting a 32-bit value to the right by *R* bits and setting the left-most 32 bits to zero:

   ```
   RISBG R1,R2,32+R,X'80'+63,-R
   ```

4. The following example illustrates zeroing bits *S* through *E* of general register 3.

   ```
   RISBG 3,3,mod64(E+1),128+mod64(S-1),0
   ```

   Mod64 represents a modulo function which effectively ANDs the value in parentheses with 3F hex.

ROTATE THEN OR SELECTED BITS

The following example illustrates the use of ROTATE THEN OR SELECTED BITS to rotate and combine selected bits of general registers 6 and 8.

```
LG    6,=X'004C487040CF4600'
LG    8,=X'C07FAF37FC968280'
ROSBG 6,8,8,54,32
```

```
GR6: 00DECAF0C0FFEE00
```

Note that only bits 8-54 of the registers are ORed; bits 0-7 and 55-63 of general register 6 are not modified.

SEARCH STRING (SRST)

The SEARCH STRING instruction is used to search a second operand designated by general register R_2 for a character specified in general register 0. The length of the second operand is known — the address of the first byte after the second operand is in general register R_1.

When the specified character is found, condition code 1 is set, the address of the character is placed in general register R_1, and the contents of general register R_2 remain unchanged. When the address of the next second-operand byte to be examined equals the address in general register R_1, condition code 2 is set, and the contents of general register R_1 and R_2 remain unchanged. When a CPU-determined number of second-operand bytes have been examined, condition code 3 is set, the address of the next byte to be processed in the second operand is placed in general register R_2, and the contents of general register R_1 remain unchanged.

SRST Example 1

Following is an example program that determines the end of string A, as indicated by an ending character equal to 00 hex (as in the C programming language), and then determines the address of the first character equal to C1 hex in the string. The program is based on the assumption that the second operand does not begin at location 0 or wrap around in storage, and, therefore, condition code 2 will not be set by the first SEARCH STRING instruction because of the address in general register 0. The program is not written for execution in the access-register mode.

```
     L     5,STRAADR
     SR    0,0
LOOP1 SRST  0,5
     BC    1,LOOP1
     L     5,STRAADR
     LR    4,0
     LA    0,X'C1'
LOOP2 SRST  4,5
     BC    1,LOOP2
     BC    2,NOTFND
FOUND [Any instruction]
     ...
NOTFND [Any instruction]
```

SRST Example 2

Following is an example program that determines the address of the first character equal to C1 hex in the string A whose length is known. The program is not written for execution in the access-register mode.

```
     L     5,STRAADR
     L     4,STRALEN
     AR    4,5
     LA    0,X'C1'
LOOP1 SRST  4,5
     BC    1,LOOP1
     BC    2,NOTFND
FOUND [Any instruction]
     ...
NOTFND [Any instruction]
```

In this example, the value in STRALEN may be a length that either does or does not include an ending character at the end of the string, provided that the ending character is not the character for which the search is made.

SHIFT LEFT DOUBLE (SLDA)

The SHIFT LEFT DOUBLE instruction shifts the 63 numeric bits of an even-odd register pair to the left, leaving the sign bit unchanged. Thus, the instruction performs an algebraic left shift of a 64-bit signed binary integer.

For example, if the contents of registers 2 and 3 are:

```
00 7F 0A 72   FE DC BA 98 =
00000000 01111111 00001010 01110010
11111110 11011100 10111010 10011000₂
```

The instruction:

Machine Format

Op Code	R_1		B_2	D_2
8F	2	////	0	01F

Assembler Format

```
Op Code  R1,D2(B2)
SLDA     2,31(0)
```

results in registers 2 and 3 both being left-shifted 31 bit positions, so that their new contents are:

```
7F 6E 5D 4C   00 00 00 00 =
01111111 01101110 01011101 01001100
00000000 00000000 00000000 00000000₂
```

Because significant bits are shifted out of bit position 1 of register 2, overflow is indicated by setting condition code 3, and, if the fixed-point-overflow mask bit in the PSW is one, a fixed-point-overflow program interruption occurs.

SHIFT LEFT SINGLE (SLA)

The SHIFT LEFT SINGLE instruction is similar to SHIFT LEFT DOUBLE, except that it shifts only the 31 numeric bits of a single register. Therefore, this instruction performs an algebraic left shift of a 32-bit signed binary integer.

For example, if the contents of register 2 are:

```
00 7F 0A 72 = 00000000 01111111 00001010 01110010₂
```

The instruction:

Machine Format

Op Code	R_1		B_2	D_2
8B	2	////	0	008

Assembler Format

```
Op Code  R1,D2(B2)
──────────────────
  SLA    2,8(0)
```

results in register 2 being shifted left eight bit positions so that its new contents are:

```
7F 0A 72 00 = 01111111 00001010 01110010 00000000₂
```

Condition code 2 is set to indicate that the result is greater than zero.

If a left shift of nine places had been specified, a significant bit would have been shifted out of bit position 1. Condition code 3 would have been set to indicate this overflow and, if the fixed-point-overflow mask bit in the PSW were one, a fixed-point overflow interruption would have occurred.

STORE CHARACTERS UNDER MASK (STCM)

STORE CHARACTERS UNDER MASK (STCM) may be used to place selected bytes from a register into storage. For example, if it is desired to store a three-byte address from general register 8 into location FIELD3, assume:

Machine Format

Op Code	R_1	M_3	S_2
BE	8	7	* * * *

Register Format

```
Op Code  R1,M3,S2
─────────────────────────
 STCM    8,B'0111',FIELD3

Register 8:       12 34 56 78
FIELD3 (before):  not significant
FIELD3 (after):   34 56 78
```

As another example:

Machine Format

Op Code	R_1	M_3	S_2
BE	9	5	* * * *

Register Format

```
Op Code  R1,M3,S2
─────────────────────────
 STCM    9,B'0101',FIELD2

Register 9:      01 23 45 67
FIELD2 (before): not significant
FIELD2 (after):  23 67
```

STORE MULTIPLE (STM)

Assume that the contents of general registers 14, 15, 0, and 1 are to be stored in consecutive four-byte fields starting with location 4050 and that:

Register 14 contains 00 00 25 63.

Register 15 contains 00 01 27 36.

Register 0 contains 12 43 00 62.

Register 1 contains 73 26 12 57.

Register 6 contains 00 00 40 00.

The initial contents of locations 4050-405F are not significant.

The STORE MULTIPLE instruction allows the use of just one instruction to store the contents of the four registers:

Machine Format

Op Code	R_1	R_3	B_2	D_2
90	E	1	6	050

Assembler Format

```
Op Code  R1,R3,D2(B2)
-----------------------
  STM    14,1,X'50'(6)
```

After the instruction is executed:

Locations 4050-4053 contain 00 00 25 63.

Locations 4054-4057 contain 00 01 27 36.

Locations 4058-405B contain 12 43 00 62.

Locations 405C-405F contain 73 26 12 57.

TEST UNDER MASK (TM)

The TEST UNDER MASK instruction examines selected bits of a byte and sets the condition code accordingly. For example, assume that:

Storage location 9999 contains FB.

Register 7 contains 00 00 99 90.

Assume the instruction to be:

Machine Format

Op Code	I_2	B_1	D_1
91	C3	7	009

Assembler Format

```
Op Code  D1(B1),I2
-------------------
   TM    9(7),B'11000011'
```

The instruction tests only those bits of the byte in storage for which the mask bits are ones:

```
FB   = 1111 1011
Mask = 1100 0011
-----------------
Test = 11xx xx11
```

Condition code 3 is set: all selected bits in the test result are ones. (The bits marked “x” are ignored.)

If location 9999 had contained B9, the test would have been:

```
B9   = 1011 1001
Mask = 1100 0011
-----------------
Test = 10xx xx01
```

Condition code 1 is set: the selected bits are both zeros and ones.

If location 9999 had contained 3C, the test would have been:

```
3C   = 0011 1100
Mask = 1100 0011
-----------------
Test = 00xx xx00
```

Condition code 0 is set: all selected bits are zeros.

Note: Storage location 9999 remains unchanged.

TRANSLATE (TR)

The TRANSLATE instruction can be used to translate data from any character code to any other desired code, provided that each character code consists of eight bits or fewer. An appropriate translation table is required in storage.

In the following example, EBCDIC code is translated to ASCII code. The first step is to create a 256-byte table in storage locations 1000-10FF. This table contains the characters of the ASCII code in the sequence of the binary representation of the EBCDIC code; that is, the ASCII representation of a character is placed in storage at the starting address of the table plus the binary value of the EBCDIC representation of the same character.

For simplicity, the example shows only the part of the table containing the decimal digits:

10F0									10F9
30	31	32	33	34	35	36	37	38	39

Assume that the four-byte field at storage location 2100 contains the EBCDIC code for the digits 1984:

Locations 2100-2103 contain F1 F9 F8 F4.

Register 12 contains 00 00 21 00.

Register 15 contains 00 00 10 00.

As the instruction:

Machine Format

Op Code	L	B_1	D_1	B_2	D_2
DC	03	C	000	F	000

Assembler Format

```
Op Code  D1(L,B1),D2(B2)
_____________________________

   TR    0(4,12),0(15)
```

is executed, the binary value of each EBCDIC byte is added to the starting address of the table, and the resulting address is used to fetch an ASCII byte:

```
Table starting address:    1000
First EBCDIC byte:           F1
_____________________________________

Address of ASCII byte:     10F1
```

After execution of the instruction:

Locations 2100-2103 contain 31 39 38 34.

Thus, the ASCII code for the digits 1984 has replaced the EBCDIC code in the four-byte field at storage location 2100.

TRANSLATE AND TEST (TRT)

The TRANSLATE AND TEST instruction can be used to scan a data field for characters with a special meaning. To indicate which characters have a special meaning, a table similar to the one used for the TRANSLATE instruction is set up, except that zeros in the table indicate characters without any special meaning and nonzero values indicate characters with a special meaning.

Figure A-4 on page A-34 has been set up to distinguish alphameric characters (A to Z and 0 to 9) from blanks, certain special symbols, and all other characters which are considered invalid. EBCDIC coding is assumed. The 256-byte table is assumed stored at locations 2000-20FF.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
200_	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
201_	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
202_	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
203_	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
204_	04	40	40	40	40	40	40	40	40	40	40	08	40	0C	10	40
205_	14	40	40	40	40	40	40	40	40	40	40	18	1C	20	40	40
206_	24	28	40	40	40	40	40	40	40	40	40	2C	40	40	40	40
207_	40	40	40	40	40	40	40	40	40	40	40	30	34	38	3C	40
208_	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
209_	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
20A_	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
20B_	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
20C_	40	00	00	00	00	00	00	00	00	00	40	40	40	40	40	40
20D_	40	00	00	00	00	00	00	00	00	00	40	40	40	40	40	40
20E_	40	40	00	00	00	00	00	00	00	00	40	40	40	40	40	40
20F_	00	00	00	00	00	00	00	00	00	00	40	40	40	40	40	40

Note: If the character codes in the statement being translated occupy a range smaller than 00 through FF_{16}, a table of fewer than 256 bytes can be used.

Figure A-4. Translate and Test Table

The table entries for the alphameric characters in EBCDIC are 00; thus, the letter A (code C1) corresponds to byte location 20C1, which contains 00.

The 15 special symbols have nonzero entries from 04_{16} to $3C_{16}$ in increments of 4. Thus, the blank (code 40) has the entry 04_{16}, the period (code 4B) has the entry 08_{16}, and so on.

All other table positions have the entry 40_{16} to indicate an invalid character.

The table entries are chosen so that they may be used to select one of a list of 16 words containing addresses of different routines to be entered for each special symbol or invalid character encountered during the scan.

Assume that this list of 16 branch addresses is stored at locations 3004-3043.

Starting at storage location CA80, there is the following sequence of 21_{10} EBCDIC characters, where "b" stands for a blank.

Locations CA80-CA94:
UNPKbPROUT(9),WORD(5)

Also assume:

Register 1 contains 00 00 CA 7F.

Register 2 contains 00 00 30 00.

Register 15 contains 00 00 20 00.

As the instruction:

Machine Format

Op Code	L	B_1	D_1	B_2	D_2
DD	14	1	001	F	000

Assembler Format

```
Op Code  D1(L,B1),D2(B2)
TRT      1(21,1),0(15)
```

is executed, the value of the first source byte, the EBCDIC code for the letter U, is added to the starting address of the table to produce the address of the table entry to be examined:

```
Table starting address      2000
First source byte (U)         E4
Address of table entry      20E4
```

Because zeros were placed in storage location 20E4, no special action occurs. The operation continues with the second and subsequent source bytes until it reaches the blank in location CA84. When this symbol is reached, its value is added to the starting address of the table, as usual:

```
Table starting address      2000
Source byte (blank)           40
Address of table entry      2040
```

Because location 2040 contains a nonzero value, the following actions occur:

The address of the source byte, 00CA84, is placed in the rightmost 24 bits of register 1.

The table entry, 04, is placed in the rightmost eight bits of register 2, which now contains 00 00 30 04.

Condition code 1 is set (scan not completed).

The TRANSLATE AND TEST instruction may be followed by instructions to branch to the routine at the address found at location 3004, which corresponds to the blank character encountered in the scan. When this routine is completed, program control may return to the TRANSLATE AND TEST instruction to continue the scan, except that the length must first be adjusted for the characters already scanned.

For this purpose, the TRANSLATE AND TEST may be executed by the use of an EXECUTE instruction, which supplies the length specification from a general register. In this way, a complete statement scan can be performed with a single TRANSLATE AND TEST instruction used repeatedly by means of EXECUTE, and without modifying any instructions in storage. In the example, after the first execution of TRANSLATE AND TEST, register 1 contains the address of the last source byte translated. It is then a simple matter to subtract this address from the address of the last source byte (CA94) to produce a length specification. This length minus one is placed in the register that is referenced as the R_1 field of the EXECUTE instruction. (Note that the length code in the machine format is one less than the total number of bytes in the field.) The second-operand address of the EXECUTE instruction points to the TRANSLATE AND TEST instruction, which is the same as illustrated above, except for the length (L) which is set to zero.

UNPACK (UNPK)

Assume that storage locations 2501-2502 contain a signed-packed-decimal number that is to be unpacked and placed in storage locations 1000-1004. Also assume:

Register 12 contains 00 00 10 00.

Register 13 contains 00 00 25 00.

Storage locations 2501-2502 contain 12 3D.

The initial contents of storage locations 1000-1004 are not significant.

After the instruction:

Machine Format

Op Code	L_1	L_2	B_1	D_1	B_2	D_2
F3	4	1	C	000	D	001

Assembler Format

```
Op Code  D1(L1,B1),D2(L2,B2)
UNPK     0(5,12),1(2,13)
```

is executed, the storage locations 1000-1004 contain F0 F0 F1 F2 D3.

UPDATE TREE (UPT)

See “Sorting Instructions” on page A-53.

Decimal Instructions

(See Chapter 8, “Decimal Instructions” for a complete description of the decimal instructions.)

ADD DECIMAL (AP)

Assume that the signed-packed-decimal number at storage locations 500-503 is to be added to the signed-packed-decimal number at locations 2000-2002. Also assume:

Register 12 contains 00 00 20 00.

Register 13 contains 00 00 05 00.

Storage locations 2000-2002 contain 38 46 0D (a negative number).

Storage locations 500-503 contain 01 12 34 5C (a positive number).

After the instruction:

Machine Format

Op Code	L_1	L_2	B_1	D_1	B_2	D_2
FA	2	3	C	000	D	000

Assembler Format

```
Op Code  D1(L1,B1),D2(L2,B2)
AP       0(3,12),0(4,13)
```

is executed, the storage locations 2000-2002 contain 73 88 5C; condition code 2 is set to indicate that the result is greater than zero. Note that:

1. Because the two numbers had different signs, they were in effect subtracted.
2. Although the second operand is longer than the first operand, no overflow interruption occurs because the result can be entirely contained within the first operand.

COMPARE DECIMAL (CP)

Assume that the signed-packed-decimal contents of storage locations 700-703 are to be algebraically compared with the signed-packed-decimal contents of locations 500-502. Also assume:

Register 12 contains 00 00 06 00.

Register 13 contains 00 00 03 00.

Storage locations 700-703 contain 17 25 35 6D.

Storage locations 500-502 contain 72 14 2D.

After the instruction:

Machine Format

Op Code	L_1	L_2	B_1	D_1	B_2	D_2
F9	3	2	C	100	D	200

Assembler Format

```
Op Code  D1(L1,B1),D2(L2,B2)
CP       X'100'(4,12),X'200'(3,13)
```

is executed, condition code 1 is set, indicating that the first operand (the contents of locations 700-703) is less than the second.

DIVIDE DECIMAL (DP)

Assume that the signed-packed-decimal number at storage locations 2000-2004 (the dividend) is to be divided by the signed-packed-decimal number at locations 3000-3001 (the divisor). Also assume:

Register 12 contains 00 00 20 00.

Register 13 contains 00 00 30 00.

Storage locations 2000-2004 contain 01 23 45 67 8C.

Storage locations 3000-3001 contain 32 1D.

After the instruction:

Machine Format

Op Code	L_1	L_2	B_1	D_1	B_2	D_2
FD	4	1	C	000	D	000

Assembler Format

```
Op Code  D1(L1,B1),D2(L2,B2)
-------------------------------
   DP    0(5,12),0(2,13)
```

is executed, the dividend is entirely replaced by the signed quotient and remainder, as follows:

	2000				2004
Locations 2000-2004	38	46	0D	01	8C
	quotient			remainder	

Notes:

1. Because the dividend and divisor have different signs, the quotient receives a negative sign.
2. The remainder receives the sign of the dividend and the length of the divisor.
3. If an attempt were made to divide the dividend by the one-byte field at location 3001, the quotient would be too long to fit within the four bytes allotted to it. A decimal-divide exception would exist, causing a program interruption.

EDIT (ED)

Before data in the signed-packed-decimal or unsigned-packed-decimal format can be used in a printed report, digits and signs must be converted to printable characters. Moreover, punctuation marks, such as commas and decimal points, may have to be inserted in appropriate places. The highly flexible EDIT instruction performs these functions in a single instruction execution.

This example shows step-by-step one way that the EDIT instruction can be used. The field to be edited (the source) is four bytes long; it is edited against a pattern 13 bytes long. The following symbols are used:

Symbol	Meaning
b (Hexadecimal 40)	Blank character
((Hexadecimal 21)	Significance starter
d (Hexadecimal 20)	Digit selector

Assume that register 12 contains:

```
00 00 10 00
```

and that the source and pattern fields are:

Source

1200			1203
02	57	42	6C

+

Pattern

1000												100C
40	20	20	6B	20	21	20	4B	20	20	40	C3	D9
b	d	d	,	d	(	d	.	d	d	b	C	R

Execution of the instruction:

Machine Format

Op Code	L	B_1	D_1	B_2	D_2
DE	0C	C	000	C	200

Assembler Format

```
Op Code  D1(L,B1),D2(B2)
-------------------------------
   ED    0(13,12),X'200'(12)
```

alters the pattern field as follows:

Pattern	Digit	Significance Indicator (Before/ After)	Rule	Location 1000-100C

b		off/off	leave(1)	bdd,d(d.ddbCR
d	0	off/off	fill	bbd,d(d.ddbCR
d	2	off/on(2)	digit	bb2,d(d.ddbCR
,		on/on	leave	same
d	5	on/on	digit	bb2,5(d.ddbCR
(	7	on/on	digit	bb2,57d.ddbCR
d	4	on/on	digit	bb2,574.ddbCR
.		on/on	leave	same
d	2	on/on	digit	bb2,574.2dbCR
d	6+	on/off(3)	digit	bb2,574.26bCR
b		off/off	fill	same
C		off/off	fill	bb2,574.26bbR
R		off/off	fill	bb2,574.26bbb

Notes:
1. This character is the fill byte.
2. First nonzero decimal source digit turns on significance indicator.
3. Plus sign in the four rightmost bits of the byte turns off significance indicator.

Thus, after the instruction is executed, the pattern field contains the result as follows:

Pattern

1000												100C
40	40	F2	6B	F5	F7	F4	4B	F2	F6	40	40	40
b	b	2	,	5	7	4	.	2	6	b	b	b

This pattern field prints as:

```
2,574.26
```

The source field remains unchanged. Condition code 2 is set because the number was greater than zero.

If the number in the source field is changed to the negative number 00 00 02 6D and the original pattern is used, the edited result this time is:

Pattern

1000												100C
40	40	40	40	40	40	F0	4B	F2	F6	40	C3	D9
b	b	b	b	b	b	0	.	2	6	b	C	R

This pattern field prints as:

```
0.26 CR
```

The significance starter forces the significance indicator to the on state and hence causes a leading zero and the decimal point to be preserved. Because the minus-sign code has no effect on the significance indicator, the characters CR are printed to show a negative (credit) amount.

Condition code 1 is set (number less than zero).

EDIT AND MARK (EDMK)

The EDIT AND MARK instruction may be used, in addition to the functions of EDIT, to insert a currency symbol, such as a dollar sign, at the appropriate position in the edited result. Assume the same source in storage locations 1200-1203, the same pattern in locations 1000-100C, and the same contents of general register 12 as for the EDIT instruction above. The previous contents of general register 1 (GR1) are not significant; a LOAD ADDRESS instruction is used to set up the first digit position that is forced to print if no significant digits occur to the left.

The instructions:

LA	1,6(0,12)	Load address of forced significant digit into GR1
EDMK	0(13,12),X‘200’(12)	Leave address of first significant digit in GR1
BCTR	1,0	Subtract 1 from address in GR1
MVI	0(1),C ‘$’	Store dollar sign at address in GR1

produce the following results for the two examples under EDIT:

Pattern

1000												100C
40	5B	F2	6B	F5	F7	F4	4B	F2	F6	40	40	40
b	$	2	,	5	7	4	.	2	6	b	b	b

This pattern field prints as:

```
$2,574.26
```

Condition code 2 is set to indicate that the number edited was greater than zero.

Pattern

1000												100C
40	40	40	40	40	5B	F0	4B	F2	F6	40	C3	D9
b	b	b	b	b	$	0	.	2	6	b	C	R

This pattern field prints as:

```
$0.26 CR
```

Condition code 1 is set because the number is less than zero.

MULTIPLY DECIMAL (MP)

Assume that the signed-packed-decimal number in storage locations 1202-1204 (the multiplicand) is to be multiplied by the signed-packed-decimal number in locations 500-501 (the multiplier).

	1202	1204
Multiplicand	38	46 0D

	500	501
Multiplier	32	1D

The multiplicand must first be extended to have at least two bytes of leftmost zeros, corresponding to the multiplier length, so as to avoid a data exception during the multiplication. ZERO AND ADD can be used to move the multiplicand into a longer field. Assume:

Register 4 contains 00 00 12 00.

Register 6 contains 00 00 05 00.

Then execution of the instruction:

```
ZAP X'100'(5,4),2(3,4)
```

sets up a new multiplicand in storage locations 1300-1304:

	1300				1304
Multiplicand (new)	00	00	38	46	0D

Now, after the instruction:

Machine Format

Op Code	L_1	L_2	B_1	D_1	B_2	D_2
FC	4	1	4	100	6	000

Assembler Format

```
Op Code  D1(L1,B1),D2(L2,B2)
-------------------------------
   MP  X'100'(5,4),0(2,6)
```

is executed, storage locations 1300-1304 contain the product: 01 23 45 66 0C.

SHIFT AND ROUND DECIMAL (SRP)

The SHIFT AND ROUND DECIMAL (SRP) instruction can be used for shifting decimal numbers in storage to the left or right. When a number is shifted right, rounding can also be done.

Decimal Left Shift

In this example, the contents of storage location FIELD1 are shifted three places to the left, effectively multiplying the contents of FIELD1 by 1000. FIELD1 is six bytes long. The following instruction performs the operation:

Machine Format

Op Code	L_1	I_3	S_1	B_2	D_2
F0	5	0	* * * *	0	003

Assembler Format

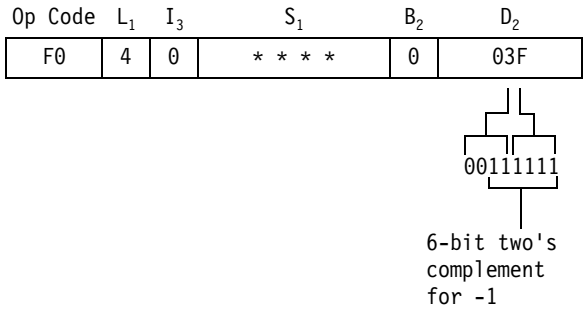
```
Op Code  S1(L1),S2,I3
-----------------------
  SRP   FIELD1(6),3,0

FIELD1 (before):  00 01 23 45 67 8C

FIELD1 (after):   12 34 56 78 00 0C
```

The second-operand address in this instruction specifies the shift amount (three places). The rounding digit, I_3, is not used in a left shift, but it must be a valid decimal digit. After execution, condition code 2 is set to show that the result is greater than zero.

Decimal Right Shift

In this example, the contents of storage location FIELD2 are shifted one place to the right, effectively dividing the contents of FIELD2 by 10 and discarding the remainder. FIELD2 is five bytes in length. The following instruction performs this operation:

Machine Format



Assembler Format

```
Op Code  S1(L1),S2,I3
-------------------------
  SRP    FIELD2(5),64-1,0

FIELD 2 (before):  01 23 45 67 8C

FIELD 2 (after):   00 12 34 56 7C
```

In the SRP instruction, shifts to the right are specified in the second-operand address by negative shift values, which are represented as a six-bit value in two's complement form.

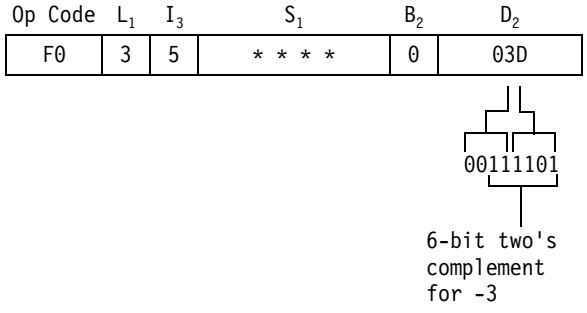
The six-bit two's complement of a number, n, can be specified as 64 - n. In this example, a right shift of one is represented as 64 - 1.

Condition code 2 is set.

Decimal Right Shift and Round

In this example, the contents of storage location FIELD3 are shifted three places to the right and rounded, in effect dividing by 1000 and rounding up. FIELD3 is four bytes in length.

Machine Format



Assembler Format

```
Op Code  S1(L1),S2,I3
-------------------------
  SRP    FIELD3(4),64-3,5

FIELD 3 (before):  12 39 60 0D

FIELD 3 (after):   00 01 24 0D
```

The shift amount (three places) is specified in the D_2 field. The I_3 field specifies a rounding digit of 5. The rounding digit is added to the last digit shifted out (which is a 6), and the carry is propagated to the left. The sign is ignored during the addition.

Condition code 1 is set because the result is less than zero.

Multiplying by a Variable Power of 10

Since the shift value specified by the SRP instruction specifies both the direction and amount of the shift, the operation is equivalent to multiplying the decimal first operand by 10 raised to the power specified by the shift value.

If the shift value is to be variable, it may be specified by the B_2 field instead of the displacement D_2 of the SRP instruction. The general register designated by B_2 should contain the shift value (power of 10) as a signed binary integer.

A fixed scale factor modifying the variable power of 10 may be specified by using both the B_2 field (variable part in a general register) and the D_2 field (fixed part in the displacement).

The SRP instruction uses only the rightmost six bits of the effective address $D_2(B_2)$ and interprets them as a six-bit signed binary integer to control the left or right shift as in the preceding shift examples.

ZERO AND ADD (ZAP)

Assume that the signed-packed-decimal number at storage locations 4500-4502 is to be moved to locations 4000-4004 with four leading zeros in the result field. Also assume:

Register 9 contains 00 00 40 00.

Storage locations 4000-4004 contain 12 34 56 78 90.

Storage locations 4500-4502 contain 38 46 0D.

After the instruction:

Machine Format

Op Code	L_1	L_2	B_1	D_1	B_2	D_2
F8	4	2	9	000	9	500

Assembler Format

```
Op Code  D1(L1,B1),D2(L2,B2)
ZAP      0(5,9),X'500'(3,9)
```

is executed, the storage locations 4000-4004 contain 00 00 38 46 0D; condition code 1 is set to indicate a negative result without overflow.

Note that, because the first operand is not checked for valid sign and digit codes, it may contain any combination of hexadecimal digits before the operation.

Hexadecimal-Floating-Point Instructions

(See Chapter 9, “Floating-Point Overview and Support Instructions” for a complete description of the hexadecimal-floating-point instructions.)

In this section, the abbreviations FPR0, FPR2, FPR4, and FPR6 stand for floating-point registers 0, 2, 4, and 6 respectively.

ADD NORMALIZED (AD, ADR, AE, AER, AXR)

The ADD NORMALIZED instruction performs the addition of two HFP numbers and places the normalized result in a floating-point register. Neither of the two numbers to be added must necessarily be in normalized form before addition occurs. For example, assume that:

FPR6 contains the unnormalized number C3 08 21 00 00 00 00 00 = -82.1_{16} = -130.06_{10} approximately.

Storage locations 2000-2007 contain the normalized number 41 12 34 56 00 00 00 00 = $+1.23456_{16}$ = $+1.14_{10}$ approximately.

Register 13 contains 00 00 20 00.

The instruction:

Machine Format

Op Code	R_1	X_2	B_2	D_2
7A	6	0	D	000

Assembler Format

```
Op Code  R1,D2(X2,B2)
AE       6,0(0,13)
```

performs the short-precision addition of the two operands, as follows.

The characteristics of the two numbers (43 and 41) are compared. Since the number in storage has a characteristic that is smaller by 2, it is right-shifted two hexadecimal digit positions. One guard digit is retained on the right. The fractions of the two numbers are then added algebraically:

```
                                Fraction GD[1]
FPR6                          -43 08 21 00
Shifted number from storage   +43 00 12 34  5
-------------------------------------------------
Intermediate sum              -43 08 0E CB  B
Left-shifted sum              -42 80 EC BB
```

[1] Guard digit

Because the intermediate sum is unnormalized, it is left-shifted to form the normalized HFP number $-80.ECBB_{16}$ = -128.92_{10} approximately. Combining the sign with the characteristic, the result is C2 80 EC BB, which replaces the left half of FPR6. The right half of FPR6 and the contents of storage locations 2000-2007 are unchanged. Condition code 1 is set to indicate a result less than zero.

If the long-precision instruction AD were used, the result in FPR6 would be C2 80 EC BA A0 00 00 00. Note that use of the long-precision instruction would avoid a loss of precision in this example.

ADD UNNORMALIZED (AU, AUR, AW, AWR)

The ADD UNNORMALIZED instruction operates the same as the ADD NORMALIZED instruction, except

that the final result is not normalized. For example, using the same operands as in the example for ADD NORMALIZED, when the short-precision instruction:

Machine Format

Op Code	R_1	X_2	B_2	D_2
7E	6	0	D	000

Assembler Format

```
Op Code  R1,D2(X2,B2)
-------------------
   AU    6,0(0,13)
```

is executed, the two numbers are added as follows:

```
                                Fraction GD[1]
FPR6                           -43 08 21 00
Shifted number from storage    +43 00 12 34  5
-------------------------------------------
Intermediate sum               -43 08 0E CB  B
```

[1] Guard digit

The guard digit participates in the addition but is discarded. The unnormalized sum replaces the left half of FPR6. Condition code 1 is set because the result is less than zero.

The truncated result in FPR6 (C3 08 0E CB 00 00 00 00) shows a loss of a significant digit when compared to the result of short-precision normalized addition.

COMPARE (CD, CDR, CE, CER)

Assume that FPR4 contains 43 00 00 00 00 00 00 00 (zero), and FPR6 contains 35 12 34 56 78 9A BC DE (a positive number). The contents of the two registers are to be compared using a long-precision COMPARE instruction.

Machine Format

Op Code	R_1	R_2
29	4	6

Assembler Format

```
Op Code  R1,R2
--------------
  CDR    4,6
```

The number with the smaller characteristic, which is in register FPR6, is right-shifted 43 - 35 hex (67 - 53 decimal) or 14 digit positions, so that the two characteristics agree. The shifted number is 43 00 00 00 00 00 00 00, with a guard digit of one. Therefore, when the two numbers are compared, condition code 1 is set, indicating that operand 1 in FPR4 is less than operand 2 in FPR6.

If the example is changed to a second operand with a characteristic of 34 instead of 35, so that FPR6 contains 34 12 34 56 78 9A BC DE, the operand is right-shifted 15 positions, leaving all fraction digits and the guard digit as zeros. Condition code 0 is set, indicating equality. This example shows that two HFP numbers with different characteristics or fractions may compare equal if the numbers are unnormalized or zero.

As another example of comparing unnormalized HFP numbers, 41 00 12 34 56 78 9A BC compares equal to all numbers of the form 3F 12 34 56 78 9A BC 0X (X represents any hexadecimal digit). When the COMPARE instruction is executed, the two rightmost digits are shifted right two places, the 0 becomes the guard digit, and the X does not participate in the comparison.

However, when two normalized HFP numbers are compared, the relationship between numbers that compare equal is unique: each digit in one number must be the same as the corresponding digit in the other number.

DIVIDE (DD, DDR, DE, DER)

Assume that the first operand (the dividend) is in FPR2 and the second operand (the divisor) in FPR0. If the operands are in the short-precision format, the resulting quotient is returned to FPR2 by the instruction:

Machine Format

Op Code	R_1	R_2
3D	2	0

Assembler Format

```
Op Code  R1,R2
--------------
  DER    2,0
```

Several examples of short-precision HFP division, with the dividend in FPR2 and the divisor in FPR0, are shown below. For case A, the result, which replaces the dividend, is obtained in the following steps.

```
                 7.2522F
          ------------
.123400 | .821000
          7F6C00
          -------
           2A400 0
           24680 0
           -------
            5D80 00
            5B04 00
            -------
             27C 000
             246 800
             -------
              35 8000
              24 6800
              -------
              11 18000
              11 10C00
              --------
                  7400
```

Case	FPR2 Before (Dividend)	FPR0 (Divisor)	FPR2 After (Quotient)
A	-43 082100	+43 001234	-42 72522F
B	+42 101010	+45 111111	+3D F0F0F0
C	+48 30000F	+41 400000	+47 C0003C
D	+48 30000F	+41 200000	+48 180007
E	+48 180007	+41 200000	+47 C00038

Case C shows a number being divided by 4.0. Case D divides the same number by 2.0, and case E divides the result of case D again by 2.0. The results of cases C and E differ in the rightmost hexadecimal digit position, which illustrates an effect of result truncation.

HALVE (HDR, HER)

HALVE produces the same result as HFP DIVIDE with a divisor of 2.0. Assume FPR2 contains the long-precision number +48 30 00 00 00 00 00 0F. The following HALVE instruction produces the result +48 18 00 00 00 00 00 07 in FPR2:

Machine Format

Op Code	R_1	R_2
24	2	2

Assembler Format

Op Code	R_1,R_2
HDR	2,2

MULTIPLY (MD, MDR, MDE, MDER, MXD, MXDR, MXR)

For this example, the following long-precision operands are in FPR0 and FPR2:

```
FPR0:  -33 606060 60606060
FPR2:  -5A 200000 20000020
```

A long-precision product is generated by the instruction:

Machine Format

Op Code	R_1	R_2
2C	0	2

Assembler Format

Op Code	R_1,R_2
MDR	0,2

If the operands were not already normalized, the instruction would first normalize them. It then generates an intermediate result consisting of the full 28-digit hexadecimal product fraction obtained by multiplying the 14-digit hexadecimal operand fractions, together with the appropriate sign and a characteristic that is the sum of the operand characteristics less 64 (40 hex):

The fraction multiplication is performed as follows:

```
                .60606060606060
                .20000020000020
                ---------------
                C0C0C0C0C0C0C00
          C0C0C0C0C0C0C0
    C0C0C0C0C0C0C0
  ------------------------------
.0C0C0C18181824181818 0C0C0C00
```

Attaching the sign and characteristic to the fraction gives:

```
+4D 0C0C0C 18181824 1818180C 0C0C00
```

Because this intermediate product has a leading zero, it is then normalized. The truncated final result placed in FPR0 is:

```
+4C C0C0C1 81818241
```

Hexadecimal-Floating-Point-Number Conversion

The following examples illustrate one method of converting between binary fixed-point numbers (32-bit signed binary integers) and normalized HFP numbers. Conversion must provide for the different representations used with negative numbers: the two's-complement form for signed binary integers, and the signed-absolute-value form for the fractions of HFP numbers.

Fixed Point to Hexadecimal Floating Point

The method used here inverts the leftmost bit of the 32-bit signed binary integer, which is equivalent to adding 2^{31} to the number and considering the result to be positive. This changes the number from a signed integer in the range 2^{31} - 1 through -2^{31} to an unsigned integer in the range 2^{32} - 1 through 0. After conversion to the long HFP format, the value 2^{31} is subtracted again.

Assume that general register 9 (GR9) contains the integer -59 in two's-complement form:

```
GR9:     FF FF FF C5
```

Further, assume two eight-byte fields in storage: TEMP, for use as temporary storage, and TWO31, which contains the floating-point constant 2^{31} in the following format:

```
TWO31:   4E 00 00 00 80 00 00 00
```

This is an unnormalized long HFP number with the characteristic 4E, which corresponds to a radix point (hexadecimal point) to the right of the number.

The following instruction sequence performs the conversion:

		Result
X	9,TWO31+4	GR9: 7FFF FFC5
ST	9,TEMP+4	TEMP: xxxx xxxx 7FFF FFC5
MVC	TEMP(4),TWO31	TEMP: 4E00 0000 7FFF FFC5
LD	2,TEMP	FPR2: 4E00 0000 7FFF FFC5
SD	2,TWO31	FPR2: C23B 0000 0000 0000

The EXCLUSIVE OR (X) instruction inverts the leftmost bit in general register 9, using the right half of the constant as the source for a leftmost one bit. The next two instructions assemble the modified number in an unnormalized long HFP format, using the left half of the constant as the plus sign, the characteristic, and the leading zeros of the fraction. LOAD (LD) places the number unchanged in floating-point register 2. The SUBTRACT NORMALIZED (SD) instruction performs the final two steps by subtracting 2^{31} in HFP form and normalizing the result.

Hexadecimal Floating Point to Fixed Point

The procedure described here consists basically in reversing the steps of the previous procedure. Two additional considerations must be taken into account. First: the HFP number may not be an exact integer. Truncating the excess hexadecimal digits on the right requires shifting the number one digit position farther to the right than desired for the final result, so that the units digit occupies the position of the guard digit. Second: the HFP number may have to be tested as to whether it is outside the range of numbers representable as a 32-bit signed binary integer.

Assume that floating-point register 6 contains the number $59.25_{10} = 3B.4_{16}$ in normalized form:

```
FPR6:    42 3B 40 00 00 00 00 00
```

Further, assume three eight-byte fields in storage: TEMP, for use as temporary storage, and the constants 2^{32} (TWO32) and 2^{31} (TWO31R) in the following formats:

```
TWO32:   4E 00 00 01 00 00 00 00
TWO31R:  4F 00 00 00 08 00 00 00
```

The constant TWO31R is shifted right one more position than the constant TWO31 of the previous example, so as to force the units digit into the guard-digit position.

The following instruction sequence performs the integer truncation, range tests, and conversion to a signed binary integer in general register 8 (GR8):

		Result
SD	6,TWO31R	FPR6: C87F FFFF C500 0000
BC	11,OVERFLOW	Branch to overflow routine if result is greater than or equal to zero
AW	6,TWO32	FPR6: 4E00 0000 8000 003B
BC	4,OVERFLOW	Branch to overflow routine if result is less than zero
STD	6,TEMP	TEMP: 4E00 0000 8000 003B
XI	TEMP+4,X‘80’	TEMP: 4E00 0000 0000 003B
L	8,TEMP+4	GR8: 0000 003B

The SUBTRACT NORMALIZED (SD) instruction shifts the fraction of the number to the right until it lines up with TWO31R, which causes the fraction digit 4 to fall to the right of the guard digit and be lost; the result of subtracting 2^{31} from the remaining digits is renormalized. The result should be less than zero; if not, the original number was too large in the positive direction. The first BRANCH ON CONDITION (BC) performs this test.

The ADD UNNORMALIZED (AW) instruction adds 2^{32}: 2^{31} to correct for the previous subtraction and another 2^{31} to change to an all-positive range. The second BC tests for a result less than zero, showing that the original number was too large in the negative direction. The unnormalized result is placed in temporary storage by the STORE (STD) instruction. There the leftmost bit of the binary integer is inverted by the EXCLUSIVE OR (XI) instruction to subtract 2^{31} and thus convert the unsigned number to the signed format. The final result is loaded into GR8.

Multiprogramming and Multiprocessing Examples

When two or more programs sharing common storage locations are being executed concurrently in a multiprogramming or multiprocessing environment, one program may, for example, set a flag bit in the common-storage area for testing by another program. It should be noted that the instructions AND (NI, NIY, or NC), EXCLUSIVE OR (XI, XIY, or XC), and OR (OI, OIY, or OC) could be used to set flag bits in a multiprogramming environment; but when the interlocked-access facility 2 is not installed, the same instructions may cause program logic errors in a multiprocessing configuration where two or more CPUs can fetch, modify, and store data in the same storage locations simultaneously.

Example of a Program Failure Using OR Immediate

The following example assumes that the interlocked-access facility 2 is not installed.

Assume that two independent programs try to set different bits to one in a common byte in storage. The following example shows how the use of the instruction OR immediate (OI) can fail to accomplish this, if the programs are executed simultaneously on two different CPUs. One of the possible error situations is depicted.

Execution of instruction OI FLAGS,X’01’ on CPU A	FLAGS	Execution of instruction OI FLAGS,X’80’ on CPU B
	X’00’	Fetch FLAGS X’00’
Fetch FLAGS X’00’	X’00’	
	X’00’	OR X’80’ into X’00’
OR X’01’ into X’00’	X’00’	
	X’80’	Store X’80’ into FLAGS
Store X’01’ into FLAGS	X’01’	
FLAGS should have value of X’81’ following both updates.		

The problem shown here is that the value stored by the OI instruction executed on CPU A overlays the value that was stored by CPU B. The X‘80’ flag bit was erroneously turned off, and the data is now invalid.

The COMPARE AND SWAP instruction has been provided to overcome this and similar problems.

When the interlocked-access facility 2 is installed, the update performed by OR (OI) instruction is an interlocked-update reference, and the problem illustrated here does not occur.

Conditional Swapping Instructions (CS, CDS)

The COMPARE AND SWAP (CS) and COMPARE DOUBLE AND SWAP (CDS) instructions can be used in multiprogramming or multiprocessing environments to serialize access to counters, flags, control words, and other common storage areas.

The following examples of the use of the COMPARE AND SWAP and COMPARE DOUBLE AND SWAP instructions illustrate the applications for which the instructions are intended. It is important to note that these are examples of functions that can be performed by programs while the CPU is enabled for interruption (multiprogramming) or by programs that are being executed in a multiprocessing configuration. That is, the routine allows a program to modify the contents of a storage location while the CPU is enabled, even though the routine may be interrupted by another program on the same CPU that will update the location, and even though the possibility exists that another CPU may simultaneously update the same location.

The COMPARE AND SWAP instruction first checks the value of a storage location and then modifies it only if the value is what the program expects; normally this would be a previously fetched value. If the value in storage is not what the program expects, then the location is not modified; instead, the current value of the location is loaded into a general register, in preparation for the program to loop back and try again. During the execution of COMPARE AND SWAP, no other CPU can perform a store access or interlocked-update access at the specified location.

To ensure successful updating of a common storage field by two or more CPUs, all updates must be done by means of an interlocked-update reference. See the programming notes of COMPARE AND SWAP for an example of how COMPARE AND SWAP can be unsuccessful due to an OR IMMEDIATE instruction executed by another CPU.

Setting a Single Bit

The following instruction sequence shows how the COMPARE AND SWAP instruction can be used to set a single bit in storage to one. Assume that the first byte of a word in storage called “WORD” contains eight flag bits.

```
      LA  6,X'80'  Put bit to be ORed into GR6
      SLL 6,24     Shift left 24 places to
                     align the byte to be ORed
                     with the location of the
                     flag bits within WORD
      L   7,WORD   Fetch current flag values
RETRY LR  8,7      Load flags into GR8
      OR  8,6      Set bit to one
      CS  7,8,WORD Store new flags if current
                     flags unchanged, or re-
                     fetch current flag values
                     if changed
      BC  4,RETRY  If new flags are not stored,
                     try again
```

The format of the COMPARE AND SWAP instruction is:

Machine Format

Op Code	R_1	R_3	S_2
BA	7	8	* * * *

Assembler Format

```
Op Code  R1,R3,S2
-----------------
   CS    7,8,WORD
```

The COMPARE AND SWAP instruction compares the first operand (general register 7 containing the current flag values) to the second operand in storage (WORD) while no CPU other than the one executing the COMPARE AND SWAP instruction is permitted to perform a store access or interlocked-update access at the specified storage location.

If the comparison is successful, indicating that the flag bits have not been changed since they were fetched, the modified copy in general register 8 is stored into WORD. If the flags have been changed, the compare will not be successful, and their new values are loaded into general register 7.

The conditional branch (BC) instruction tests the condition code and reexecutes the flag-modifying instructions if the COMPARE AND SWAP instruction

indicated an unsuccessful comparison (condition code 1). When the COMPARE AND SWAP instruction is successful (condition code 0), the flags contain valid data, and the program exits from the loop.

The branch to RETRY will be taken only if some other program modifies the contents of WORD. This type of a loop differs from the typical “bit-spin” loop. In a bit-spin loop, the program continues to loop until the bit changes. In this example, the program continues to loop only if the value does change during each iteration. If a number of CPUs simultaneously attempt to modify a single location by using the sample instruction sequence, one CPU will fall through on the first try, another will loop once, and so on until all CPUs have succeeded.

Updating Counters

In this example, a 32-bit counter is updated by a program using the COMPARE AND SWAP instruction to ensure that the counter will be correctly updated. The original value of the counter is obtained by loading the word containing the counter into general register 7. This value is moved into general register 8 to provide a modifiable copy, and general register 6 (containing an increment to the counter) is added to the modifiable copy to provide the updated counter value. The COMPARE AND SWAP instruction is used to ensure valid storing of the counter.

The program updating the counter checks the result by examining the condition code. The condition code 0 indicates a successful update, and the program can proceed. If the counter had been changed between the time that the program loaded its original value and the time that it executed the COMPARE AND SWAP instruction, the execution would have loaded the new counter value into general register 7 and set the condition code to 1, indicating an unsuccessful update. The program must then repeat the update sequence until the execution of the COMPARE AND SWAP instruction results in a successful update.

The following instruction sequence performs the above procedure:

```
     LA  6,1      Put increment (1) into GR6
     L   7,CNTR   Put original counter value
                    into GR7
LOOP LR  8,7      Set up copy in GR8 to modify
     AR  8,6      Increment copy
     CS  7,8,CNTR Update counter in storage
     BC  4,LOOP   If original value had
                    changed, update new value
```

The following shows two CPUs, A and B, executing this instruction sequence simultaneously: both CPUs attempt to add one to CNTR.

CPU A			CPU B		
GR7	GR8	CNTR	GR7	GR8	Comments
		16			
16	16				CPU A loads GR7 and GR8 from CNTR
			16	16	CPU B loads GR7 and GR8 from CNTR
				17	CPU B adds one to GR8
	17				CPU A adds one to GR8
		17			CPU A executes CS; successful match, store
			17		CPU B executes CS; no match, GR7 changed to CNTR value
				18	CPU B loads GR8 from GR7, adds one to GR8
		18			CPU B executes CS; successful match, store

Bypassing Post and Wait

See the *MVS Programming: Authorized Assembler Services Guide* (SA22-7608) for details on bypassing the z/OS WAIT and POST routines.

Lock/Unlock

When a common storage area larger than a doubleword is to be updated, it is usually necessary to provide special interlocks to ensure that a single program at a time updates the common area. Such an area is called a serially reusable resource (SRR).

In general, updating a list, or even scanning a list, cannot be safely accomplished without first “freezing” the list. However, the COMPARE AND SWAP and COMPARE DOUBLE AND SWAP instructions can be used in certain restricted situations to perform queuing and list manipulation. Of prime importance is the capability to perform the lock/unlock functions and to provide sufficient queuing to resolve contentions, either in a LIFO or FIFO manner. The lock/unlock functions can then be used as the interlock mechanism for updating an SRR of any complexity.

The lock/unlock functions are based on the use of a "header" associated with the SRR. The header is the common starting point for determining the states of the SRR, either free or in use, and also is used for queuing requests when contentions occur. Contentions are resolved using WAIT and POST. The general programming technique requires that the program that encounters a "locked" SRR must "leave a mark on the wall" indicating the address of an ECB on which it will WAIT. The "unlocking" program sees the mark and posts the ECB, thus permitting the waiting program to continue. In the two examples given, all programs using a particular SRR must use either the LIFO queuing scheme or the FIFO scheme; the two cannot be mixed. When more complex queuing is required, it is suggested that the queue for the SRR be locked using one of the two methods shown.

Lock/Unlock with LIFO Queuing for Contentions

The header consists of a word, that is, a four-byte field aligned on a word boundary. The word can contain zero, a positive value, or a negative value.

- A zero value indicates that the serially reusable resource (SRR) is free.
- A negative value indicates that the SRR is in use but no additional programs are waiting for the SRR.
- A positive value indicates that the SRR is in use and that one or more additional programs are waiting for the SRR. Each waiting program is identified by an element in a chained list. The positive value in the header is the address of the element most recently added to the list.

Each element consists of two words. The first word is used as an ECB; the second word is used as a pointer to the next element in the list. A negative value in a pointer indicates that the element is the last element in the list. The element is required only if the program finds the SRR locked and desires to be placed in the list.

The following chart describes the action taken for LIFO LOCK and LIFO UNLOCK routines. The routines following the chart allow enabled code to perform the actions described in the chart.

	Action		
Function	**Header Contains Zero**	**Header Contains Positive Value**	**Header Contains Negative Value**
LIFO LOCK (the incoming element is at location A)	SRR is free. Set the header to a negative value. Use the SRR.	SRR is in use. Store the contents of the header into location A+4. Store address A into the header. WAIT; the ECB is at location A.	
LIFO UNLOCK	Error	Some program is waiting for the SRR. Move the pointer from the "last in" element into the header. POST; the ECB is in the "last in" element.	The list is empty. Store zeros into the header. The SRR is free.

LIFO LOCK Routine:

Initial Conditions:

- GR1 contains the address of the incoming element.
- GR2 contains the address of the header.

```
LLOCK  SR     3,3       GR3 = 0
       ST     3,0(1)    Initialize the ECB
       LNR    0,1       GR0 = a negative value
TRYAGN CS     3,0,0(2)  Set the header to a
                           negative value if the
                           header contains zeros
       BC     8,USE     Did the header contain
                           zeros?
       ST     3,4(1)    No, store the value of
                           the header into the
                           pointer in the in-
                           coming element
       CS     3,1,0(2)  Store the address of
                           the incoming element
                           into the header
       LA     3,0(0)    GR3 = 0
       BC     7,TRYAGN  Did the header get up-
                           dated?
       WAIT   ECB=(1)   Yes, wait for the re-
                           source; the ECB is in
                           the incoming element
USE    [Any instruction]
```

LIFO UNLOCK Routine:

Initial Conditions:

- GR2 contains the address of the header.

```
LUNLK L    1,0(2)    GR1 = the contents of
                       the header
A     LTR  1,1       Does the header contain
      BC   4,B         a negative value?
      L    0,4(1)    No, load the pointer
      CS   1,0,0(2)    from the 'last in'
                       element and store it in
                       the header
      BC   7,A       Did the header get
                       updated?
      POST (1)       Yes, post the 'last in'
                       element
      BC   15,EXIT   Continue
B     SR   0,0       The header contains a
      CS   1,0,0(2)    negative value; free
      BC   7,A         the header and continue
EXIT  [Any instruction]
```

Note that the LOAD instruction L 1,0(2) at location LUNLK would have to be CS 1,1,0(2) if it were not for the rule concerning storage-operand consistency. This rule requires the LOAD instruction to fetch a four-byte operand aligned on a word boundary such that, if another CPU changes the word being fetched by an operation which is also at least word-consistent, either the entire new or the entire old value of the word is obtained, and not a combination of the two. (See “Storage-Operand Consistency” on page 5-125.)

Lock/Unlock with FIFO Queuing for Contentions

The header always contains the address of the most recently entered element. The header is originally initialized to contain the address of a posted ECB. Each program using the serially reusable resource (SRR) must provide an element regardless of whether contention occurs. Each program then enters the address of the element which it has provided into the header, while simultaneously it removes the address previously contained in the header. Thus, associated with any particular program attempting to use the SRR are two elements, called the “entered element” and the “removed element” The “entered element” of one program becomes the “removed element” for the immediately following program. Each program then waits on the removed element, uses the SRR, and then posts the entered element.

When no contention occurs, that is, when the second program does not attempt to use the SRR until after the first program is finished, then the POST of the first program occurs before the WAIT of the second program. In this case, the bypass-post and bypass-wait routines described in the preceding section are applicable. For simplicity, these two routines are shown only by name rather than as individual instructions.

In the example, the element need be only a single word, that is, an ECB. However, in actual practice, the element could be made larger to include a pointer to the previous element, along with a program identification. Such information would be useful in an error situation to permit starting with the header and chaining through the list of elements to find the program currently holding the SRR.

It should be noted that the element provided by the program remains pointed to by the header until the next program attempts to lock. Thus, in general, the entered element cannot be reused by the program. However, the removed element is available, so each program gives up one element and gains a new one. It is expected that the element removed by a particular program during one use of the SRR would then be used by that program as the entry element for the next request to the SRR.

It should be noted that, since the elements are exchanged from one program to the next, the elements cannot be allocated from storage that would be freed and reused when the program ends. It is expected that a program would obtain its first element and release its last element by means of the routines described in “Free-Pool Manipulation”.

The following chart describes the action taken for FIFO LOCK and FIFO UNLOCK.

Function	Action
FIFO LOCK (the incoming element is at location A)	Store address A into the header. WAIT; the ECB is at the location addressed by the old contents of the header.
FIFO UNLOCK	POST; the ECB is at location A.

The following routines allow enabled code to perform the actions described in the previous chart.

FIFO Lock Routine:

Initial conditions:

- GR3 contains the address of the header.
- GR4 contains the address, A, of the element currently owned by this program. This element becomes the entered element.

```
FLOCK  LR   2,4       GR2 now contains
                        address of element to
                        be entered
       SR   1,1       GR1 = 0
       ST   1,0(2)    Initialize the ECB
       L    1,0(3)    GR1 = contents of the
                        header
TRYAGN CS   1,2,0(3)  Enter address A into
                        header while remember-
       BC   7,TRYAGN    ing old contents of
                        header into GR1; GR1
                        now contains address
                        of removed element
       LR   4,1       Removed element becomes
                        new currently owned
                        element
       HSWAIT         Perform bypass-wait
                        routine; if ECB
                        already posted, con-
                        tinue; if not, wait;
                        GR1 contains the
                        address of the ECB
USE    [Any instruction]
```

FIFO Unlock Routine:

Initial conditions:

- GR2 contains the address of the removed element, obtained during the FLOCK routine.
- GR5 contains 40 00 00 00_{16}

```
FUNLK  LR  1,2  Place address of entered
                  element in GR1; GR1 = ad-
                  dress of ECB to be posted
       SR  0,0  GR0 = 0; GR0 has a post code
                  of zero
       OR  0,5  Set bit 1 of GR0 to one
       HSPOST   Perform bypass-post routine;
                  if ECB has not been waited
                  on, then mark posted and
                  continue; if it has been
                  waited on, then post
CONTINUE [Any instruction]
```

Free-Pool Manipulation

It is anticipated that a program will need to add and delete items from a free list without using the lock/unlock routines. This is especially likely since the lock/unlock routines require storage elements for queuing and may require working storage. The lock/unlock routines discussed previously allow simultaneous lock routines but permit only one unlock routine at a time. In such a situation, multiple additions and a single deletion to the list may all occur simultaneously, but multiple deletions cannot occur at the same time. In the case of a chain of pointers containing free storage buffers, multiple deletions along with additions can occur simultaneously. In this case, the removal cannot be done using the COMPARE AND SWAP instruction without a certain degree of exposure.

Consider a chained list of the type used in the example in section “Lock/Unlock with LIFO Queuing for Contentions” on page A-48. Assume that the first two elements are at locations A and B, respectively. If one program attempted to remove the first element and was interrupted between the fourth and fifth instructions of the LUNLK routine, the list could be changed so that elements A and C are the first two elements when the interrupted program resumes execution. The COMPARE AND SWAP instruction would then succeed in storing the value B into the header, thereby destroying the list.

The probability of the occurrence of such list destruction can be reduced to *near* zero by appending to the header a counter that indicates the number of times elements have been added to the list. The use of a 32-bit counter guarantees that the list will not be destroyed unless the following events occur, in the exact sequence:

1. An unlock routine is interrupted between the fetch of the pointer from the first element and the update of the header.
2. The list is manipulated, including the deletion of the element referenced in 1, and exactly 2^{32} (or an integer multiple of 2^{32}) additions to the list are performed. Note that this takes on the order of days to perform in any practical situation.
3. The element referenced in 1 is added to the list.
4. The unlock routine interrupted in 1 resumes execution.

The following routines use such a counter in order to allow multiple, simultaneous additions and removals at the head of a chain of pointers.

The list consists of a doubleword header and a chain of elements. The first word of the header contains a pointer to the first element in the list. The second word of the header contains a 32-bit counter indicating the number of additions that have been made to the list. Each element contains a pointer to the next element in the list. A zero value indicates the end of the list.

The following chart describes the free-pool-list manipulation.

Function	Action	
	Header = 0,Count	Header = A,Count
ADD TO LIST (the incoming element is at location A)	Store the first word of the header into location A. Store the address A into the first word of the header. Decrement the second word of the header by one.	
DELETE FROM LIST	The list is empty.	Set the first word of the header to the value of the contents of location A. Use element A.

The following routines allow enabled code to perform the free-pool-list manipulation described in the above chart.

ADD TO FREE LIST Routine:

Initial Conditions:

- GR2 contains the address of the element to be added.
- GR4 contains the address of the header.

```
ADDQ   LM   0,1,0(4) GR0,GR1 = contents of
                       the header
TRYAGN ST   0,4(2)   Point the new element to
                       the top of the list
       LR   3,1      Move the count to GR3
       AHI  3,1      Increment the count
       CDS  0,2,0(4) Update the header
       BC   7,TRYAGN
```

DELETE FROM FREE LIST Routine:

Initial conditions:

- GR4 contains the address of the header.

```
DELETQ LM   2,3,0(4)    GR2,GR3 = contents
                          of the header
TRYAGN LTR  2,2         Is the list empty?
       BC   8,EMPTY     Yes, get help
       L    0,4(2)      No, GR0 = the pointer
                          from the first
                          element
       LR   1,3         Move the count to GR1
       CDS  2,0,0(4)    Update the header
       BC   7,TRYAGN
USE    [Any instruction] The address of the re-
                          moved element is in
                          GR2
```

Notes:

1. The LM (LOAD MULTIPLE) instructions at locations ADDQ and DELETQ would have to be CDS (COMPARE DOUBLE AND SWAP) instructions if it were not for the rule concerning storage-operand consistency. This rule requires the LOAD MULTIPLE instructions to fetch an eight-byte operand aligned on a doubleword boundary such that, if another CPU changes the doubleword being fetched by an operation which is also at least doubleword-consistent, either the entire new or the entire old value of the doubleword is obtained, and not a combination of the two. (See “Storage-Operand Consistency” on page 5-125.)
2. The add-to-free-list and delete-from-free-list examples shown above assume that the pointer to the next element in the list is in the second word of the element, as described “Lock/Unlock with LIFO Queuing for Contentions” on page A-48.

PERFORM LOCKED OPERATION (PLO)

The PERFORM LOCKED OPERATION instruction can be used in a multiprogramming or multiprocessing environment to perform compare, load, compare-and-swap, and store operations on two or more discontiguous locations that can be words or doublewords. The operations are performed as an atomic set of operations under the control of a lock that is held only for the duration of the execution of a single PERFORM LOCKED OPERATION instruction, as opposed to across the execution of multiple instructions. Since lock contention is resolved by the CPU and is very brief, the program need not include a method for dealing with the case when the lock to be

used is held by a program being executed by another CPU. Also, there need be no concern that the program may be interrupted while it holds a lock, since PERFORM LOCKED OPERATION will complete its operation and release its lock before an interruption can occur.

PERFORM LOCKED OPERATION can be thought of as performing concurrent interlocked updates of multiple operands. However, the instruction does not actually perform any interlocked update, and a serially reusable resource cannot be updated predictably through the use of both PERFORM LOCKED OPERATION and conditional-swapping instructions (CS and CDS).

Following is an example of how PERFORM LOCKED OPERATION can be used to add an element at the beginning of a queue.

Assume the following variables associated with the queue: S, which is a sequence number that is incremented anytime the queue is changed; H (for head), which is the address of the first element on the queue; and C, which is a count of the number of elements on the queue. Assume a queue element contains a variable, F (for forward), which is the address of the next element on the queue. If a new element, N, is to be enqueued at the head of the queue, that can be done by setting F in N to H and then performing the following atomic set of operations:

S+1 → S
A(N) → H
C+1 → C

where A(N) is the address of N.

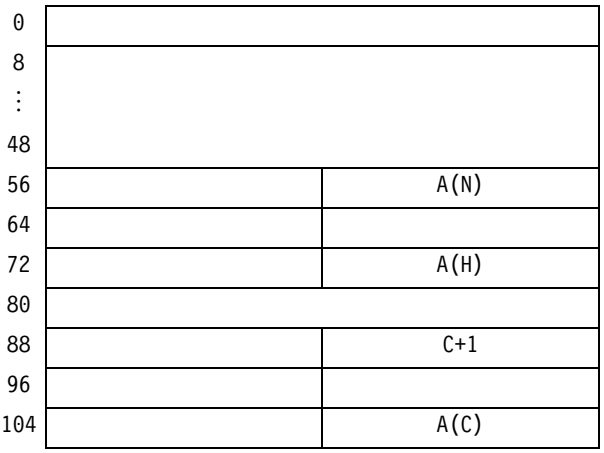
The enqueueing of N can be done by means of the following steps:

1. Obtain consistent values of S, H, and C, meaning obtain S and obtain the H and C that are consistent with that value of S.
2. Store H in N.F.
3. By means of PLO.csdst (PERFORM LOCKED OPERATION performing compare and swap and double store), with S as the swap variable and H and C as the store variables, add one to S, set H to A(N), and add one to C, provided that S still has the value obtained in step 1. If S has already been changed, go back to step 1.

Consistent values of S, H, and C cannot necessarily be obtained simply by using three LOAD instructions because a PERFORM LOCKED OPERATION instruction being executed by another CPU may have completed an update of S but not yet of H or C. In this case, the three LOAD instructions will obtain the new S but the old H or C. However, as will be described, it may be possible to use three LOAD instructions.

If S is obtained while holding the lock, meaning by means of PERFORM LOCKED OPERATION, then H and C can be obtained by LOAD instructions since no other CPU can subsequently change H or C without changing S, as observed when the lock is held.

The parameter list used by the PLO.csdst is as follows, assuming the access-register mode is not used:



The program is as follows:

```
      LA    RT,H        Initialize addresses in
                          PL (T = temp)
      ST    RT,PL+76    Op4 address (address of
                          H)
      LA    RT,C
      ST    RT,PL+108   Op6 address (address of
                          C)
      LA    RN,N       Address of N
      ST    RN,PL+60   Initialize op3 in PL
                         (address of N)
      LA    R1,S       PLT address = address
                         of S
-----------------------------------------------
      SR    RS,RS      Dummy S.  CC1 will
                         probably be set
      SR    R0,R0      Function code 0
                         (compare and load)
```

```
      PLO   RS,S,RS,S Obtain S while holding
                      lock
-------------------------------------------
      LA    R0,16     Function code 16 (csdst)
LOOP  L     RT,H      Consistent H
      ST    RT,OFSTF(,RN)  OFSTF = offset of
                           F in N
      L     RT,C      Consistent C
      LA    RT,1(,RT) C+1
      ST    RT,PL+92  Initialize op5 in PL
                      (C+1)
      LA    RSP,1(,RS) RS/RSP = even-odd pair.
                      S+1 in RSP
      PLO   RS,S,0,PL
      BNZ   LOOP      Br if S changed (if CC
                      not 0)
```

Note the following about the first PERFORM LOCKED OPERATION instruction (PLO.cl). If S is not zero (which is probably true), S (the second operand, op2) is loaded into RS (the first-operand comparison value, op1c). If S is zero, S (the fourth operand, op4) is loaded into RS (the third operand, op3). Either of these loads occurs while the lock is held. It is unnecessary to test the condition code to determine which load occurred.

The above program may be a simplification. If the queue has associated with it a variable, T (for tail), that is the address of the last element on the queue, and the queue is currently empty, T also must be set when N is added to the queue. This would require a different program using a compare-and-swap-and-triple-store operation.

If the queue is added to, deleted from, and rearranged by means of PERFORM LOCKED OPERATION instructions in which the sequence number, S, is always the second operand, then, since the definition of PERFORM LOCKED OPERATION specifies that the second operand is always stored last, the first PERFORM LOCKED OPERATION instruction in the above program can be replaced by a LOAD instruction. The three instructions within the dashed lines would be replaced by L RS,S.

Sorting Instructions

Tree Format

Two instructions, COMPARE AND FORM CODEWORD and UPDATE TREE, refer to a tree — a data structure with a specific format. A tree consists of some number (always odd) of consecutively numbered nodes. Node 1 is the root of the tree. Every node except the root has one parent node in the same tree. Every parent node has two son nodes. Every even-numbered node is the *leftson* of its parent node, and every odd-numbered node (except node 1) is the *rightson* of its parent node. Division by two (ignoring remainder) of the node number gives the parent node number. Nodes with sons are also called internal nodes, and nodes without sons are called terminal nodes. Figure A-5 on page A-54 illustrates schematically a 21-node tree with arrows drawn from each parent node to each son node.

A tree is used for merging several sorted sequences of records into a single merged sequence of records. At each step in the merging process, there exists the initial part of the merged sequence and the remaining parts of each of the sorted sequences that are being merged. Each step consists in selecting the lowest record (the record with the lowest key when sorting in ascending sequence) from all of the as yet unmerged parts of the sorted sequences and adding it to the merged sequence. Each terminal node in the tree represents one of the sorted sequences. The number of internal nodes in the tree is one less than the number of sorted sequences. Each internal node conceptually contains one record from each of the sorted sequences but one; these are the lowest records, from all but one of the sorted sequences, that have not yet been added to the merged sequence. In addition, there is the lowest record from the one remaining sorted sequence. This additional record is compared and interchanged with nodes of the tree to select the record to be added next to the merged sequence. This processing begins with the parent of the terminal node that represents the one remaining sorted sequence, and it continues from that node along the path to the root of the tree. The selected record emerges from the root of the tree.

The tree may perhaps be most easily explained by considering each node to represent a comparison operation in an “elimination tournament” to find the lowest record. After the tournament has been com-

pleted, each node has an associated “loser” record which had a higher key in the comparison represented by that node. Besides a loser record at each node, there is one record (the “winner”) which is not associated with any node since it never compared high. The next step would be to introduce a new record from the same sorted sequence from which the winner record originated and replay the tournament with the new record in place of the former winner. It can be seen that it is unnecessary to do all the comparisons represented by all the nodes in the tree — most of them are unaffected by the new record replacing the former winner. In fact, it is sufficient to redo only those node comparisons in which the former winner record participated. Each new record is inserted into the tree at the terminal node that represents the sorted sequence containing the record. The use of the tree assumes that programming provides a method of remembering at which terminal node each winning record originated. The instruction UPDATE TREE allows for a new record to be inserted at a terminal node and the tree to be updated so that a new winner record is left in the general registers.

Rather than comparing the actual keys of records, much of the merge logic can be performed using “codewords” to represent a record key rather than referring to actual keys. The value of a codeword at a node in the tree depends not only on the record’s key but also on the key of the winning record in the last comparison at that node. The codeword consists of two parts:

1. Bits 16-31 contain the one’s complement of the first halfword in which the record key differs from that of the node’s winning record.
2. Bits 0-15 specify the byte offset of the halfword in this record’s key just beyond the halfword value (complemented) in bit positions 16-31.

When comparing records in the path of the last winner record, if the new record is also represented by a codeword resulting from a comparison with the last winner, all codewords in the update path are with respect to the same winner. When comparing such codewords, a high codeword represents a low key and vice versa. Thus, when codewords are unequal, a node entry with a high codeword (representing a low actual key) should move up the tree.

In the case of a tie value of codewords, it is necessary to refer to the actual keys. This is done by the instruction COMPARE AND FORM CODEWORD, which resolves the ambiguity and computes a new codeword for the high-key (loser) record.

The eight bytes at each node of a tree consist of (1) a codeword for this record, computed with respect to the last record which compared low against this record and (2) a parameter usable to locate this record, for example, a direct or indirect address.

The instruction UPDATE TREE is so defined that tree updating stops after equal codewords are detected and the tie-breaking instruction COMPARE AND FORM CODEWORD can be used, after which UPDATE TREE can resume tree updating at the point where equal codewords were previously found.

COMPARE AND FORM CODEWORD may alternatively be used for merging in descending sequence. In that case, bits 16-31 of the codeword at a node contain the true value of the first halfword in which the record key differs from that of the node’s winning record. When the descending option of COMPARE AND FORM CODEWORD is used, the higher of two codewords represents the higher key.

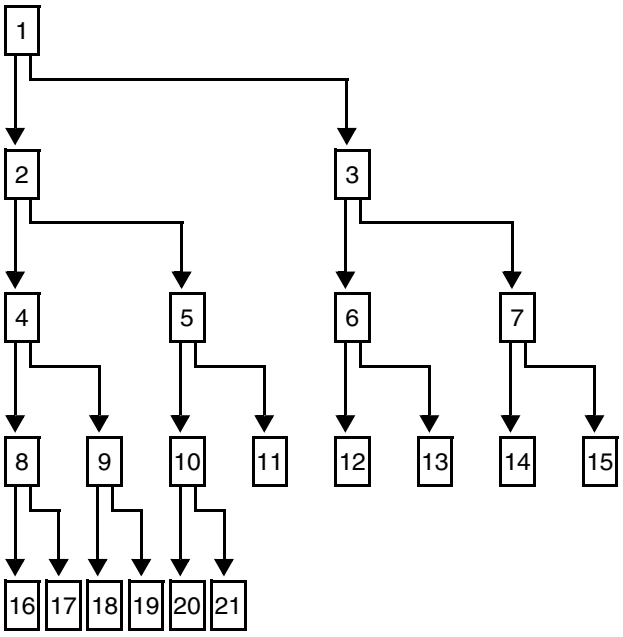


Figure A-5. Schematic Diagram of Merge Control Tree with 21 Nodes

Example of Use of Sort Instructions

An example illustrates how the instructions UPDATE TREE and COMPARE AND FORM CODEWORD may be used in the merge operation within a sort program. A five-way merge requires a tree data structure with four internal nodes and five terminal-node positions. The schematic diagram shown later in this section illustrates such a tree, containing four internal nodes (not counting the dummy node) and five input sequences for a merge, one sequence at each terminal-node position. Each record in an input sequence in the diagram is indicated by its address. The actual record contents are shown in Figure A-7 on page A-58. Each record contains 16 bytes, consisting of the following fields:

Byte Offset (hexadecimal)	Field
0-5	Six-byte record key.
6-7	Halfword node index specifying the input sequence of the next record of this input sequence.
8-B	Address of the next record in the same input sequence.
C-F	This chaining field is initially zero. At the completion of the merge, this field is to contain the address of the next record in the merged sequence.

The merge process forms a single sorted sequence from five input sequences, each of which is in sorted order. This process can be subdivided into three steps:

1. A priming step takes the first record from each of the five input sequences and places them in the tree data structure. For each record to be introduced into the tree, first its codeword value is computed with respect to the lowest possible key value of all zeros. This codeword, with a second word which contains the address of the actual record, forms a doubleword node value that can be placed at the appropriate node. After priming, the node values, one each from each of the five input sequences, will have been placed in the tree so that each of the four internal nodes contains one node value and the node value for a winner record has emerged from the root of the tree.

2. After each winner emerges from the tree, the main merge process is performed repeatedly. Each iteration introduces the node value for one new record into the tree and produces a node value for a new winner record. The tree plus the winner must at all times contain precisely one node value from each input sequence being merged. Therefore, the new node value that is introduced into the tree on each iteration must come from the same input sequence from which the winner node value in the preceding iteration originated.

3. When the node value for the last record of an input sequence emerges as a winner, there is no successor record from that input sequence to be introduced into the tree on the next iteration. Hence, the order of the merge must be reduced by one for each such occurrence. This runout process will consist of one or more iterations for each of a four-way, three-way, two-way, and one-way merge. The onset of runout occurs in the example when it is found that the next input record from a sequence is lower than its predecessor (a sequence break).

The priming process is discussed next, and the state of the tree is shown after priming is complete. Then, a short program that uses the instructions UPDATE TREE and COMPARE AND FORM CODEWORD to perform the main merge is described. An abbreviated trace is then presented to show the status of the tree and certain general registers for 16 iterations of the main merge. The runout process is not discussed in this example.

Priming begins by forming the node value for the first record of each input sequence. The first word of the node value is the codeword formed by executing COMPARE AND FORM CODEWORD on a record key containing all binary zeros. The second word of the node value is the address of the record represented by that node value. The node values for the first record of each input sequence are:

```
Sequence Index          Node Values

      28          0006 FFFC 0000 1030
      30          0006 FFFB 0000 1040
      38          0006 FFFA 0000 1050
      40          0004 FFFE 0000 1080
      48          0006 FFF0 0000 1060
```

In the example, the tree data structure is assumed to have base address X‘1000’, which is kept in general register 4 (to match the expected use in UPDATE TREE). Similarly, internal-node index values and

input-sequence index values are always used from general register 5.

Although the tree-priming program is not part of this example, the UPDATE TREE instruction is used in creating it as follows. First, the codeword position for each internal node of the tree is initialized to all ones (X'FFFF FFFF'). This artifice fills the tree with dummy low records. Then, for each record in the table, (1) the sequence index is loaded into general register 5, (2) the node value is loaded into general registers 0 and 1, and (3) UPDATE TREE is executed. At the completion of this priming process, the tree-node contents in the example are as shown on line 0 of Figure A-9 on page A-60. The contents of the general registers are as shown on the first line of Figure A-8 on page A-59.

The figure illustrating the program for the main merge is divided into three groups of columns, containing the absolute program, the general-register trace, and the symbolic program. The first part of the program extends from symbolic locations L1 through L2; it introduces a new record into the tree and executes an UPDATE TREE instruction. If no tied codewords are encountered in UPDATE TREE, then the BRANCH ON CONDITION instruction following UPDATE TREE loops back to L1 to introduce the next record into the tree. This BRANCH ON CONDITION instruction is suitable for use when UPDATE TREE operates in accordance with either its method 1 (setting condition code 1) or its method 2 (setting condition code 3). (The preceding sentence applies to 370-XA. In ESA/370 and ESA/390, UPDATE TREE operates in accordance with only method 2, which is not to say that it cannot set condition code 1. Method 2, but not method 1, tests for the condition that sets condition code 3.)

If UPDATE TREE encounters tied codewords, then the UPDATE TREE instruction is completed, the subsequent BRANCH ON CONDITION instruction does not branch, and control falls through to the second part of the program, which handles entries with tied codewords. This part then branches back to UPDATE TREE at L2, which resumes the tree updating. It is possible for tied codewords to be encountered at any level in the tree (or indeed at all levels), so that the tied-codeword part of the program may be entered up to three times for each record introduced.

The general-register trace for the first part of the main merge shows the contents of the first seven general registers after each instruction is executed during the first iteration. Note that the merged-chain field (at 1140) serves as the anchor for the merged-chain address chain through the records. The trace shows only the lower half of certain general registers, whose upper half is always zero.

Figure A-9 on page A-60 gives an abbreviated trace of the entire main merge of 16 records. For each record introduced into the tree, there are one or more lines (always an odd number) given in the figure to show the tree updating, which results finally in a winner in GR0 and GR1. The first line for each record shows the values of GR5, GR2, and GR3 before the first or only execution of UPDATE TREE. For the even-numbered lines, the storage updating by UPDATE TREE of tree nodes is shown (read left to right to follow the order of swapping). For example, consider line 10 and the corresponding UPDATE TREE: since GR5 contains 28, the first storage node examined is 1010 (refer to the schematic diagram). Since the codeword in GR0 is 0004 FFFE (same as for GR2), which is less than that of the word at 1010 (0006 FFF0), the doubleword at 1010 is swapped with that in GR0 and GR1. A second comparison at 1008 in the same execution of UPDATE TREE causes another register-storage doubleword swap, which leaves the winner (record 1040) in GR1 at the completion of UPDATE TREE (see the column at the far right of Figure A-9 on page A-60).

When a codeword comparison is made which does not result in a tie or a swap (that is, when the storage-codeword value is low), an asterisk appears in the trace for that storage entry.

When equal codewords are found, the execution of UPDATE TREE is completed. The following line in each such case shows the result of the tied-codeword routine, which always stores a new codeword and may also store a new record address before branching back to L2 to execute UPDATE TREE again. In this line, the notation “loses” or “wins” means that the node loses or wins, respectively.

The tie-break trace part of Figure A-8 on page A-59 shows the treatment of the third record (that is, the first record for which UPDATE TREE encounters a tied codeword). This corresponds to line 31 in Figure A-9 on page A-60.

The following is a summary of the steps that are needed to use this example for verification purposes:

1. Initialize storage as follows:

a. 1008 through 102F from line 0 of Figure A-9 on page A-60

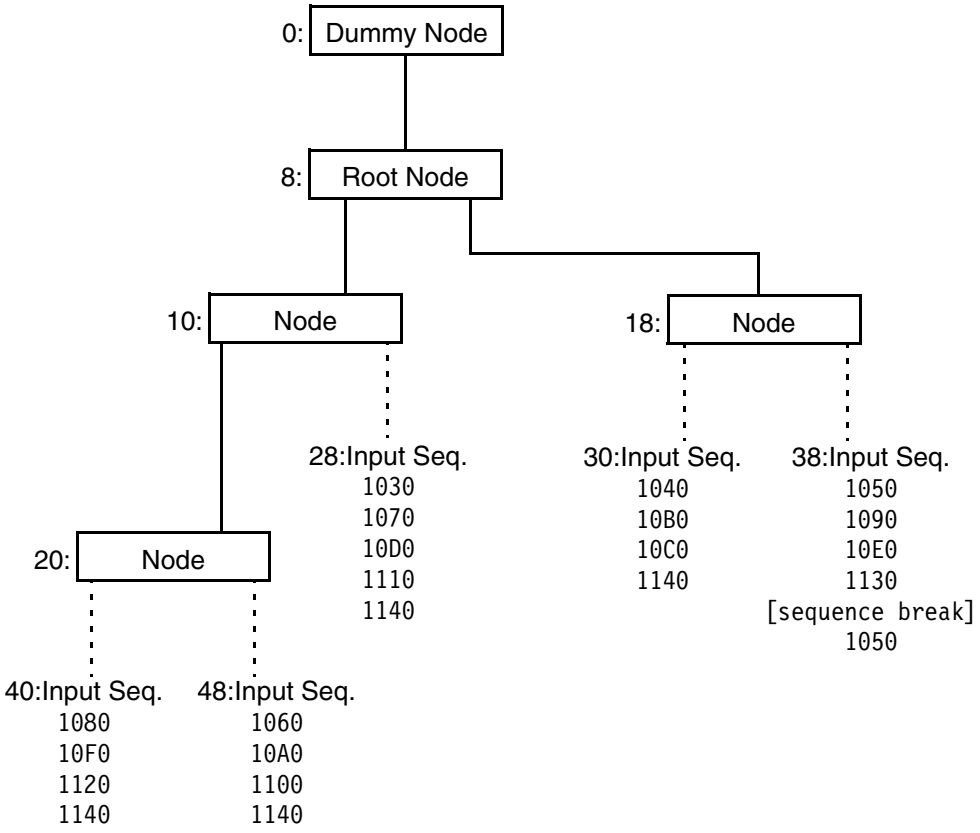
b. 1030 through 114F from Figure A-7 on page A-58

c. 1150 through 1189 from Figure A-8 on page A-59

2. Initialize GRs per first line in Figure A-8 and trace first record per Figure A-8.

3. Trace to completion of each UPT or BC 15,L2 (once for each line of Figure A-9). A detailed trace of the GRs for the tied-codeword part of line 31 of Figure A-9 is given in the lower part of Figure A-8.

4. Verify that addresses in the chain beginning at 103C and continuing through 114C are as shown in the right-hand column of Figure A-7.



Note: Each node and input sequence is identified by a number which is the hexadecimal node index. Each input sequence is given as a list of record addresses (also in hexadecimal).

Figure A-6. Schematic Diagram for Example of Merge to Be Performed

	Record Key at Hex Byte Offset						Successor Record						Merged-Chain Address			
							Index		Location							
Location	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1030	00	00	00	00	00	03	00	28	00	00	10	70	00	00	10	40
1040	00	00	00	00	00	04	00	30	00	00	10	B0	00	00	10	50
1050	00	00	00	00	00	05	00	38	00	00	10	90	00	00	10	60
1060	00	00	00	00	00	0F	00	48	00	00	10	A0	00	00	10	80
1070	00	00	00	01	FF	FF	00	28	00	00	10	D0	00	00	10	90
1080	00	00	00	01	FF	FF	00	40	00	00	10	F0	00	00	10	70
1090	00	00	00	00	FF	FF	00	38	00	00	10	E0	00	00	10	A0
10A0	00	00	00	00	FF	FF	00	48	00	00	11	00	00	00	10	B0
10B0	00	00	00	00	FF	FF	00	30	00	00	10	C0	00	00	10	C0
10C0	00	00	00	00	FF	FF	00	30	00	00	11	40	00	00	10	D0
10D0	00	00	00	01	00	00	00	28	00	00	11	10	00	00	10	E0
10E0	00	08	00	00	00	00	00	38	00	00	11	30	00	00	10	F0
10F0	00	08	00	00	00	02	00	40	00	00	11	20	00	00	11	00
1100	00	08	00	00	00	02	00	50	00	00	11	40	00	00	11	10
1110	00	08	00	00	00	03	00	00	00	00	11	40	00	00	11	20
1120	00	09	00	00	00	00	00	00	00	00	11	40	00	00	11	30
1130	FF	FF	FF	FF	FF	FE	00	38	00	00	10	50	00	00	00	00
1140	FF	FF	FF	FF	FF	FF	00	00	00	00	00	00	00	00	10	30

Figure A-7. Contents of Records to Be Merged

Absolute		General-Register Trace							Symbolic Program		
Loc	INSTR	GR0	GR1	GR2	GR3	GR4	GR5	GR6	Loc	Instruction	
		0006FFFC	1030		0000	1000	0000	1140		Using X'1000',4	
		'	'		'	'	'	'			
1150	5010600C	'	'		'	'	↓	'	L1	ST 1,12(,6)	Store merged-chain address
1154	48501006	'	'		'	'	0028	'		LH 5,6(,1)	Load node index of input
		'	'		↓	'	'	'			sequence of winner
1158	58301008	'	'		1070	'	'	↓		L 3,8(,1)	Load successor-record address
115C	1861	'	'		'	'	'	1030		LR 6,1	Save old winner address for
		'	'		'	'	'	'			next merged-chain store
115E	1B22	'	'	00000000	'	'	'	'		SR 2,2	Zero GR2 as initial offset
1160	B21A0004	'	'	0004FFFE	'	'	'	'		CFC 4	Compute codeword of new
		'	'	'	'	'	'	'			record based on new winner
1164	4720418A	'	↓	'	'	'	'	'		BC 2,L3	Exit on CC=2 (sequence break)
1168	1813	'	1070	'	'	'	'	'		LR 1,3 }	Move new record entry
		↓	'	'	'	'	'	'			to GRs 0-1
116A	1802	0004FFFE	↓	'	'	'	↓	'		LR 0,2 }	
116C	0102	0006FFFB	1040	'	'	'	0000	'	L2	UPT	Update tree data structure
116E	47504150	'	'	'	'	'	'	'		BC 5,L1	If no codeword tie found,
		↓	↓	↓	↓	'	↓	↓			branch to next iteration
						'			*	Fall through on tied codewords	
		00040000	1090	00040000	10B0	'	0018	1050	*	← GR values for tie-break trace	
		'	'		'	'	'	'			
1172	88200010	'	'	00000004	'	'	'	'		SRL 2,16	Shift codeword offset to
		'	'		'	'	'	'			initial offset position
		'	'		'	'	'	'			for CFC
1176	B21A0004	'	'	0006FFFD	'	'	'	'		CFC 4	Compute loser codeword
117A	50254000	'	'	[CC=1]	'	'	'	'		ST 2,0(5,4)	Store loser codeword in
		'	'	↓	'	'	'	'			current storage node
117E	47C0416C	↓	↓	branch	↓	'	↓	↓		BC 12,L2	Resume tree update if old
				taken		'					storage-node entry is loser
1182	50354004					'				ST 3,4(5,4)	Store loser record address
1186	47F0416C					'				BC 15,L2	Resume tree update
118A	...					↓			L3	...	Control reaches here at end

Figure A-8. Program for Main Merge

	General Regs after CFC at Location 1160				Storage Trace of Node Entries								General Regs after UPT or BC 15,L2	
L#	GR5	GR2	GR3	Comment	1020		1018		1010		1008		GR0	GR1
0[1]					0004FFFE	1080	0006FFFA	1050	0006FFF0	1060	0006FFFB	1040	0006FFFC	1030
10	28	0004FFFE	1070	No tie					0004FFFE	1070	0006FFF0	1060	0006FFFB	1040
20	30	00040000	10B0	No tie			00040000	10B0			*		0006FFFA	1050
30	38	00040000	1090	CC = 0			Tie						00040000	1090
31				Loses			0006FFFD						00040000	1090
32				No tie							00040000	1090	0006FFF0	1060
40	48	00040000	10A0	CC = 0	00040000	10A0			Tie				0004FFFE	1080
41				Equal					80001070				0004FFFE	1080
42				No tie							*		0004FFFE	1080
50	40	0002FF7F	10F0	No tie	0002FF7F	10F0			00040000	10A0	**		80001070	1070
60	28	0002FFFE	10D0	CC = 0					0002FFFE	10D0	Tie		00040000	10A0
61				Wins							0006FFFE	10A0	00040000	1090
62				No comp									00040000	1090
70	38	0002FF7F	10E0	No tie			0002FF7F	10E0			0006FFFD	10B0	0006FFFE	10A0
80	48	0002FF7F	1100	CC = 0	Tie								0002FF7F	1100
81				Wins	0006FFAF	1100							0002FF7F	10F0
82				No tie					0002FF7F	10F0	0002FFFE	10D0	0006FFFD	10B0
90	30	800010C0	10C0	No tie			*				*		800010C0	10C0
100	30	00020000	1140	No tie			00020000	1140			0002FF7F	10E0	0002FFFE	10D0
110	28	0002FF7F	1110	CC = 0					Tie				0002FF7F	1110
111				Wins					0004FFFC	1110			0002FF7F	10F0
112				CC = 0							Tie		0002FF7F	10F0
113				Wins							0004FFFD	10F0	0002FF7F	10E0
114				No comp									0002FF7F	10E0
120	38	00020000	1130	CC = 0			Tie						00020000	1130
121				Loses			00060000						00020000	1130
122				No tie							00020000	1130	0004FFFD	10F0
130	40	0002FF6F	1120	No tie	0002FF6F	1120			*		*		0006FFAF	1100
140	48	00020000	1140	No tie	00020000	1140			0002FF6F	1120	*		0004FFFC	1110
150	28	00020000	1140	No tie					00020000	1140	*		0002FF6F	1120
160	40	00020000	1140	CC = 0	Tie								00020000	1140
161				Equal	80001140								00020000	1140
162				CC = 0					Tie				00020000	1140
163				Equal					80001140				00020000	1140
164				CC = 0							Tie		00020000	1140
165				Wins							00060000	1140	00020000	1130
166				No comp									00020000	1130
170	38	00020000	1050	Branch										

Explanation:

[1] Line 0 shows the values in the tree after it is primed.

* Means no swap.

** Means no swap if UPDATE TREE method 1 is used or no examination if UPDATE TREE method 2 is used. Only method 2 is included in ESA/370 and ESA/390.

CC = 0 UPDATE TREE finds a tie and sets condition code 0.

Loses The tied-codeword routine finds that the node loses.

Wins The tied-codeword routine finds that the node wins.

Equal The tied-codeword routine finds that the keys are equal.

Branch Branches to terminate at 118A on sequence break.

No compNo compare.

Figure A-9. Abbreviated Trace of Main Merge Processing

Appendix B. Lists of Instructions

The following figures list instructions by name, mnemonic, and operation code. Some models may offer instructions that do not appear in the figures, such as those provided for assists or as part of special or custom features.

The operation code for the interpretive execution facility is not included in this appendix. See the publication *IBM System/370 Extended Architecture Interpretive Execution*, SA22-7095, for the operation code associated with that facility.

The operation code 00 hex with a two-byte instruction format is allocated for use by the program when an indication of an invalid operation is required. Operation code 00 hex will never be assigned to an instruction implemented in the CPU.

Explanation of Symbols in "Characteristics" Columns:

- $ — Causes serialization.
- ¢ — Causes serialization and checkpoint synchronization.
- £ — Causes specific-operand serialization.
- ¢[1] — Causes serialization and checkpoint synchronization when the M_1 and R_2 fields contain 1111 binary and 0000 binary, respectively. Causes only serialization when the fast-BCR-serialization facility is installed, and the M_1 and R_2 fields contain 1110 binary and 0000 binary, respectively.
- £[1] — Causes specific-operand serialization when the interlocked-access facility 1 is installed and the storage operand is aligned on an integral boundary corresponding to its size.
- ¤[1] — Restricted from transactional execution.
- ¢[2] — Causes serialization and checkpoint synchronization when the state entry to be unstacked is a program-call state entry.
- £[2] — Causes specific-operand serialization when the interlocked-access facility 2 is installed.
- ¤[2] — Restricted from transactional execution when R_2 nonzero and branch tracing is enabled.
- ¢[3] — Causes serialization and checkpoint synchronization when the set-key control is one.
- ¤[3] — Restricted from transactional execution when mode tracing is enabled.
- ¤[4] — Restricted from transactional execution when a monitor-event condition occurs.
- ¤[5] — Model dependent whether the instruction is restricted from transactional execution.
- ¤[6] — Restricted from transactional execution when the effective allow-AR-modification control is zero.
- ¤[7] — Restricted from transactional execution when the effective allow-floating-point-operation control is zero.
- ¤[8] — May be restricted from transactional execution depending on machine conditions.
- ¤[9] — Restricted in the constrained transactional-execution mode; a transaction-constraint program exception may be recognized. For STCMH, the instruction is restricted only when the M_3 field is zero.
- ¤[10] — Restricted to forward branches in the constrained transactional-execution mode.
- ¤[11] — For PFD and PFDRL, it is model dependent whether the instruction is restricted from transactional execution when the code in the M_1 field is 6 or 7; for STCMH, it is model dependent whether the instruction is restricted when the M_3 field is zero and the code in the R_1 field is 6 or 7.
- ¤[12] — Restricted from transactional execution when a guarded-storage event is recognized. When in the transactional-execution mode, the transaction is aborted, and the guarded-storage event is processed.
- A* — Access exceptions for logical addresses. The optional asterisk indicates that a PER ZAD event is not recognized.
- A[1]* — Access exceptions; not all access exceptions may occur; see instruction description for details. The optional asterisk indicates that a PER ZAD event is not recognized.
- AI — Access exceptions for instruction address.
- B — PER branch event. (For LGG and LLGFSG, the PER branch event is only recognized coincident with a guarded-storage event.)
- B† — B_1 field designates an access register when bit 47 of GR0 is zero, and bits 16-17 of the current PSW are 01 binary; or when bit 47 of GR0 is one, and bits 40-41 of GR0 are 01 binary.

© Copyright IBM Corp. 2000, 2017

B‡ B_2 field designates an access register when bit 63 of GR0 is zero, and bits 16-17 of the current PSW are 01 binary; or when bit 63 of GR0 is one, and bits 56-57 of GR0 are 01 binary.

B_1 B_1 field designates an access register in the access-register mode.

B_2 B_2 field designates an access register in the access-register mode.

B_4 B_4 field designates an access register in the access-register mode.

BP B_2 field designates an access register when PSW bits 16 and 17 have the value 01.

C Condition code is set.

C* Condition code is optionally set.

C^1 Condition code is set when the conditional-SSKE facility is installed, and either or both of the MR and MC bits are one.

CS Compare-and-swap-and-store facility.

CT Configuration-topology facility.

CX Constrained transactional-execution facility.

D2 DAT-enhancement facility 2.

Da AFP-register data exception.

Db BFP-instruction data exception.

Dc Compare-and-trap data exception.

DE DAT-enhancement facility.

DF Decimal-overflow exception.

Dg General-operand data exception.

DK Decimal-divide exception.

DM Depending on the model, DIAGNOSE may generate various program exceptions and may change the condition code.

DO Distinct-operands facility.

Dt DFP-instruction data exception.

Dv Vector-instruction data exception.

E E instruction format.

E2 Extended-translation facility 2.

E3 Extended-translation facility 3.

ED1 Enhanced-DAT facility 1.

ED2 Enhanced-DAT facility 2.

EH Execution-hint facility.

EI Extended-immediate facility.

EO HFP-exponent-overflow exception.

ES Expanded-storage facility.

ET Extract-CPU-time facility.

EU HFP-exponent-underflow exception.

EX Execute exception.

F Floating-point-extension facility.

FC Designation of access registers depends on the function code of the instruction.

FG FPR-GR-transfer facility.

FK HFP-floating-point-divide exception.

FL Store-facility-list-extended facility.

FS Floating-point-support-sign-handling facility.

G0 Instruction execution includes the implied use of general register 0.

G1 Instruction execution includes the implied use of general register 1.

G2 Instruction execution includes the implied use of general register 2.

G4 Instruction execution includes the implied use of general register 4.

GE General-instructions-extension facility.

GF Guarded-storage facility.

GM Instruction execution includes the implied use of multiple general registers.

GS Instruction execution includes the implied use of general register 1 as the subsystem-identification word.

HM HFP-multiply-add/subtract facility.

HW High-word facility.

I I instruction format.

I1 Access register 1 is implicitly designated in the access-register mode.

I4 Access register 4 is implicitly designated in the access-register mode.

IA Interlocked-access facility.

IC Condition code alternative to interruptible instruction.

IE IE instruction format.

IF Fixed-point-overflow exception.

II Interruptible instruction.

IK Fixed-point-divide exception.

IM Insert-reference-bits-multiple facility

L New condition code is loaded.

L1 Load/store-on-condition facility 1.

L2 Load/store-on-condition facility 2.

LD Long-displacement facility.

LS HFP-significance exception.

LT Load-and-trap facility.

LZ Load-and-zero-rightmost-byte facility.

M3 Message-security assist extension 3.

M4 Message-security assist extension 4.

M5 Message-security assist extension 5.

MD Designation of access registers in the access-register mode is model-dependent.

ME Monitor event.

MI1 Miscellaneous-instruction-extensions facility 1.

MI2 Miscellaneous-instruction-extensions facility 2.

MII MII instruction format.

MO Move-with-optional-specifications facility.

MS Message-security assist.

N Instruction is new in z/Architecture as compared to ESA/390.

N3 Instruction is new in z/Architecture and has been added to ESA/390.

OP Operand exception.

P Privileged-operation exception; also, restricted from transactional execution.

PA Processor-assist facility.

PC DFP packed-conversion facility.

PE Parsing-enhancement facility.

PF PFPO facility.

PK Population-count facility.

Q Privileged-operation exception for semiprivileged instructions; also, restricted from transactional execution.

R_1 R_1 field designates an access register in the access-register mode.

R_2 R_2 field designates an access register in the access-register mode.

R_3 R_3 field designates an access register in the access-register mode.

RA Reusable-ASN-and-LX facility.

RB Reset-reference-bits-multiple facility.

RI RI instruction format.

RIE RIE instruction format.

RIL RIL instruction format.

RIS RIS instruction format.

RM R_1 field designates an access register in the access-register mode, and access-register 1 also is used in the access-register mode.

RR RR instruction format.

RRD RRD instruction format.

RRE RRE instruction format.

RRF RRF instruction format.

RRS RRS instruction format.

RS RS instruction format.

RSI RSI instruction format.

RSL RSL instruction format.

RSY RSY instruction format.

RX RX instruction format.

RXE RXE instruction format.

RXF RXF instruction format.

RXY RXY instruction format.

S S instruction format.

SC Store-clock-fast facility.

SE Special operation, stack-empty, stack-specification, and stack-type exceptions.

SF Special-operation, stack-full, and stack-specification exceptions.

SI SI instruction format.

SIL SIL instruction format.

SIY SIY instruction format.

SMI SMI instruction format.

SO Special-operation exception.

SP Specification exception.

SQ HFP-square-root exception.

SS SS instruction format.

SSE SSE instruction format.

SSF SSF instruction format.

ST PER storage-alteration event. (For LGG and LLGFSG, the PER storage-alteration event is only recognized coincident with a guarded-storage event.)

SU PER store-using-real-address event.

SW Special-operation exception and space-switch event.

T Trace exceptions (which include trace table, addressing, and low-address protection).

TE Test-pending-external-interruption facility.

TF Decimal-floating-point facility.

TR Decimal-floating-point-rounding facility.

TS TOD-clock-steering facility.

TX Transactional-execution facility.

U Condition code is unpredictable.

U_1 R_1 field designates an access register unconditionally.

U_2 R_2 field designates an access register unconditionally.

UB R_1 and R_3 fields designate access registers unconditionally, and B_2 field designates an access register in the access-register mode.

UE HFP unnormalized-extensions facility.

VD Vector packed-decimal facility.

VF Vector facility for z/Architecture.

VRI VRI instruction format.

VRR VRR instruction format.

VRS VRS instruction format.

VRV VRV instruction format.

VRX VRX instruction format.

VSI VSI instruction format.

WE Space-switch event.

XF IEEE-exception-simulation facility.

Xg Simulated IEEE exception.

Xi IEEE invalid-operation data or vector-processing exception.

Xo IEEE overflow data or vector-processing exception.

Xq Quantum data exception (if the floating-point extension facility is installed).

Xu IEEE underflow data or vector-processing exception.

XX Execute-extension facility.

Xx IEEE inexact data or vector-processing exception.

Xz IEEE division-by-zero data or vector-processing exception.

Z^1 Additional exceptions and events for PROGRAM CALL (which include ASX-translation, EX-translation, LX-translation, PC-translation-specification, special-operation, stack-full, and stack-specification exceptions and space-switch event).

Z^2 Additional exceptions and events for PROGRAM TRANSFER (which include AFX-translation, ASX-translation, primary-authority, and special-operation exceptions and space-switch event).

Z^3 Additional exceptions for SET SECONDARY ASN (which include AFX translation, ASX translation, secondary authority, and special operation).

Z^4 Additional exceptions and events for PROGRAM RETURN (which include AFX-translation, ASX-translation, secondary-authority, special-operation, stack-empty, stack-operation, stack-specification, and stack-type exceptions and space-switch event).

Z^5 Additional exceptions for BRANCH AND STACK (which include special operation, stack full, and stack specification).

Z^6 Additional exceptions and events for PROGRAM TRANSFER WITH INSTANCE (which include AFX-translation, ASTE-instance, ASX-translation, primary-authority, special-operation, and subspace-replacement exceptions and space-switch event).

Z^7 Additional exceptions for SET SECONDARY ASN WITH INSTANCE (which include AFX translation, ASTE instance, ASX translation, secondary authority, special operation, and subspace replacement).

ZF Decimal-floating-point-zoned-conversion facility.

Instructions Arranged by Name

Name	Mnemonic	Characteristics																	Opcode	Page
ADD (32)	A	RX-a	C			A			IF								B_2		5A	7-26
ADD (32)	AR	RR	C						IF										1A	7-26
ADD (32)	ARK	RRF-a	C	DO					IF										B9F8	7-26
ADD (32)	AY	RXY-a	C	LD		A			IF								B_2		E35A	7-27
ADD (64)	AG	RXY-a	C	N		A			IF								B_2		E308	7-27
ADD (64)	AGR	RRE	C	N					IF										B908	7-26
ADD (64)	AGRK	RRF-a	C	DO					IF										B9E8	7-26
ADD (64←32)	AGF	RXY-a	C	N		A			IF								B_2		E318	7-27
ADD (64←32)	AGFR	RRE	C	N					IF										B918	7-26
ADD (extended BFP)	AXBR	RRE	C		α[7,9]		SP	Db	Xi		Xo	Xu	Xx						B34A	19-15
ADD (extended DFP)	AXTR	RRF-a	C	TF	α[7,9]		SP	Dt	Xi		Xo	Xu	Xx						B3DA	20-19
ADD (extended DFP)	AXTRA	RRF-a	C	F	α[7,9]		SP	Dt	Xi		Xo	Xu	Xx	Xq					B3DA	20-19
ADD (long BFP)	ADB	RXE	C		α[7,9]	A		Db	Xi		Xo	Xu	Xx				B_2		ED1A	19-15
ADD (long BFP)	ADBR	RRE	C		α[7,9]			Db	Xi		Xo	Xu	Xx						B31A	19-15
ADD (long DFP)	ADTR	RRF-a	C	TF	α[7,9]			Dt	Xi		Xo	Xu	Xx						B3D2	20-19
ADD (long DFP)	ADTRA	RRF-a	C	F	α[7,9]			Dt	Xi		Xo	Xu	Xx	Xq					B3D2	20-19
ADD (short BFP)	AEB	RXE	C		α[7,9]	A		Db	Xi		Xo	Xu	Xx				B_2		ED0A	19-15
ADD (short BFP)	AEBR	RRE	C		α[7,9]			Db	Xi		Xo	Xu	Xx						B30A	19-15
ADD DECIMAL	AP	SS-b	C		α[9]	A		Dg	DF						ST	B_1	B_2		FA	8-6
ADD HALFWORD (32←16)	AH	RX-a	C			A			IF								B_2		4A	7-28
ADD HALFWORD (32←16)	AHY	RXY-a	C	LD		A			IF								B_2		E37A	7-28
ADD HALFWORD (64←16)	AGH	RXY-a	C	MI2		A			IF								B_2		E338	7-28
ADD HALFWORD IMMEDIATE (32←16)	AHI	RI-a	C						IF										A7A	7-28
ADD HALFWORD IMMEDIATE (64←16)	AGHI	RI-a	C	N					IF										A7B	7-29
ADD HIGH (32)	AHHHR	RRF-a	C	HW					IF										B9C8	7-29
ADD HIGH (32)	AHHLR	RRF-a	C	HW					IF										B9D8	7-29
ADD IMMEDIATE (32)	AFI	RIL-a	C	EI					IF										C29	7-27
ADD IMMEDIATE (32←16)	AHIK	RIE-d	C	DO					IF										ECD8	7-27
ADD IMMEDIATE (32←8)	ASI	SIY	C	GE		A			IF	£[1]					ST	B_1			EB6A	7-27
ADD IMMEDIATE (64←16)	AGHIK	RIE-d	C	DO					IF										ECD9	7-27
ADD IMMEDIATE (64←32)	AGFI	RIL-a	C	EI					IF										C28	7-27
ADD IMMEDIATE (64←8)	AGSI	SIY	C	GE		A			IF	£[1]					ST	B_1			EB7A	7-27
ADD IMMEDIATE HIGH (32)	AIH	RIL-a	C	HW					IF										CC8	7-29
ADD LOGICAL (32)	AL	RX-a	C			A											B_2		5E	7-30
ADD LOGICAL (32)	ALR	RR	C																1E	7-30
ADD LOGICAL (32)	ALRK	RRF-a	C	DO															B9FA	7-30
ADD LOGICAL (32)	ALY	RXY-a	C	LD		A											B_2		E35E	7-30
ADD LOGICAL (64)	ALG	RXY-a	C	N		A											B_2		E30A	7-30
ADD LOGICAL (64)	ALGR	RRE	C	N															B90A	7-30
ADD LOGICAL (64)	ALGRK	RRF-a	C	DO															B9EA	7-30
ADD LOGICAL (64←32)	ALGF	RXY-a	C	N		A											B_2		E31A	7-30
ADD LOGICAL (64←32)	ALGFR	RRE	C	N															B91A	7-30
ADD LOGICAL HIGH (32)	ALHHHR	RRF-a	C	HW															B9CA	7-31
ADD LOGICAL HIGH (32)	ALHHLR	RRF-a	C	HW															B9DA	7-31
ADD LOGICAL IMMEDIATE (32)	ALFI	RIL-a	C	EI															C2B	7-30
ADD LOGICAL IMMEDIATE (64←32)	ALGFI	RIL-a	C	EI															C2A	7-30
ADD LOGICAL WITH CARRY (32)	ALC	RXY-a	C	N3		A											B_2		E398	7-31
ADD LOGICAL WITH CARRY (32)	ALCR	RRE	C	N3															B998	7-31
ADD LOGICAL WITH CARRY (64)	ALCG	RXY-a	C	N		A											B_2		E388	7-31
ADD LOGICAL WITH CARRY (64)	ALCGR	RRE	C	N															B988	7-31

Figure B-1. Instructions Arranged by Name (Part 1 of 23)

Name	Mne-monic	Characteristics													Op-code	Page
ADD LOGICAL WITH SIGNED IMMEDIATE (32←16)	ALHSIK	RIE-d	C	DO											ECDA	7-32
ADD LOGICAL WITH SIGNED IMMEDIATE (32←8)	ALSI	SIY	C	GE		A			£[1]				ST	B_1	EB6E	7-32
ADD LOGICAL WITH SIGNED IMMEDIATE (64←16)	ALGHSIK	RIE-d	C	DO											ECDB	7-32
ADD LOGICAL WITH SIGNED IMMEDIATE (64←8)	ALGSI	SIY	C	GE		A			£[1]				ST	B_1	EB7E	7-32
ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (32)	ALSIH	RIL-a	C	HW											CCA	7-33
ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (32)	ALSIHN	RIL-a		HW											CCB	7-33
ADD NORMALIZED (extended HFP)	AXR	RR	C		¤[7,9]		SP	Da	EU	EO	LS				36	18-8
ADD NORMALIZED (long HFP)	AD	RX-a	C		¤[7,9]	A		Da	EU	EO	LS			B_2	6A	18-8
ADD NORMALIZED (long HFP)	ADR	RR	C		¤[7,9]			Da	EU	EO	LS				2A	18-8
ADD NORMALIZED (short HFP)	AE	RX-a	C		¤[7,9]	A		Da	EU	EO	LS			B_2	7A	18-8
ADD NORMALIZED (short HFP)	AER	RR	C		¤[7,9]			Da	EU	EO	LS				3A	18-8
ADD UNNORMALIZED (long HFP)	AW	RX-a	C		¤[7,9]	A		Da		EO	LS			B_2	6E	18-9
ADD UNNORMALIZED (long HFP)	AWR	RR	C		¤[7,9]			Da		EO	LS				2E	18-9
ADD UNNORMALIZED (short HFP)	AU	RX-a	C		¤[7,9]	A		Da		EO	LS			B_2	7E	18-9
ADD UNNORMALIZED (short HFP)	AUR	RR	C		¤[7,9]			Da		EO	LS				3E	18-9
AND (32)	N	RX-a	C			A								B_2	54	7-33
AND (32)	NR	RR	C												14	7-33
AND (32)	NRK	RRF-a	C	DO											B9F4	7-33
AND (32)	NY	RXY-a	C	LD		A								B_2	E354	7-33
AND (64)	NG	RXY-a	C	N		A								B_2	E380	7-33
AND (64)	NGR	RRE	C	N											B980	7-33
AND (64)	NGRK	RRF-a	C	DO											B9E4	7-33
AND (character)	NC	SS-a	C		¤[9]	A							ST	B_1 B_2	D4	7-34
AND (immediate)	NI	SI	C			A			£[2]				ST	B_1	94	7-33
AND (immediate)	NIY	SIY	C	LD		A			£[2]				ST	B_1	EB54	7-34
AND IMMEDIATE (high high)	NIHH	RI-a	C	N											A54	7-34
AND IMMEDIATE (high low)	NIHL	RI-a	C	N											A55	7-34
AND IMMEDIATE (high)	NIHF	RIL-a	C	EI											C0A	7-34
AND IMMEDIATE (low high)	NILH	RI-a	C	N											A56	7-35
AND IMMEDIATE (low low)	NILL	RI-a	C	N											A57	7-35
AND IMMEDIATE (low)	NILF	RIL-a	C	EI											C0B	7-35
BRANCH AND LINK	BAL	RX-a			¤[9]								B		45	7-35
BRANCH AND LINK	BALR	RR			¤[2,9]				T				B		05	7-35
BRANCH AND SAVE	BAS	RX-a			¤[9]								B		4D	7-36
BRANCH AND SAVE	BASR	RR			¤[2,9]				T				B		0D	7-36
BRANCH AND SAVE AND SET MODE	BASSM	RR			¤[2,3,9]				T				B		0C	7-37
BRANCH AND SET AUTHORITY	BSA	RRE			Q	A[1]*		SO	T				B		B25A	10-7
BRANCH AND SET MODE	BSM	RR			¤[3,9]				T				B		0B	7-38
BRANCH AND STACK	BAKR	RRE			¤[1]	A[1]*		Z[5]	T				B	ST	B240	10-11
BRANCH IN SUBSPACE GROUP	BSG	RRE			¤[1]	A[1]*		SO	T				B	R_2	B258	10-13
BRANCH INDIRECT ON CONDITION	BIC	RXY-b		MI2	¤[9]	A							B	B_2	E347	7-39
BRANCH ON CONDITION	BC	RX-b			¤[9]								B		47	7-40
BRANCH ON CONDITION	BCR	RR			¤[9]				¢[1]				B		07	7-40
BRANCH ON COUNT (32)	BCT	RX-a			¤[9]								B		46	7-41
BRANCH ON COUNT (32)	BCTR	RR			¤[9]								B		06	7-41
BRANCH ON COUNT (64)	BCTG	RXY-a		N	¤[9]								B		E346	7-41
BRANCH ON COUNT (64)	BCTGR	RRE		N	¤[9]								B		B946	7-41
BRANCH ON INDEX HIGH (32)	BXH	RS-a			¤[9]								B		86	7-42
BRANCH ON INDEX HIGH (64)	BXHG	RSY-a		N	¤[9]								B		EB44	7-42
BRANCH ON INDEX LOW OR EQUAL (32)	BXLE	RS-a			¤[9]								B		87	7-42
BRANCH ON INDEX LOW OR EQUAL (64)	BXLEG	RSY-a		N	¤[9]								B		EB45	7-42
BRANCH PREDICTION PRELOAD	BPP	SMI		EH	¤[9]										C7	7-43

Figure B-1. Instructions Arranged by Name (Part 2 of 23)

Name	Mnemonic	Characteristics															Opcode	Page
BRANCH PREDICTION RELATIVE PRELOAD	BPRP	MII		EH	α[9]												C5	7-43
BRANCH RELATIVE AND SAVE	BRAS	RI-b			α[9]									B			A75	7-46
BRANCH RELATIVE AND SAVE LONG	BRASL	RIL-b		N3	α[9]									B			C05	7-46
BRANCH RELATIVE ON CONDITION	BRC	RI-c			α[10]									B			A74	7-46
BRANCH RELATIVE ON CONDITION LONG	BRCL	RIL-c		N3	α[10]									B			C04	7-46
BRANCH RELATIVE ON COUNT (32)	BRCT	RI-b			α[9]									B			A76	7-47
BRANCH RELATIVE ON COUNT (64)	BRCTG	RI-b		N	α[9]									B			A77	7-47
BRANCH RELATIVE ON COUNT HIGH (32)	BRCTH	RIL-b		HW	α[9]									B			CC6	7-47
BRANCH RELATIVE ON INDEX HIGH (32)	BRXH	RSI			α[9]									B			84	7-48
BRANCH RELATIVE ON INDEX HIGH (64)	BRXHG	RIE-e		N	α[9]									B			EC44	7-48
BRANCH RELATIVE ON INDEX LOW OR EQ. (32)	BRXLE	RSI			α[9]									B			85	7-48
BRANCH RELATIVE ON INDEX LOW OR EQ. (64)	BRXLG	RIE-e		N	α[9]									B			EC45	7-48
CANCEL SUBCHANNEL	XSCH	S	C		P			OP	¢	GS							B276	14-3
CHECKSUM	CKSM	RRE	C		α[9]	A	SP	IC								R_2	B241	7-49
CIPHER MESSAGE	KM	RRE	C	MS	α[5,9]	A	SP	IC		GM	I1		ST	R_1	R_2		B92E	7-53
CIPHER MESSAGE WITH AUTHENTICATION	KMA	RRF-b	C	M8	α[5,9]	A	SP	IC		GM	I1		ST	R_1	R_2	R_3	B929	7-78
CIPHER MESSAGE WITH CHAINING	KMC	RRE	C	MS	α[5,9]	A	SP	IC		GM	I1		ST	R_1	R_2		B92F	7-53
CIPHER MESSAGE WITH CIPHER FEEDBACK	KMF	RRE	C	M4	α[5,9]	A	SP	IC		GM	I1		ST	R_1	R_2		B92A	7-92
CIPHER MESSAGE WITH COUNTER	KMCTR	RRF-b	C	M4	α[5,9]	A	SP	IC		GM	I1		ST	R_1,R_2,R_3			B92D	7-107
CIPHER MESSAGE WITH OUTPUT FEEDBACK	KMO	RRE	C	M4	α[5,9]	A	SP	IC		GM	I1		ST	R_1	R_2		B92B	7-120
CLEAR SUBCHANNEL	CSCH	S	C		P			OP	¢	GS							B230	14-5
COMPARE (32)	C	RX-a	C			A										B_2	59	7-134
COMPARE (32)	CR	RR	C														19	7-134
COMPARE (32)	CY	RXY-a	C	LD		A										B_2	E359	7-134
COMPARE (64)	CG	RXY-a	C	N		A										B_2	E320	7-134
COMPARE (64)	CGR	RRE	C	N													B920	7-134
COMPARE (64←32)	CGF	RXY-a	C	N		A										B_2	E330	7-134
COMPARE (64←32)	CGFR	RRE	C	N													B930	7-134
COMPARE (extended BFP)	CXBR	RRE	C		α[7,9]		SP	Db	Xi								B349	19-17
COMPARE (extended DFP)	CXTR	RRE	C	TF	α[7,9]		SP	Dt	Xi								B3EC	20-22
COMPARE (extended HFP)	CXR	RRE	C		α[7,9]		SP	Da									B369	18-10
COMPARE (long BFP)	CDB	RXE	C		α[7,9]	A		Db	Xi							B_2	ED19	19-17
COMPARE (long BFP)	CDBR	RRE	C		α[7,9]			Db	Xi								B319	19-17
COMPARE (long DFP)	CDTR	RRE	C	TF	α[7,9]			Dt	Xi								B3E4	20-22
COMPARE (long HFP)	CD	RX-a	C		α[7,9]	A		Da								B_2	69	18-10
COMPARE (long HFP)	CDR	RR	C		α[7,9]			Da									29	18-10
COMPARE (short BFP)	CEB	RXE	C		α[7,9]	A		Db	Xi							B_2	ED09	19-17
COMPARE (short BFP)	CEBR	RRE	C		α[7,9]			Db	Xi								B309	19-17
COMPARE (short HFP)	CE	RX-a	C		α[7,9]	A		Da								B_2	79	18-10
COMPARE (short HFP)	CER	RR	C		α[7,9]			Da									39	18-10
COMPARE AND BRANCH (32)	CRB	RRS		GE	α[9]									B			ECF6	7-135
COMPARE AND BRANCH (64)	CGRB	RRS		GE	α[9]									B			ECE4	7-135
COMPARE AND BRANCH RELATIVE (32)	CRJ	RIE-b		GE	α[10]									B			EC76	7-135
COMPARE AND BRANCH RELATIVE (64)	CGRJ	RIE-b		GE	α[10]									B			EC64	7-136
COMPARE AND FORM CODEWORD	CFC	S	C		α[9]	A	SP	II		GM	I1						B21A	7-137
COMPARE AND REPLACE DAT TABLE ENTRY	CRDTE	RRF-b		ED2	P	A[1]	SP		$								B98F	10-17
COMPARE AND SIGNAL (extended BFP)	KXBR	RRE	C		α[7,9]		SP	Db	Xi								B348	19-18
COMPARE AND SIGNAL (extended DFP)	KXTR	RRE	C	TF	α[7,9]		SP	Dt	Xi								B3E8	20-23
COMPARE AND SIGNAL (long BFP)	KDB	RXE	C		α[7,9]	A		Db	Xi							B_2	ED18	19-18
COMPARE AND SIGNAL (long BFP)	KDBR	RRE	C		α[7,9]			Db	Xi								B318	19-18
COMPARE AND SIGNAL (long DFP)	KDTR	RRE	C	TF	α[7,9]			Dt	Xi								B3E0	20-23
COMPARE AND SIGNAL (short BFP)	KEB	RXE	C		α[7,9]	A		Db	Xi							B_2	ED08	19-18
COMPARE AND SIGNAL (short BFP)	KEBR	RRE	C		α[7,9]			Db	Xi								B308	19-18
COMPARE AND SWAP (32)	CS	RS-a	C		α[9]	A	SP		$				ST			B_2	BA	7-144
COMPARE AND SWAP (32)	CSY	RSY-a	C	LD	α[9]	A	SP		$				ST			B_2	EB14	7-144

Figure B-1. Instructions Arranged by Name (Part 3 of 23)

Name	Mne-monic	Characteristics												Op-code	Page
COMPARE AND SWAP (64)	CSG	RSY-a	C	N	α[9]	A	SP		$		ST		B_2	EB30	7-144
COMPARE AND SWAP AND PURGE (32)	CSP	RRE	C		P	A[1]	SP		$		ST		R_2	B250	10-21
COMPARE AND SWAP AND PURGE (64)	CSPG	RRE	C	DE	P	A[1]	SP		$		ST		R_2	B98A	10-21
COMPARE AND SWAP AND STORE	CSST	SSF	C	CS	α[1]	A	SP		$	GM	ST	B_1	B_2	C82	7-146
COMPARE AND TRAP (32)	CRT	RRF-c		GE				Dc						B972	7-149
COMPARE AND TRAP (64)	CGRT	RRF-c		GE				Dc						B960	7-149
COMPARE BIASED EXPONENT (extended DFP)	CEXTR	RRE	C	TF	α[7,9]		SP	Dt						B3FC	20-23
COMPARE BIASED EXPONENT (long DFP)	CEDTR	RRE	C	TF	α[7,9]			Dt						B3F4	20-23
COMPARE DECIMAL	CP	SS-b	C		α[9]	A		Dg				B_1	B_2	F9	8-7
COMPARE DOUBLE AND SWAP (32)	CDS	RS-a	C		α[9]	A	SP		$		ST		B_2	BB	7-144
COMPARE DOUBLE AND SWAP (32)	CDSY	RSY-a	C	LD	α[9]	A	SP		$		ST		B_2	EB31	7-144
COMPARE DOUBLE AND SWAP (64)	CDSG	RSY-a	C	N	α[9]	A	SP		$		ST		B_2	EB3E	7-144
COMPARE HALFWORD (32←16)	CH	RX-a	C			A							B_2	49	7-150
COMPARE HALFWORD (32←16)	CHY	RXY-a	C	LD		A							B_2	E379	7-150
COMPARE HALFWORD (64←16)	CGH	RXY-a	C	GE		A							B_2	E334	7-150
COMPARE HALFWORD IMMEDIATE (16←16)	CHHSI	SIL	C	GE		A						B_1		E554	7-150
COMPARE HALFWORD IMMEDIATE (32←16)	CHI	RI-a	C											A7E	7-150
COMPARE HALFWORD IMMEDIATE (32←16)	CHSI	SIL	C	GE		A						B_1		E55C	7-150
COMPARE HALFWORD IMMEDIATE (64←16)	CGHI	RI-a	C	N										A7F	7-150
COMPARE HALFWORD IMMEDIATE (64←16)	CGHSI	SIL	C	GE		A						B_1		E558	7-150
COMPARE HALFWORD RELATIVE LONG (32←16)	CHRL	RIL-b	C	GE		A*								C65	7-150
COMPARE HALFWORD RELATIVE LONG (64←16)	CGHRL	RIL-b	C	GE		A*								C64	7-150
COMPARE HIGH (32)	CHF	RXY-a	C	HW		A							B_2	E3CD	7-151
COMPARE HIGH (32)	CHHR	RRE	C	HW										B9CD	7-151
COMPARE HIGH (32)	CHLR	RRE	C	HW										B9DD	7-151
COMPARE IMMEDIATE (32)	CFI	RIL-a	C	EI										C2D	7-134
COMPARE IMMEDIATE (64←32)	CGFI	RIL-a	C	EI										C2C	7-135
COMPARE IMMEDIATE AND BRANCH (32←8)	CIB	RIS		GE	α[9]						B			ECFE	7-136
COMPARE IMMEDIATE AND BRANCH (64←8)	CGIB	RIS		GE	α[9]						B			ECFC	7-136
COMPARE IMMEDIATE AND BRANCH RELATIVE (32←8)	CIJ	RIE-c		GE	α[10]						B			EC7E	7-136
COMPARE IMMEDIATE AND BRANCH RELATIVE (64←8)	CGIJ	RIE-c		GE	α[10]						B			EC7C	7-136
COMPARE IMMEDIATE AND TRAP (32←16)	CIT	RIE-a		GE				Dc						EC72	7-149
COMPARE IMMEDIATE AND TRAP (64←16)	CGIT	RIE-a		GE				Dc						EC70	7-149
COMPARE IMMEDIATE HIGH (32)	CIH	RIL-a	C	HW										CCD	7-151
COMPARE LOGICAL (32)	CL	RX-a	C			A							B_2	55	7-152
COMPARE LOGICAL (32)	CLR	RR	C											15	7-152
COMPARE LOGICAL (32)	CLY	RXY-a	C	LD		A							B_2	E355	7-152
COMPARE LOGICAL (64)	CLG	RXY-a	C	N		A							B_2	E321	7-152
COMPARE LOGICAL (64)	CLGR	RRE	C	N										B921	7-152
COMPARE LOGICAL (64←32)	CLGF	RXY-a	C	N		A							B_2	E331	7-152
COMPARE LOGICAL (64←32)	CLGFR	RRE	C	N										B931	7-152
COMPARE LOGICAL (character)	CLC	SS-a	C		α[9]	A						B_1	B_2	D5	7-152
COMPARE LOGICAL (immediate)	CLI	SI	C			A						B_1		95	7-152
COMPARE LOGICAL (immediate)	CLIY	SIY	C	LD		A						B_1		EB55	7-152
COMPARE LOGICAL AND BRANCH (32)	CLRB	RRS		GE	α[9]						B			ECF7	7-154
COMPARE LOGICAL AND BRANCH (64)	CLGRB	RRS		GE	α[9]						B			ECE5	7-154
COMPARE LOGICAL AND BRANCH RELATIVE (32)	CLRJ	RIE-b		GE	α[10]						B			EC77	7-154
COMPARE LOGICAL AND BRANCH RELATIVE (64)	CLGRJ	RIE-b		GE	α[10]						B			EC65	7-154
COMPARE LOGICAL AND TRAP (32)	CLRT	RRF-c		GE				Dc						B973	7-155
COMPARE LOGICAL AND TRAP (32)	CLT	RSY-b		MI1		A		Dc					B_2	EB23	7-155
COMPARE LOGICAL AND TRAP (64)	CLGRT	RRF-c		GE				Dc						B961	7-155
COMPARE LOGICAL AND TRAP (64)	CLGT	RSY-b		MI1		A		Dc					B_2	EB2B	7-155

Figure B-1. Instructions Arranged by Name (Part 4 of 23)

Name	Mne-monic	Characteristics												Op-code	Page
COMPARE LOGICAL CHAR. UNDER MASK (high)	CLMH	RSY-b	C	N		A						B_2		EB20	7-157
COMPARE LOGICAL CHAR. UNDER MASK (low)	CLM	RS-b	C			A						B_2		BD	7-157
COMPARE LOGICAL CHAR. UNDER MASK (low)	CLMY	RSY-b	C	LD		A						B_2		EB21	7-157
COMPARE LOGICAL HIGH (32)	CLHF	RXY-a	C	HW		A						B_2		E3CF	7-157
COMPARE LOGICAL HIGH (32)	CLHHR	RRE	C	HW										B9CF	7-157
COMPARE LOGICAL HIGH (32)	CLHLR	RRE	C	HW										B9DF	7-157
COMPARE LOGICAL IMMEDIATE (16←16)	CLHHSI	SIL	C	GE		A						B_1		E555	7-152
COMPARE LOGICAL IMMEDIATE (32)	CLFI	RIL-a	C	EI										C2F	7-152
COMPARE LOGICAL IMMEDIATE (32←16)	CLFHSI	SIL	C	GE		A						B_1		E55D	7-152
COMPARE LOGICAL IMMEDIATE (64←16)	CLGHSI	SIL	C	GE		A						B_1		E559	7-152
COMPARE LOGICAL IMMEDIATE (64←32)	CLGFI	RIL-a	C	EI										C2E	7-152
COMPARE LOGICAL IMMEDIATE AND BRANCH (32←8)	CLIB	RIS		GE	α[9]						B			ECFF	7-154
COMPARE LOGICAL IMMEDIATE AND BRANCH (64←8)	CLGIB	RIS		GE	α[9]						B			ECFD	7-154
COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (32←8)	CLIJ	RIE-c		GE	α[10]						B			EC7F	7-154
COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (64←8)	CLGIJ	RIE-c		GE	α[10]						B			EC7D	7-154
COMPARE LOGICAL IMMEDIATE AND TRAP (32←16)	CLFIT	RIE-a		GE				Dc						EC73	7-156
COMPARE LOGICAL IMMEDIATE AND TRAP (64←16)	CLGIT	RIE-a		GE				Dc						EC71	7-156
COMPARE LOGICAL IMMEDIATE HIGH (32)	CLIH	RIL-a	C	HW										CCF	7-158
COMPARE LOGICAL LONG	CLCL	RR	C		α[9]	A	SP	Il				R_1	R_2	0F	7-158
COMPARE LOGICAL LONG EXTENDED	CLCLE	RS-a	C		α[9]	A	SP	IC				R_1	R_3	A9	7-160
COMPARE LOGICAL LONG UNICODE	CLCLU	RSY-a	C	E2	α[9]	A	SP	IC				R_1	R_2	EB8F	7-163
COMPARE LOGICAL RELATIVE LONG (32)	CLRL	RIL-b	C	GE		A*	SP							C6F	7-153
COMPARE LOGICAL RELATIVE LONG (32←16)	CLHRL	RIL-b	C	GE		A*								C67	7-153
COMPARE LOGICAL RELATIVE LONG (64)	CLGRL	RIL-b	C	GE		A*	SP							C6A	7-153
COMPARE LOGICAL RELATIVE LONG (64←16)	CLGHRL	RIL-b	C	GE		A*								C66	7-153
COMPARE LOGICAL RELATIVE LONG (64←32)	CLGFRL	RIL-b	C	GE		A*	SP							C6E	7-153
COMPARE LOGICAL STRING	CLST	RRE	C		α[9]	A	SP	IC		G0		R_1	R_2	B25D	7-166
COMPARE RELATIVE LONG (32)	CRL	RIL-b	C	GE		A*	SP							C6D	7-135
COMPARE RELATIVE LONG (64)	CGRL	RIL-b	C	GE		A*	SP							C68	7-135
COMPARE RELATIVE LONG (64←32)	CGFRL	RIL-b	C	GE		A*	SP							C6C	7-135
COMPARE UNTIL SUBSTRING EQUAL	CUSE	RRE	C		α[9]	A	SP	Il		GM		R_1	R_2	B257	7-167
COMPRESSION CALL	CMPSC	RRE	C		α[5,9]	A	SP	Il	Dg	GM	ST	R_1	R_2	B263	7-170
COMPUTE INTERMEDIATE MESSAGE DIGEST	KIMD	RRE	C	MS	α[5,9]	A	SP	IC		GM I1	ST		R_2	B93E	7-188
COMPUTE LAST MESSAGE DIGEST	KLMD	RRE	C	MS	α[5,9]	A	SP	IC		GM I1	ST		R_2	B93F	7-201
COMPUTE MESSAGE AUTHENTICATION CODE	KMAC	RRE	C	MS	α[5,9]	A	SP	IC		GM I1	ST		R_2	B91E	7-219
CONVERT BFP TO HFP (long)	THDR	RRE	C		α[7,9]			Da						B359	9-27
CONVERT BFP TO HFP (short to long)	THDER	RRE	C		α[7,9]			Da						B358	9-27
CONVERT FROM FIXED (32 to extended BFP)	CXFBR	RRE			α[7,9]		SP	Db						B396	19-19
CONVERT FROM FIXED (32 to extended BFP)	CXFBRA	RRF-e		F	α[7,9]		SP	Db						B396	19-19
CONVERT FROM FIXED (32 to extended DFP)	CXFTR	RRE		F	α[7,9]		SP	Dt						B959	20-24
CONVERT FROM FIXED (32 to extended HFP)	CXFR	RRE			α[7,9]		SP	Da						B3B6	18-11
CONVERT FROM FIXED (32 to long BFP)	CDFBR	RRE			α[7,9]			Db						B395	19-19
CONVERT FROM FIXED (32 to long BFP)	CDFBRA	RRF-e		F	α[7,9]		SP	Db						B395	19-19
CONVERT FROM FIXED (32 to long DFP)	CDFTR	RRE		F	α[7,9]			Dt						B951	20-24
CONVERT FROM FIXED (32 to long HFP)	CDFR	RRE			α[7,9]			Da						B3B5	18-11
CONVERT FROM FIXED (32 to short BFP)	CEFBR	RRE			α[7,9]			Db		Xx				B394	19-19
CONVERT FROM FIXED (32 to short BFP)	CEFBRA	RRF-e		F	α[7,9]		SP	Db		Xx				B394	19-19
CONVERT FROM FIXED (32 to short HFP)	CEFR	RRE			α[7,9]			Da						B3B4	18-11
CONVERT FROM FIXED (64 to extended BFP)	CXGBR	RRE		N	α[7,9]		SP	Db						B3A6	19-19
CONVERT FROM FIXED (64 to extended BFP)	CXGBRA	RRF-e		F	α[7,9]		SP	Db						B3A6	19-19

Figure B-1. Instructions Arranged by Name (Part 5 of 23)

Name	Mnemonic	Characteristics															Opcode	Page
CONVERT FROM FIXED (64 to extended DFP)	CXGTR	RRE	TF	¤[7,9]		SP	Dt										B3F9	20-24
CONVERT FROM FIXED (64 to extended DFP)	CXGTRA	RRF-e	F	¤[7,9]		SP	Dt										B3F9	20-24
CONVERT FROM FIXED (64 to extended HFP)	CXGR	RRE	N	¤[7,9]		SP	Da										B3C6	18-11
CONVERT FROM FIXED (64 to long BFP)	CDGBR	RRE	N	¤[7,9]			Db			Xx							B3A5	19-19
CONVERT FROM FIXED (64 to long BFP)	CDGBRA	RRF-e	F	¤[7,9]		SP	Db			Xx							B3A5	19-19
CONVERT FROM FIXED (64 to long DFP)	CDGTR	RRE	TF	¤[7,9]			Dt			Xx							B3F1	20-24
CONVERT FROM FIXED (64 to long DFP)	CDGTRA	RRF-e	F	¤[7,9]			Dt			Xx	Xq						B3F1	20-24
CONVERT FROM FIXED (64 to long HFP)	CDGR	RRE	N	¤[7,9]			Da										B3C5	18-11
CONVERT FROM FIXED (64 to short BFP)	CEGBR	RRE	N	¤[7,9]			Db			Xx							B3A4	19-19
CONVERT FROM FIXED (64 to short BFP)	CEGBRA	RRF-e	F	¤[7,9]		SP	Db			Xx							B3A4	19-19
CONVERT FROM FIXED (64 to short HFP)	CEGR	RRE	N	¤[7,9]			Da										B3C4	18-11
CONVERT FROM LOGICAL (32 to extended BFP)	CXLFBR	RRF-e	F	¤[7,9]		SP	Db										B392	19-21
CONVERT FROM LOGICAL (32 to extended DFP)	CXLFTR	RRF-e	F	¤[7,9]		SP	Dt										B95B	20-25
CONVERT FROM LOGICAL (32 to long BFP)	CDLFBR	RRF-e	F	¤[7,9]		SP	Db										B391	19-21
CONVERT FROM LOGICAL (32 to long DFP)	CDLFTR	RRF-e	F	¤[7,9]			Dt										B953	20-25
CONVERT FROM LOGICAL (32 to short BFP)	CELFBR	RRF-e	F	¤[7,9]		SP	Db			Xx							B390	19-21
CONVERT FROM LOGICAL (64 to extended BFP)	CXLGBR	RRF-e	F	¤[7,9]		SP	Db										B3A2	19-21
CONVERT FROM LOGICAL (64 to extended DFP)	CXLGTR	RRF-e	F	¤[7,9]		SP	Dt										B95A	20-25
CONVERT FROM LOGICAL (64 to long BFP)	CDLGBR	RRF-e	F	¤[7,9]		SP	Db			Xx							B3A1	19-21
CONVERT FROM LOGICAL (64 to long DFP)	CDLGTR	RRF-e	F	¤[7,9]			Dt			Xx	Xq						B952	20-25
CONVERT FROM LOGICAL (64 to short BFP)	CELGBR	RRF-e	F	¤[7,9]		SP	Db			Xx							B3A0	19-21
CONVERT FROM PACKED (to extended DFP)	CXPT	RSL-b	PC	¤[7,9]	A	SP	Dt	Dg								B_2	EDAF	20-26
CONVERT FROM PACKED (to long DFP)	CDPT	RSL-b	PC	¤[7,9]	A	SP	Dt	Dg								B_2	EDAE	20-26
CONVERT FROM SIGNED PACKED (128 to extended DFP)	CXSTR	RRE	TF	¤[7,9]		SP	Dt	Dg									B3FB	20-28
CONVERT FROM SIGNED PACKED (64 to long DFP)	CDSTR	RRE	TF	¤[7,9]			Dt	Dg									B3F3	20-28
CONVERT FROM UNSIGNED PACKED (128 to ext. DFP)	CXUTR	RRE	TF	¤[7,9]		SP	Dt	Dg									B3FA	20-28
CONVERT FROM UNSIGNED PACKED (64 to long DFP)	CDUTR	RRE	TF	¤[7,9]			Dt	Dg									B3F2	20-28
CONVERT FROM ZONED (to extended DFP)	CXZT	RSL-b	ZF	¤[7,9]	A	SP	Dt	Dg								B_2	EDAB	20-29
CONVERT FROM ZONED (to long DFP)	CDZT	RSL-b	ZF	¤[7,9]	A	SP	Dt	Dg								B_2	EDAA	20-29
CONVERT HFP TO BFP (long to short)	TBEDR	RRF-e	C	¤[7,9]		SP	Da										B350	9-28
CONVERT HFP TO BFP (long)	TBDR	RRF-e	C	¤[7,9]		SP	Da										B351	9-28
CONVERT TO BINARY (32)	CVB	RX-a		¤[9]	A			Dg	IK							B_2	4F	7-230
CONVERT TO BINARY (32)	CVBY	RXY-a	LD	¤[9]	A			Dg	IK							B_2	E306	7-230
CONVERT TO BINARY (64)	CVBG	RXY-a	N	¤[9]	A			Dg	IK							B_2	E30E	7-230
CONVERT TO DECIMAL (32)	CVD	RX-a		¤[9]	A									ST		B_2	4E	7-231
CONVERT TO DECIMAL (32)	CVDY	RXY-a	LD	¤[9]	A									ST		B_2	E326	7-231
CONVERT TO DECIMAL (64)	CVDG	RXY-a	N	¤[9]	A									ST		B_2	E32E	7-231
CONVERT TO FIXED (extended BFP to 32)	CFXBR	RRF-e	C	¤[7,9]		SP	Db	Xi		Xx							B39A	19-22
CONVERT TO FIXED (extended BFP to 32)	CFXBRA	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx						B39A	19-22
CONVERT TO FIXED (extended BFP to 64)	CGXBR	RRF-e	C	N	¤[7,9]		SP	Db	Xi		Xx						B3AA	19-22
CONVERT TO FIXED (extended BFP to 64)	CGXBRA	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx						B3AA	19-22
CONVERT TO FIXED (extended DFP to 32)	CFXTR	RRF-e	C	F	¤[7,9]		SP	Dt	Xi		Xx						B949	20-30
CONVERT TO FIXED (extended DFP to 64)	CGXTR	RRF-e	C	TF	¤[7,9]		SP	Dt	Xi		Xx						B3E9	20-29
CONVERT TO FIXED (extended DFP to 64)	CGXTRA	RRF-e	C	F	¤[7,9]		SP	Dt	Xi		Xx						B3E9	20-30
CONVERT TO FIXED (extended HFP to 32)	CFXR	RRF-e	C		¤[7,9]		SP	Da									B3BA	18-11
CONVERT TO FIXED (extended HFP to 64)	CGXR	RRF-e	C	N	¤[7,9]		SP	Da									B3CA	18-11
CONVERT TO FIXED (long BFP to 32)	CFDBR	RRF-e	C		¤[7,9]		SP	Db	Xi		Xx						B399	19-22
CONVERT TO FIXED (long BFP to 32)	CFDBRA	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx						B399	19-22
CONVERT TO FIXED (long BFP to 64)	CGDBR	RRF-e	C	N	¤[7,9]		SP	Db	Xi		Xx						B3A9	19-22
CONVERT TO FIXED (long BFP to 64)	CGDBRA	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx						B3A9	19-22
CONVERT TO FIXED (long DFP to 32)	CFDTR	RRF-e	C	F	¤[7,9]			Dt	Xi		Xx						B941	20-30
CONVERT TO FIXED (long DFP to 64)	CGDTR	RRF-e	C	TF	¤[7,9]			Dt	Xi		Xx						B3E1	20-29

Figure B-1. Instructions Arranged by Name (Part 6 of 23)

Name	Mnemonic	Characteristics																						Opcode	Page
CONVERT TO FIXED (long DFP to 64)	CGDTRA	RRF-e	C		F	α[7,9]				Dt	Xi				Xx									B3E1	20-30
CONVERT TO FIXED (long HFP to 32)	CFDR	RRF-e	C			α[7,9]		SP		Da														B3B9	18-11
CONVERT TO FIXED (long HFP to 64)	CGDR	RRF-e	C		N	α[7,9]		SP		Da														B3C9	18-11
CONVERT TO FIXED (short BFP to 32)	CFEBR	RRF-e	C			α[7,9]		SP		Db	Xi				Xx									B398	19-22
CONVERT TO FIXED (short BFP to 32)	CFEBRA	RRF-e	C		F	α[7,9]		SP		Db	Xi				Xx									B398	19-22
CONVERT TO FIXED (short BFP to 64)	CGEBR	RRF-e	C		N	α[7,9]		SP		Db	Xi				Xx									B3A8	19-22
CONVERT TO FIXED (short BFP to 64)	CGEBRA	RRF-e	C		F	α[7,9]		SP		Db	Xi				Xx									B3A8	19-22
CONVERT TO FIXED (short HFP to 32)	CFER	RRF-e	C			α[7,9]		SP		Da														B3B8	18-11
CONVERT TO FIXED (short HFP to 64)	CGER	RRF-e	C		N	α[7,9]		SP		Da														B3C8	18-11
CONVERT TO LOGICAL (extended BFP to 32)	CLFXBR	RRF-e	C		F	α[7,9]		SP		Db	Xi				Xx									B39E	19-25
CONVERT TO LOGICAL (extended BFP to 64)	CLGXBR	RRF-e	C		F	α[7,9]		SP		Db	Xi				Xx									B3AE	19-25
CONVERT TO LOGICAL (extended DFP to 32)	CLFXTR	RRF-e	C		F	α[7,9]		SP		Dt	Xi				Xx									B94B	20-32
CONVERT TO LOGICAL (extended DFP to 64)	CLGXTR	RRF-e	C		F	α[7,9]		SP		Dt	Xi				Xx									B94A	20-32
CONVERT TO LOGICAL (long BFP to 32)	CLFDBR	RRF-e	C		F	α[7,9]		SP		Db	Xi				Xx									B39D	19-25
CONVERT TO LOGICAL (long BFP to 64)	CLGDBR	RRF-e	C		F	α[7,9]		SP		Db	Xi				Xx									B3AD	19-25
CONVERT TO LOGICAL (long DFP to 32)	CLFDTR	RRF-e	C		F	α[7,9]				Dt	Xi				Xx									B943	20-32
CONVERT TO LOGICAL (long DFP to 64)	CLGDTR	RRF-e	C		F	α[7,9]				Dt	Xi				Xx									B942	20-32
CONVERT TO LOGICAL (short BFP to 32)	CLFEBR	RRF-e	C		F	α[7,9]		SP		Db	Xi				Xx									B39C	19-25
CONVERT TO LOGICAL (short BFP to 64)	CLGEBR	RRF-e	C		F	α[7,9]		SP		Db	Xi				Xx									B3AC	19-25
CONVERT TO PACKED (from extended DFP)	CPXT	RSL-b	C		PC	α[7,9]	A	SP		Dt	DF						ST		B_2					EDAD	20-33
CONVERT TO PACKED (from long DFP)	CPDT	RSL-b	C		PC	α[7,9]	A	SP		Dt	DF						ST		B_2					EDAC	20-33
CONVERT TO SIGNED PACKED (extended DFP to 128)	CSXTR	RRF-d			TF	α[7,9]		SP		Dt														B3EB	20-35
CONVERT TO SIGNED PACKED (long DFP to 64)	CSDTR	RRF-d			TF	α[7,9]				Dt														B3E3	20-35
CONVERT TO UNSIGNED PACKED (extended DFP to 128)	CUXTR	RRE			TF	α[7,9]		SP		Dt														B3EA	20-35
CONVERT TO UNSIGNED PACKED (long DFP to 64)	CUDTR	RRE			TF	α[7,9]				Dt														B3E2	20-35
CONVERT TO ZONED (from extended DFP)	CZXT	RSL-b	C		ZF	α[7,9]	A	SP									ST		B_2					EDA9	20-36
CONVERT TO ZONED (from long DFP)	CZDT	RSL-b	C		ZF	α[7,9]	A	SP									ST		B_2					EDA8	20-36
CONVERT UNICODE TO UTF-8	CUUTF	RRF-c	C			α[5,9]	A	SP		IC							ST	R_1	R_2					B2A6	7-234
CONVERT UTF-16 TO UTF-32	CU24	RRF-c	C		E3	α[5,9]	A	SP		IC							ST	R_1	R_2					B9B1	7-231
CONVERT UTF-16 TO UTF-8	CU21	RRF-c	C			α[5,9]	A	SP		IC							ST	R_1	R_2					B2A6	7-234
CONVERT UTF-8 TO UNICODE	CUTFU	RRF-c	C			α[5,9]	A	SP		IC							ST	R_1	R_2					B2A7	7-244
CONVERT UTF-8 TO UTF-16	CU12	RRF-c	C			α[5,9]	A	SP		IC							ST	R_1	R_2					B2A7	7-244
CONVERT UTF-8 TO UTF-32	CU14	RRF-c	C		E3	α[5,9]	A	SP		IC							ST	R_1	R_2					B9B0	7-248
CONVERT UTF-32 TO UTF-16	CU42	RRE	C		E3	α[5,9]	A	SP		IC							ST	R_1	R_2					B9B3	7-238
CONVERT UTF-32 TO UTF-8	CU41	RRE	C		E3	α[5,9]	A	SP		IC							ST	R_1	R_2					B9B2	7-241
COPY ACCESS	CPYA	RRE				α[6]												U_1	U_2					B24D	7-252
COPY SIGN (long)	CPSDR	RRF-b			FS	α[7,9]				Da														B372	9-30
DIAGNOSE	—		DM			P	DM																MD	83	10-23
DIVIDE (32←64)	D	RX-a				α[9]	A	SP				IK							B_2					5D	7-252
DIVIDE (32←64)	DR	RR				α[9]		SP				IK												1D	7-252
DIVIDE (extended BFP)	DXBR	RRE				α[7,9]		SP		Db	Xi	Xz	Xo	Xu	Xx									B34D	19-27
DIVIDE (extended DFP)	DXTR	RRF-a			TF	α[7,9]		SP		Dt	Xi	Xz	Xo	Xu	Xx									B3D9	20-37
DIVIDE (extended DFP)	DXTRA	RRF-a			F	α[7,9]		SP		Dt	Xi	Xz	Xo	Xu	Xx	Xq								B3D9	20-37
DIVIDE (extended HFP)	DXR	RRE				α[7,9]		SP		Da	EU	EO	FK											B22D	18-12
DIVIDE (long BFP)	DDB	RXE				α[7,9]	A			Db	Xi	Xz	Xo	Xu	Xx				B_2					ED1D	19-27
DIVIDE (long BFP)	DDBR	RRE				α[7,9]				Db	Xi	Xz	Xo	Xu	Xx									B31D	19-27
DIVIDE (long DFP)	DDTR	RRF-a			TF	α[7,9]				Dt	Xi	Xz	Xo	Xu	Xx									B3D1	20-37
DIVIDE (long DFP)	DDTRA	RRF-a			F	α[7,9]				Dt	Xi	Xz	Xo	Xu	Xx	Xq								B3D1	20-37
DIVIDE (long HFP)	DD	RX-a				α[7,9]	A			Da	EU	EO	FK						B_2					6D	18-12
DIVIDE (long HFP)	DDR	RR				α[7,9]				Da	EU	EO	FK											2D	18-12
DIVIDE (short BFP)	DEB	RXE				α[7,9]	A			Db	Xi	Xz	Xo	Xu	Xx				B_2					ED0D	19-27
DIVIDE (short BFP)	DEBR	RRE				α[7,9]				Db	Xi	Xz	Xo	Xu	Xx									B30D	19-27

Figure B-1. Instructions Arranged by Name (Part 7 of 23)

Name	Mnemonic	Characteristics																						Opcode	Page
DIVIDE (short HFP)	DE	RX-a			¤[7,9]	A		Da	EU	EO	FK							ST	B_1	B_2				7D	18-12
DIVIDE (short HFP)	DER	RR			¤[7,9]			Da	EU	EO	FK													3D	18-12
DIVIDE DECIMAL	DP	SS-b			¤[9]	A	SP	Dg			DK							ST	B_1	B_2				FD	8-7
DIVIDE LOGICAL (32←64)	DL	RXY-a		N3	¤[9]	A	SP				IK									B_2				E397	7-253
DIVIDE LOGICAL (32←64)	DLR	RRE		N3	¤[9]		SP				IK													B997	7-253
DIVIDE LOGICAL (64←128)	DLG	RXY-a		N	¤[9]	A	SP				IK									B_2				E387	7-253
DIVIDE LOGICAL (64←128)	DLGR	RRE		N	¤[9]		SP				IK													B987	7-253
DIVIDE SINGLE (64)	DSG	RXY-a		N	¤[9]	A	SP				IK									B_2				E30D	7-254
DIVIDE SINGLE (64)	DSGR	RRE		N	¤[9]		SP				IK													B90D	7-254
DIVIDE SINGLE (64←32)	DSGF	RXY-a		N	¤[9]	A	SP				IK									B_2				E31D	7-254
DIVIDE SINGLE (64←32)	DSGFR	RRE		N	¤[9]		SP				IK													B91D	7-254
DIVIDE TO INTEGER (long BFP)	DIDBR	RRF-b	C		¤[7,9]		SP	Db	Xi					Xu	Xx									B35B	19-28
DIVIDE TO INTEGER (short BFP)	DIEBR	RRF-b	C		¤[7,9]		SP	Db	Xi					Xu	Xx									B353	19-28
EDIT	ED	SS-a	C		¤[9]	A		Dg										ST	B_1	B_2				DE	8-8
EDIT AND MARK	EDMK	SS-a	C		¤[9]	A		Dg				G1						ST	B_1	B_2				DF	8-11
EXCLUSIVE OR (32)	X	RX-a	C			A														B_2				57	7-254
EXCLUSIVE OR (32)	XR	RR	C																					17	7-254
EXCLUSIVE OR (32)	XRK	RRF-a	C	DO																				B9F7	7-254
EXCLUSIVE OR (32)	XY	RXY-a	C	LD		A														B_2				E357	7-254
EXCLUSIVE OR (64)	XG	RXY-a	C	N		A														B_2				E382	7-254
EXCLUSIVE OR (64)	XGR	RRE	C	N																				B982	7-254
EXCLUSIVE OR (64)	XGRK	RRF-a	C	DO																				B9E7	7-254
EXCLUSIVE OR (character)	XC	SS-a	C		¤[9]	A												ST	B_1	B_2				D7	7-255
EXCLUSIVE OR (immediate)	XI	SI	C			A												ST	B_1					97	7-255
EXCLUSIVE OR (immediate)	XIY	SIY	C	LD		A												ST	B_1					EB57	7-255
EXCLUSIVE OR IMMEDIATE (high)	XIHF	RIL-a	C	EI																				C06	7-256
EXCLUSIVE OR IMMEDIATE (low)	XILF	RIL-a	C	EI																				C07	7-256
EXECUTE	EX	RX-a			¤[9]	AI	SP					EX												44	7-256
EXECUTE RELATIVE LONG	EXRL	RIL-b		XX	¤[9]	AI*						EX												C60	7-256
EXTRACT ACCESS	EAR	RRE																			U_2			B24F	7-257
EXTRACT AND SET EXTENDED AUTHORITY	ESEA	RRE		N	P																			B99D	10-24
EXTRACT BIASED EXPONENT (extended DFP to 64)	EEXTR	RRE		TF	¤[7,9]		SP	Dt																B3ED	20-39
EXTRACT BIASED EXPONENT (long DFP to 64)	EEDTR	RRE		TF	¤[7,9]			Dt																B3E5	20-39
EXTRACT CPU ATTRIBUTE	ECAG	RSY-a		GE	¤[9]																			EB4C	7-257
EXTRACT CPU TIME	ECTG	SSF		ET	¤[8,9]	A						GM						R_3	B_1	B_2				C81	7-260
EXTRACT FPC	EFPC	RRE			¤[7,9]			Db																B38C	9-30
EXTRACT PRIMARY ASN	EPAR	RRE			Q			SO																B226	10-24
EXTRACT PRIMARY ASN AND INSTANCE	EPAIR	RRE		RA	Q			SO																B99A	10-24
EXTRACT PSW	EPSW	RRE		N3	¤[8,9]																			B98D	7-261
EXTRACT SECONDARY ASN	ESAR	RRE			Q			SO																B227	10-24
EXTRACT SECONDARY ASN AND INSTANCE	ESAIR	RRE		RA	Q			SO																B99B	10-25
EXTRACT SIGNIFICANCE (extended DFP to 64)	ESXTR	RRE		TF	¤[7,9]		SP	Dt																B3EF	20-39
EXTRACT SIGNIFICANCE (long DFP to 64)	ESDTR	RRE		TF	¤[7,9]			Dt																B3E7	20-39
EXTRACT STACKED REGISTERS (32)	EREG	RRE			¤[1]	A[1]*		SE											U_1	U_2				B249	10-25
EXTRACT STACKED REGISTERS (64)	EREGG	RRE		N	¤[1]	A[1]*		SE											U_1	U_2				B90E	10-25
EXTRACT STACKED STATE	ESTA	RRE	C		¤[1]	A[1]*	SP	SE																B24A	10-26
EXTRACT TRANSACTION NESTING DEPTH	ETND	RRE		TX	¤[9]			SO																B2EC	7-261
FIND LEFTMOST ONE	FLOGR	RRE	C	EI			SP																	B983	7-262
HALT SUBCHANNEL	HSCH	S	C		P			OP		¢	GS													B231	14-6
HALVE (long HFP)	HDR	RR			¤[7,9]			Da	EU															24	18-13
HALVE (short HFP)	HER	RR			¤[7,9]			Da	EU															34	18-13
INSERT ADDRESS SPACE CONTROL	IAC	RRE	C		Q			SO																B224	10-29
INSERT BIASED EXPONENT (64 to extended DFP)	IEXTR	RRF-b		TF	¤[7,9]		SP	Dt																B3FE	20-40
INSERT BIASED EXPONENT (64 to long DFP)	IEDTR	RRF-b		TF	¤[7,9]			Dt																B3F6	20-40

Figure B-1. Instructions Arranged by Name (Part 8 of 23)

Name	Mnemonic	Characteristics												Opcode	Page
INSERT CHARACTER	IC	RX-a				A							B_2	43	7-262
INSERT CHARACTER	ICY	RXY-a		LD		A							B_2	E373	7-262
INSERT CHARACTERS UNDER MASK (high)	ICMH	RSY-b	C	N		A							B_2	EB80	7-262
INSERT CHARACTERS UNDER MASK (low)	ICM	RS-b	C			A							B_2	BF	7-262
INSERT CHARACTERS UNDER MASK (low)	ICMY	RSY-b	C	LD		A							B_2	EB81	7-262
INSERT IMMEDIATE (high high)	IIHH	RI-a		N										A50	7-263
INSERT IMMEDIATE (high low)	IIHL	RI-a		N										A51	7-263
INSERT IMMEDIATE (high)	IIHF	RIL-a		EI										C08	7-263
INSERT IMMEDIATE (low high)	IILH	RI-a		N										A52	7-263
INSERT IMMEDIATE (low low)	IILL	RI-a		N										A53	7-263
INSERT IMMEDIATE (low)	IILF	RIL-a		EI										C09	7-263
INSERT PROGRAM MASK	IPM	RRE												B222	7-264
INSERT PSW KEY	IPK	S			Q				G2					B20B	10-30
INSERT REFERENCE BITS MULTIPLE	IRBM	RRE		IM	P	A[1]*								B9AC	10-30
INSERT STORAGE KEY EXTENDED	ISKE	RRE			P	A[1]*								B229	10-30
INSERT VIRTUAL STORAGE KEY	IVSK	RRE			Q	A[1]*		SO					R_2	B223	10-31
INVALIDATE DAT TABLE ENTRY	IDTE	RRF-b	U	DE	P	A[1]	SP		$					B98E	10-32
INVALIDATE PAGE TABLE ENTRY	IPTE	RRF-a			P	A[1]	SP		$					B221	10-37
LOAD (32)	L	RX-a				A							B_2	58	7-264
LOAD (32)	LR	RR												18	7-264
LOAD (32)	LY	RXY-a		LD		A							B_2	E358	7-264
LOAD (64)	LG	RXY-a		N		A							B_2	E304	7-264
LOAD (64)	LGR	RRE		N										B904	7-264
LOAD (64←32)	LGF	RXY-a		N		A							B_2	E314	7-264
LOAD (64←32)	LGFR	RRE		N										B914	7-264
LOAD (extended)	LXR	RRE			¤[7,9]		SP	Da						B365	9-31
LOAD (long)	LD	RX-a			¤[7,9]	A		Da					B_2	68	9-31
LOAD (long)	LDR	RR			¤[7,9]			Da						28	9-31
LOAD (long)	LDY	RXY-a		LD	¤[7,9]	A		Da					B_2	ED65	9-31
LOAD (short)	LE	RX-a			¤[7,9]	A		Da					B_2	78	9-31
LOAD (short)	LER	RR			¤[7,9]			Da						38	9-31
LOAD (short)	LEY	RXY-a		LD	¤[7,9]	A		Da					B_2	ED64	9-31
LOAD ACCESS MULTIPLE	LAM	RS-a			¤[6]	A	SP						UB	9A	7-265
LOAD ACCESS MULTIPLE	LAMY	RSY-a		LD	¤[6]	A	SP						UB	EB9A	7-265
LOAD ADDRESS	LA	RX-a												41	7-266
LOAD ADDRESS	LAY	RXY-a		LD										E371	7-266
LOAD ADDRESS EXTENDED	LAE	RX-a			¤[6]								U_1 BP	51	7-266
LOAD ADDRESS EXTENDED	LAEY	RXY-a		GE	¤[6]								U_1 BP	E375	7-266
LOAD ADDRESS RELATIVE LONG	LARL	RIL-b		N3										C00	7-267
LOAD ADDRESS SPACE PARAMETERS	LASP	SSE	C		P	A[1]	SP	SO					B_1	E500	10-41
LOAD AND ADD (32)	LAA	RSY-a	C	IA	¤[9]	A	SP		IF	£		ST	B_2	EBF8	7-268
LOAD AND ADD (64)	LAAG	RSY-a	C	IA	¤[9]	A	SP		IF	£		ST	B_2	EBE8	7-268
LOAD AND ADD LOGICAL (32)	LAAL	RSY-a	C	IA	¤[9]	A	SP			£		ST	B_2	EBFA	7-268
LOAD AND ADD LOGICAL (64)	LAALG	RSY-a	C	IA	¤[9]	A	SP			£		ST	B_2	EBEA	7-268
LOAD AND AND (32)	LAN	RSY-a	C	IA	¤[9]	A	SP			£		ST	B_2	EBF4	7-269
LOAD AND AND (64)	LANG	RSY-a	C	IA	¤[9]	A	SP			£		ST	B_2	EBE4	7-269
LOAD AND EXCLUSIVE OR (32)	LAX	RSY-a	C	IA	¤[9]	A	SP			£		ST	B_2	EBF7	7-269
LOAD AND EXCLUSIVE OR (64)	LAXG	RSY-a	C	IA	¤[9]	A	SP			£		ST	B_2	EBE7	7-269
LOAD AND OR (32)	LAO	RSY-a	C	IA	¤[9]	A	SP			£		ST	B_2	EBF6	7-270
LOAD AND OR (64)	LAOG	RSY-a	C	IA	¤[9]	A	SP			£		ST	B_2	EBE6	7-270
LOAD AND TEST (32)	LT	RXY-a	C	EI		A							B_2	E312	7-271
LOAD AND TEST (32)	LTR	RR	C											12	7-270
LOAD AND TEST (64)	LTG	RXY-a	C	EI		A							B_2	E302	7-271
LOAD AND TEST (64)	LTGR	RRE	C	N										B902	7-270
LOAD AND TEST (64←32)	LTGF	RXY-a	C	GE		A							B_2	E332	7-271

Figure B-1. Instructions Arranged by Name (Part 9 of 23)

Name	Mne-monic	Characteristics													Op-code	Page
LOAD AND TEST (64←32)	LTGFR	RRE	C	N											B912	7-270
LOAD AND TEST (extended BFP)	LTXBR	RRE	C		[7,9]		SP	Db	Xi						B342	19-31
LOAD AND TEST (extended DFP)	LTXTR	RRE	C	TF	[7,9]		SP	Dt	Xi						B3DE	20-41
LOAD AND TEST (extended HFP)	LTXR	RRE	C		[7,9]		SP	Da							B362	18-14
LOAD AND TEST (long BFP)	LTDBR	RRE	C		[7,9]			Db	Xi						B312	19-31
LOAD AND TEST (long DFP)	LTDTR	RRE	C	TF	[7,9]			Dt	Xi						B3D6	20-41
LOAD AND TEST (long HFP)	LTDR	RR	C		[7,9]			Da							22	18-13
LOAD AND TEST (short BFP)	LTEBR	RRE	C		[7,9]			Db	Xi						B302	19-31
LOAD AND TEST (short HFP)	LTER	RR	C		[7,9]			Da							32	18-13
LOAD AND TRAP (32L←32)	LAT	RXY-a		LT		A			Dc					B_2	E39F	7-271
LOAD AND TRAP (64)	LGAT	RXY-a		LT		A			Dc					B_2	E385	7-271
LOAD AND ZERO RIGHTMOST BYTE (32)	LZRF	RXY-a		LZ		A								B_2	E33B	7-271
LOAD AND ZERO RIGHTMOST BYTE (64)	LZRG	RXY-a		LZ		A								B_2	E32A	7-271
LOAD BYTE (32←8)	LB	RXY-a		LD		A									E376	7-272
LOAD BYTE (32←8)	LBR	RRE		EI											B926	7-272
LOAD BYTE (64←8)	LGB	RXY-a		LD		A									E377	7-272
LOAD BYTE (64←8)	LGBR	RRE		EI											B906	7-272
LOAD BYTE HIGH (32←8)	LBH	RXY-a		HW		A								B_2	E3C0	7-272
LOAD COMPLEMENT (32)	LCR	RR	C						IF						13	7-272
LOAD COMPLEMENT (64)	LCGR	RRE	C	N					IF						B903	7-273
LOAD COMPLEMENT (64←32)	LCGFR	RRE	C	N											B913	7-273
LOAD COMPLEMENT (extended BFP)	LCXBR	RRE	C		[7,9]		SP	Db							B343	19-31
LOAD COMPLEMENT (extended HFP)	LCXR	RRE	C		[7,9]		SP	Da							B363	18-14
LOAD COMPLEMENT (long BFP)	LCDBR	RRE	C		[7,9]			Db							B313	19-31
LOAD COMPLEMENT (long HFP)	LCDR	RR	C		[7,9]			Da							23	18-14
LOAD COMPLEMENT (long)	LCDFR	RRE		FS	[7,9]			Da							B373	9-31
LOAD COMPLEMENT (short BFP)	LCEBR	RRE	C		[7,9]			Db							B303	19-31
LOAD COMPLEMENT (short HFP)	LCER	RR	C		[7,9]			Da							33	18-14
LOAD CONTROL (32)	LCTL	RS-a			P	A	SP							B_2	B7	10-50
LOAD CONTROL (64)	LCTLG	RSY-a		N	P	A	SP							B_2	EB2F	10-50
LOAD COUNT TO BLOCK BOUNDARY	LCBB	RXE	C	VF			SP								E727	7-273
LOAD FP INTEGER (extended BFP)	FIXBR	RRF-e			[7,9]		SP	Db	Xi		Xx				B347	19-32
LOAD FP INTEGER (extended BFP)	FIXBRA	RRF-e		F	[7,9]		SP	Db	Xi		Xx				B347	19-32
LOAD FP INTEGER (extended DFP)	FIXTR	RRF-e		TF	[7,9]		SP	Dt	Xi		Xx	Xq			B3DF	20-42
LOAD FP INTEGER (extended HFP)	FIXR	RRE			[7,9]		SP	Da							B367	18-15
LOAD FP INTEGER (long BFP)	FIDBR	RRF-e			[7,9]		SP	Db	Xi		Xx				B35F	19-32
LOAD FP INTEGER (long BFP)	FIDBRA	RRF-e		F	[7,9]		SP	Db	Xi		Xx				B35F	19-32
LOAD FP INTEGER (long DFP)	FIDTR	RRF-e		TF	[7,9]			Dt	Xi		Xx	Xq			B3D7	20-42
LOAD FP INTEGER (long HFP)	FIDR	RRE			[7,9]			Da							B37F	18-15
LOAD FP INTEGER (short BFP)	FIEBR	RRF-e			[7,9]		SP	Db	Xi		Xx				B357	19-32
LOAD FP INTEGER (short BFP)	FIEBRA	RRF-e		F	[7,9]		SP	Db	Xi		Xx				B357	19-32
LOAD FP INTEGER (short HFP)	FIER	RRE			[7,9]			Da							B377	18-15
LOAD FPC	LFPC	S			[7,9]	A	SP	Db						B_2	B29D	9-31
LOAD FPC AND SIGNAL	LFAS	S		XF	[7,9]	A	SP	Dt	Xg					B_2	B2BD	9-32
LOAD FPR FROM GR (64 to long)	LDGR	RRE		FG	[7,9]			Da							B3C1	9-34
LOAD GR FROM FPR (long to 64)	LGDR	RRE		FG	[7,9]			Da							B3CD	9-34
LOAD GUARDED (64)	LGG	RXY-a		GF	[12]	A	SP						B ST	B_2	E34C	7-274
LOAD GUARDED STORAGE CONTROLS	LGSC	RXY-a		GF	[1]	A		SO						B_2	E34D	7-275
LOAD HALFWORD (32←16)	LH	RX-a				A								B_2	48	7-276
LOAD HALFWORD (32←16)	LHR	RRE		EI											B927	7-276
LOAD HALFWORD (32←16)	LHY	RXY-a		LD		A								B_2	E378	7-276
LOAD HALFWORD (64←16)	LGH	RXY-a		N		A								B_2	E315	7-276
LOAD HALFWORD (64←16)	LGHR	RRE		EI											B907	7-276
LOAD HALFWORD HIGH (32←16)	LHH	RXY-a		HW		A								B_2	E3C4	7-277

Figure B-1. Instructions Arranged by Name (Part 10 of 23)

Name	Mne-monic	Characteristics								Op-code	Page
LOAD HALFWORD HIGH IMMEDIATE ON CONDITION (32←16)	LOCHHI	RIE-g	L2							EC4E	7-277
LOAD HALFWORD IMMEDIATE (32←16	LHI	RI-a								A78	7-276
LOAD HALFWORD IMMEDIATE (64←16)	LGHI	RI-a	N							A79	7-276
LOAD HALFWORD IMMEDIATE ON CONDITION (32←16)	LOCHI	RIE-g	L2							EC42	7-277
LOAD HALFWORD IMMEDIATE ON CONDITION (64←16)	LOCGHI	RIE-g	L2							EC46	7-277
LOAD HALFWORD RELATIVE LONG (32←16)	LHRL	RIL-b	GE		A*					C45	7-276
LOAD HALFWORD RELATIVE LONG (64←16)	LGHRL	RIL-b	GE		A*					C44	7-276
LOAD HIGH (32)	LFH	RXY-a	HW		A				B_2	E3CA	7-278
LOAD HIGH AND TRAP (32H←32)	LFHAT	RXY-a	LT		A		Dc		B_2	E3C8	7-278
LOAD HIGH ON CONDITION (32)	LOCFH	RSY-b	L2		A				B_2	EBE0	7-284
LOAD HIGH ON CONDITION (32)	LOCFHR	RRF-c	L2							B9E0	7-284
LOAD IMMEDIATE (64←32)	LGFI	RIL-a	EI							C01	7-264
LOAD LENGTHENED (long to extended BFP)	LXDB	RXE		$\alpha^{7,9}$	A	SP	Db	Xi	B_2	ED05	19-34
LOAD LENGTHENED (long to extended BFP)	LXDBR	RRE		$\alpha^{7,9}$		SP	Db	Xi		B305	19-33
LOAD LENGTHENED (long to extended DFP)	LXDTR	RRF-d	TF	$\alpha^{7,9}$		SP	Dt	Xi		B3DC	20-45
LOAD LENGTHENED (long to extended HFP)	LXD	RXE		$\alpha^{7,9}$	A	SP	Da		B_2	ED25	18-15
LOAD LENGTHENED (long to extended HFP)	LXDR	RRE		$\alpha^{7,9}$		SP	Da			B325	18-15
LOAD LENGTHENED (short to extended BFP)	LXEB	RXE		$\alpha^{7,9}$	A	SP	Db	Xi	B_2	ED06	19-34
LOAD LENGTHENED (short to extended BFP)	LXEBR	RRE		$\alpha^{7,9}$		SP	Db	Xi		B306	19-33
LOAD LENGTHENED (short to extended HFP)	LXE	RXE		$\alpha^{7,9}$	A	SP	Da		B_2	ED26	18-15
LOAD LENGTHENED (short to extended HFP)	LXER	RRE		$\alpha^{7,9}$		SP	Da			B326	18-15
LOAD LENGTHENED (short to long BFP)	LDEB	RXE		$\alpha^{7,9}$	A		Db	Xi	B_2	ED04	19-34
LOAD LENGTHENED (short to long BFP)	LDEBR	RRE		$\alpha^{7,9}$			Db	Xi		B304	19-33
LOAD LENGTHENED (short to long DFP)	LDETR	RRF-d	TF	$\alpha^{7,9}$			Dt	Xi		B3D4	20-45
LOAD LENGTHENED (short to long HFP)	LDE	RXE		$\alpha^{7,9}$	A		Da		B_2	ED24	18-15
LOAD LENGTHENED (short to long HFP)	LDER	RRE		$\alpha^{7,9}$			Da			B324	18-15
LOAD LOGICAL (64←32)	LLGF	RXY-a	N		A				B_2	E316	7-278
LOAD LOGICAL (64←32)	LLGFR	RRE	N							B916	7-278
LOAD LOGICAL AND SHIFT GUARDED (64←32)	LLGFSG	RXY-a	GF	α^{12}	A	SP		B ST	B_2	E348	7-274
LOAD LOGICAL AND TRAP (64←32)	LLGFAT	RXY-a	LT		A		Dc		B_2	E39D	7-279
LOAD LOGICAL AND ZERO RIGHTMOST BYTE (64←32)	LLZRGF	RXY-a	LZ		A				B_2	E33A	7-279
LOAD LOGICAL CHARACTER (32←8)	LLC	RXY-a	EI		A				B_2	E394	7-279
LOAD LOGICAL CHARACTER (32←8)	LLCR	RRE	EI							B994	7-279
LOAD LOGICAL CHARACTER (64←8)	LLGC	RXY-a	N		A				B_2	E390	7-279
LOAD LOGICAL CHARACTER (64←8)	LLGCR	RRE	EI							B984	7-279
LOAD LOGICAL CHARACTER HIGH (32←8)	LLCH	RXY-a	HW		A				B_2	E3C2	7-280
LOAD LOGICAL HALFWORD (32←16)	LLH	RXY-a	EI		A				B_2	E395	7-280
LOAD LOGICAL HALFWORD (32←16)	LLHR	RRE	EI							B995	7-280
LOAD LOGICAL HALFWORD (64←16)	LLGH	RXY-a	N		A				B_2	E391	7-280
LOAD LOGICAL HALFWORD (64←16)	LLGHR	RRE	EI							B985	7-280
LOAD LOGICAL HALFWORD HIGH (32←16)	LLHH	RXY-a	HW		A				B_2	E3C6	7-281
LOAD LOGICAL HALFWORD RELATIVE LONG (32←16)	LLHRL	RIL-b	GE		A*					C42	7-280
LOAD LOGICAL HALFWORD RELATIVE LONG (64←16)	LLGHRL	RIL-b	GE		A*					C46	7-280
LOAD LOGICAL IMMEDIATE (high high)	LLIHH	RI-a	N							A5C	7-281
LOAD LOGICAL IMMEDIATE (high low)	LLIHL	RI-a	N							A5D	7-281
LOAD LOGICAL IMMEDIATE (high)	LLIHF	RIL-a	EI							C0E	7-281
LOAD LOGICAL IMMEDIATE (low high)	LLILH	RI-a	N							A5E	7-281
LOAD LOGICAL IMMEDIATE (low low)	LLILL	RI-a	N							A5F	7-281
LOAD LOGICAL IMMEDIATE (low)	LLILF	RIL-a	EI							C0F	7-281
LOAD LOGICAL RELATIVE LONG (64←32)	LLGFRL	RIL-b	GE		A*	SP				C4E	7-278

Figure B-1. Instructions Arranged by Name (Part 11 of 23)

Name	Mnemonic	Characteristics																Opcode	Page
LOAD LOGICAL THIRTY ONE BITS (64←31)	LLGT	RXY-a		N		A											B_2	E317	7-282
LOAD LOGICAL THIRTY ONE BITS (64←31)	LLGTR	RRE		N														B917	7-281
LOAD LOGICAL THIRTY ONE BITS AND TRAP (64←31)	LLGTAT	RXY-a		LT		A			Dc								B_2	E39C	7-282
LOAD MULTIPLE (32)	LM	RS-a				A											B_2	98	7-282
LOAD MULTIPLE (32)	LMY	RSY-a		LD		A											B_2	EB98	7-282
LOAD MULTIPLE (64)	LMG	RSY-a		N		A											B_2	EB04	7-282
LOAD MULTIPLE DISJOINT (64←32&32)	LMD	SS-e		N	¤[9]	A										B_2	B_4	EF	7-283
LOAD MULTIPLE HIGH (32)	LMH	RSY-a		N		A											B_2	EB96	7-283
LOAD NEGATIVE (32)	LNR	RR	C															11	7-283
LOAD NEGATIVE (64)	LNGR	RRE	C	N														B901	7-283
LOAD NEGATIVE (64←32)	LNGFR	RRE	C	N														B911	7-284
LOAD NEGATIVE (extended BFP)	LNXBR	RRE	C		¤[7,9]		SP	Db										B341	19-34
LOAD NEGATIVE (extended HFP)	LNXR	RRE	C		¤[7,9]		SP	Da										B361	18-16
LOAD NEGATIVE (long BFP)	LNDBR	RRE	C		¤[7,9]			Db										B311	19-34
LOAD NEGATIVE (long HFP)	LNDR	RR	C		¤[7,9]			Da										21	18-16
LOAD NEGATIVE (long)	LNDFR	RRE		FS	¤[7,9]			Da										B371	9-34
LOAD NEGATIVE (short BFP)	LNEBR	RRE	C		¤[7,9]			Db										B301	19-34
LOAD NEGATIVE (short HFP)	LNER	RR	C		¤[7,9]			Da										31	18-16
LOAD ON CONDITION (32)	LOC	RSY-b		L1		A											B_2	EBF2	7-284
LOAD ON CONDITION (32)	LOCR	RRF-c		L1		A												B9F2	7-284
LOAD ON CONDITION (64)	LOCG	RSY-b		L1		A											B_2	EBE2	7-284
LOAD ON CONDITION (64)	LOCGR	RRF-c		L1		A												B9E2	7-284
LOAD PAGE TABLE ENTRY ADDRESS	LPTEA	RRF-b	C	D2	P	A[1]*	SP	SO									R_2	B9AA	10-50
LOAD PAIR DISJOINT (32)	LPD	SSF	C	IA	¤[9]	A	SP									B_1	B_2	C84	7-285
LOAD PAIR DISJOINT (64)	LPDG	SSF	C	IA	¤[9]	A	SP									B_1	B_2	C85	7-285
LOAD PAIR FROM QUADWORD (64&64←128)	LPQ	RXY-a		N	¤[9]	A	SP										B_2	E38F	7-286
LOAD POSITIVE (32)	LPR	RR	C						IF									10	7-287
LOAD POSITIVE (64)	LPGR	RRE	C	N					IF									B900	7-287
LOAD POSITIVE (64←32)	LPGFR	RRE	C	N														B910	7-287
LOAD POSITIVE (extended BFP)	LPXBR	RRE	C		¤[7,9]		SP	Db										B340	19-35
LOAD POSITIVE (extended HFP)	LPXR	RRE	C		¤[7,9]		SP	Da										B360	18-16
LOAD POSITIVE (long BFP)	LPDBR	RRE	C		¤[7,9]			Db										B310	19-35
LOAD POSITIVE (long HFP)	LPDR	RR	C		¤[7,9]			Da										20	18-16
LOAD POSITIVE (long)	LPDFR	RRE		FS	¤[7,9]			Da										B370	9-34
LOAD POSITIVE (short BFP)	LPEBR	RRE	C		¤[7,9]			Db										B300	19-35
LOAD POSITIVE (short HFP)	LPER	RR	C		¤[7,9]			Da										30	18-16
LOAD PSW	LPSW	SI	L		P	A	SP			¢							B_2	82	10-54
LOAD PSW EXTENDED	LPSWE	S	L	N	P	A	SP			¢							B_2	B2B2	10-55
LOAD REAL ADDRESS (32)	LRA	RX-a	C		P	A[1]*		SO									BP	B1	10-56
LOAD REAL ADDRESS (32)	LRAY	RXY-a	C	LD	P	A[1]*		SO									BP	E313	10-56
LOAD REAL ADDRESS (64)	LRAG	RXY-a	C	N	P	A[1]*											BP	E303	10-56
LOAD RELATIVE LONG (32)	LRL	RIL-b		GE		A	SP											C4D	7-264
LOAD RELATIVE LONG (64)	LGRL	RIL-b		GE		A*	SP											C48	7-264
LOAD RELATIVE LONG (64←32)	LGFRL	RIL-b		GE		A*	SP											C4C	7-264
LOAD REVERSED (16)	LRVH	RXY-a		N3		A											B_2	E31F	7-287
LOAD REVERSED (32)	LRV	RXY-a		N3		A											B_2	E31E	7-287
LOAD REVERSED (32)	LRVR	RRE		N3														B91F	7-287
LOAD REVERSED (64)	LRVG	RXY-a		N		A											B_2	E30F	7-287
LOAD REVERSED (64)	LRVGR	RRE		N														B90F	7-287
LOAD ROUNDED (extended to long BFP)	LDXBR	RRE			¤[7,9]		SP	Db	Xi		Xo	Xu	Xx					B345	19-35
LOAD ROUNDED (extended to long BFP)	LDXBRA	RRF-e		F	¤[7,9]		SP	Db	Xi		Xo	Xu	Xx					B345	19-35
LOAD ROUNDED (extended to long DFP)	LDXTR	RRF-e		TF	¤[7,9]		SP	Dt	Xi		Xo	Xu	Xx	Xq				B3DD	20-46
LOAD ROUNDED (extended to long HFP)	LDXR	RR			¤[7,9]		SP	Da		EO								25	18-17
LOAD ROUNDED (extended to long HFP)	LRDR	RR			¤[7,9]		SP	Da		EO								25	18-17

Figure B-1. Instructions Arranged by Name (Part 12 of 23)

Name	Mnemonic	Characteristics																		Opcode	Page
LOAD ROUNDED (extended to short BFP)	LEXBR	RRE				[α][7,9]		SP	Db	Xi		Xo	Xu	Xx						B346	19-35
LOAD ROUNDED (extended to short BFP)	LEXBRA	RRF-e		F		[α][7,9]		SP	Db	Xi		Xo	Xu	Xx						B346	19-35
LOAD ROUNDED (extended to short HFP)	LEXR	RRE				[α][7,9]		SP	Da		EO									B366	18-17
LOAD ROUNDED (long to short BFP)	LEDBR	RRE				[α][7,9]			Db	Xi		Xo	Xu	Xx						B344	19-35
LOAD ROUNDED (long to short BFP)	LEDBRA	RRF-e		F		[α][7,9]		SP	Db	Xi		Xo	Xu	Xx						B344	19-35
LOAD ROUNDED (long to short DFP)	LEDTR	RRF-e		TF		[α][7,9]			Dt	Xi		Xo	Xu	Xx	Xq					B3D5	20-46
LOAD ROUNDED (long to short HFP)	LEDR	RR				[α][7,9]			Da		EO									35	18-17
LOAD ROUNDED (long to short HFP)	LRER	RR				[α][7,9]			Da		EO									35	18-17
LOAD USING REAL ADDRESS (32)	LURA	RRE				P	[A][1]	SP												B24B	10-60
LOAD USING REAL ADDRESS (64)	LURAG	RRE		N		P	[A][1]	SP												B905	10-60
LOAD ZERO (extended)	LZXR	RRE				[α][7,9]		SP	Da											B376	9-35
LOAD ZERO (long)	LZDR	RRE				[α][7,9]			Da											B375	9-35
LOAD ZERO (short)	LZER	RRE				[α][7,9]			Da											B374	9-35
MODIFY STACKED STATE	MSTA	RRE				[α][1]	[A][1]*	SP	SE								ST			B247	10-61
MODIFY SUBCHANNEL	MSCH	S	C			P	A	SP	OP		¢	GS							B_2	B232	14-7
MONITOR CALL	MC	SI				[α][4,8,9]		SP				ME								AF	7-288
MOVE (16←16)	MVHHI	SIL		GE			A										ST	B_1		E544	7-289
MOVE (32←16)	MVHI	SIL		GE			A										ST	B_1		E54C	7-289
MOVE (64←16)	MVGHI	SIL		GE			A										ST	B_1		E548	7-289
MOVE (character)	MVC	SS-a				[α][9]	A										ST	B_1	B_2	D2	7-289
MOVE (immediate)	MVI	SI					A										ST	B_1		92	7-289
MOVE (immediate)	MVIY	SIY		LD			A										ST	B_1		EB52	7-289
MOVE INVERSE	MVCIN	SS-a				[α][9]	A										ST	B_1	B_2	E8	7-290
MOVE LONG	MVCL	RR	C			[α][9]	A	SP	II								ST	R_1	R_2	0E	7-290
MOVE LONG EXTENDED	MVCLE	RS-a	C			[α][9]	A	SP	IC								ST	R_1	R_3	A8	7-294
MOVE LONG UNICODE	MVCLU	RSY-a	C	E2		[α][9]	A	SP	IC								ST	R_1	R_3	EB8E	7-297
MOVE NUMERICS	MVN	SS-a				[α][9]	A										ST	B_1	B_2	D1	7-301
MOVE PAGE	MVPG	RRE	C			Q	A	SP				G0					ST	R_1	R_2	B254	10-62
MOVE STRING	MVST	RRE	C			[α][9]	A	SP	IC			G0					ST	R_1	R_2	B255	7-301
MOVE TO PRIMARY	MVCP	SS-d	C			Q	A		SO		¢						ST			DA	10-64
MOVE TO SECONDARY	MVCS	SS-d	C			Q	A		SO		¢						ST			DB	10-64
MOVE WITH DESTINATION KEY	MVCDK	SSE				Q	A					GM					ST	B_1	B_2	E50F	10-66
MOVE WITH KEY	MVCK	SS-d	C			Q	A										ST	B_1	B_2	D9	10-67
MOVE WITH OFFSET	MVO	SS-b				[α][9]	A										ST	B_1	B_2	F1	7-302
MOVE WITH OPTIONAL SPECIFICATIONS	MVCOS	SSF	C	MO		Q	A		SO			G0					ST	B†	B‡	C80	10-68
MOVE WITH SOURCE KEY	MVCSK	SSE				Q	A					GM					ST	B_1	B_2	E50E	10-71
MOVE ZONES	MVZ	SS-a				[α][9]	A										ST	B_1	B_2	D3	7-303
MULTIPLY (128←64)	MG	RXY-a		MI2			A	SP											B_2	E384	7-304
MULTIPLY (128←64)	MGRK	RRF-a		MI2				SP												B9EC	7-304
MULTIPLY (64←32)	M	RX-a					A	SP											B_2	5C	7-304
MULTIPLY (64←32)	MFY	RXY-a		GE			A	SP											B_2	E35C	7-304
MULTIPLY (64←32)	MR	RR						SP												1C	7-304
MULTIPLY (extended BFP)	MXBR	RRE				[α][7,9]		SP	Db	Xi		Xo	Xu	Xx						B34C	19-37
MULTIPLY (extended DFP)	MXTR	RRF-a		TF		[α][7,9]		SP	Dt	Xi		Xo	Xu	Xx						B3D8	20-47
MULTIPLY (extended DFP)	MXTRA	RRF-a		F		[α][7,9]		SP	Dt	Xi		Xo	Xu	Xx	Xq					B3D8	20-48
MULTIPLY (extended HFP)	MXR	RR				[α][7,9]		SP	Da		EU	EO								26	18-17
MULTIPLY (long BFP)	MDB	RXE				[α][7,9]	A		Db	Xi		Xo	Xu	Xx					B_2	ED1C	19-37
MULTIPLY (long BFP)	MDBR	RRE				[α][7,9]			Db	Xi		Xo	Xu	Xx						B31C	19-37
MULTIPLY (long DFP)	MDTR	RRF-a		TF		[α][7,9]			Dt	Xi		Xo	Xu	Xx						B3D0	20-47
MULTIPLY (long DFP)	MDTRA	RRF-a		F		[α][7,9]			Dt	Xi		Xo	Xu	Xx	Xq					B3D0	20-48
MULTIPLY (long HFP)	MD	RX-a				[α][7,9]	A		Da		EU	EO							B_2	6C	18-18
MULTIPLY (long HFP)	MDR	RR				[α][7,9]			Da		EU	EO								2C	18-17
MULTIPLY (long to extended BFP)	MXDB	RXE				[α][7,9]	A	SP	Db	Xi									B_2	ED07	19-37
MULTIPLY (long to extended BFP)	MXDBR	RRE				[α][7,9]		SP	Db	Xi										B307	19-37
MULTIPLY (long to extended HFP)	MXD	RX-a				[α][7,9]	A	SP	Da		EU	EO							B_2	67	18-18

Figure B-1. Instructions Arranged by Name (Part 13 of 23)

Name	Mnemonic	Characteristics																Opcode	Page
MULTIPLY (long to extended HFP)	MXDR	RR			$\sigma^{7,9}$		SP	Da	EU	EO								27	18-17
MULTIPLY (short BFP)	MEEB	RXE			$\sigma^{7,9}$	A		Db	Xi		Xo	Xu	Xx				B_2	ED17	19-37
MULTIPLY (short BFP)	MEEBR	RRE			$\sigma^{7,9}$			Db	Xi		Xo	Xu	Xx					B317	19-37
MULTIPLY (short HFP)	MEE	RXE			$\sigma^{7,9}$	A		Da	EU	EO							B_2	ED37	18-18
MULTIPLY (short HFP)	MEER	RRE			$\sigma^{7,9}$			Da	EU	EO								B337	18-17
MULTIPLY (short to long BFP)	MDEB	RXE			$\sigma^{7,9}$	A		Db	Xi								B_2	ED0C	19-37
MULTIPLY (short to long BFP)	MDEBR	RRE			$\sigma^{7,9}$			Db	Xi									B30C	19-37
MULTIPLY (short to long HFP)	MDE	RX-a			$\sigma^{7,9}$	A		Da	EU	EO							B_2	7C	18-18
MULTIPLY (short to long HFP)	MDER	RR			$\sigma^{7,9}$			Da	EU	EO								3C	18-17
MULTIPLY (short to long HFP)	ME	RX-a			$\sigma^{7,9}$	A		Da	EU	EO							B_2	7C	18-18
MULTIPLY (short to long HFP)	MER	RR			$\sigma^{7,9}$			Da	EU	EO								3C	18-18
MULTIPLY & ADD UNNORMALIZED (long to ext. HFP)	MAY	RXF	UE		$\sigma^{7,9}$	A		Da									B_2	ED3A	18-20
MULTIPLY & ADD UNNORMALIZED (long to ext. HFP)	MAYR	RRD	UE		$\sigma^{7,9}$			Da										B33A	18-20
MULTIPLY AND ADD (long BFP)	MADB	RXF			$\sigma^{7,9}$	A		Db	Xi		Xo	Xu	Xx				B_2	ED1E	19-38
MULTIPLY AND ADD (long BFP)	MADBR	RRD			$\sigma^{7,9}$			Db	Xi		Xo	Xu	Xx					B31E	19-38
MULTIPLY AND ADD (long HFP)	MAD	RXF	HM		$\sigma^{7,9}$	A		Da	EU	EO							B_2	ED3E	18-19
MULTIPLY AND ADD (long HFP)	MADR	RRD	HM		$\sigma^{7,9}$			Da	EU	EO								B33E	18-19
MULTIPLY AND ADD (short BFP)	MAEB	RXF			$\sigma^{7,9}$	A		Db	Xi		Xo	Xu	Xx				B_2	ED0E	19-38
MULTIPLY AND ADD (short BFP)	MAEBR	RRD			$\sigma^{7,9}$			Db	Xi		Xo	Xu	Xx					B30E	19-38
MULTIPLY AND ADD (short HFP)	MAE	RXF	HM		$\sigma^{7,9}$	A		Da	EU	EO							B_2	ED2E	18-19
MULTIPLY AND ADD (short HFP)	MAER	RRD	HM		$\sigma^{7,9}$			Da	EU	EO								B32E	18-19
MULTIPLY AND ADD UNNRM. (long to ext. high HFP)	MAYH	RXF	UE		$\sigma^{7,9}$	A		Da									B_2	ED3C	18-20
MULTIPLY AND ADD UNNRM. (long to ext. high HFP)	MAYHR	RRD	UE		$\sigma^{7,9}$			Da										B33C	18-20
MULTIPLY AND ADD UNNRM. (long to ext. low HFP)	MAYL	RXF	UE		$\sigma^{7,9}$	A		Da									B_2	ED38	18-20
MULTIPLY AND ADD UNNRM. (long to ext. low HFP)	MAYLR	RRD	UE		$\sigma^{7,9}$			Da										B338	18-20
MULTIPLY AND SUBTRACT (long BFP)	MSDB	RXF			$\sigma^{7,9}$	A		Db	Xi		Xo	Xu	Xx				B_2	ED1F	19-38
MULTIPLY AND SUBTRACT (long BFP)	MSDBR	RRD			$\sigma^{7,9}$			Db	Xi		Xo	Xu	Xx					B31F	19-38
MULTIPLY AND SUBTRACT (long HFP)	MSD	RXF	HM		$\sigma^{7,9}$	A		Da	EU	EO							B_2	ED3F	18-19
MULTIPLY AND SUBTRACT (long HFP)	MSDR	RRD	HM		$\sigma^{7,9}$			Da	EU	EO								B33F	18-19
MULTIPLY AND SUBTRACT (short BFP)	MSEB	RXF			$\sigma^{7,9}$	A		Db	Xi		Xo	Xu	Xx				B_2	ED0F	19-38
MULTIPLY AND SUBTRACT (short BFP)	MSEBR	RRD			$\sigma^{7,9}$			Db	Xi		Xo	Xu	Xx					B30F	19-38
MULTIPLY AND SUBTRACT (short HFP)	MSE	RXF	HM		$\sigma^{7,9}$	A		Da	EU	EO							B_2	ED2F	18-19
MULTIPLY AND SUBTRACT (short HFP)	MSER	RRD	HM		$\sigma^{7,9}$			Da	EU	EO								B32F	18-19
MULTIPLY DECIMAL	MP	SS-b			σ^{9}	A	SP	Dg							ST	B_1	B_2	FC	8-12
MULTIPLY HALFWORD (32←16)	MH	RX-a				A											B_2	4C	7-305
MULTIPLY HALFWORD (32←16)	MHY	RXY-a	GE			A											B_2	E37C	7-305
MULTIPLY HALFWORD (64←16)	MGH	RXY-a	MI2			A											B_2	E33C	7-305
MULTIPLY HALFWORD IMMEDIATE (32←16)	MHI	RI-a																A7C	7-305
MULTIPLY HALFWORD IMMEDIATE (64←16)	MGHI	RI-a	N															A7D	7-305
MULTIPLY LOGICAL (128←64)	MLG	RXY-a	N			A	SP										B_2	E386	7-306
MULTIPLY LOGICAL (128←64)	MLGR	RRE	N				SP											B986	7-306
MULTIPLY LOGICAL (64←32)	ML	RXY-a	N3			A	SP										B_2	E396	7-306
MULTIPLY LOGICAL (64←32)	MLR	RRE	N3				SP											B996	7-305
MULTIPLY SINGLE (32)	MS	RX-a				A											B_2	71	7-307
MULTIPLY SINGLE (32)	MSC	RXY-a	C	MI2		A			IF								B_2	E353	7-307
MULTIPLY SINGLE (32)	MSR	RRE																B252	7-307
MULTIPLY SINGLE (32)	MSRKC	RRF-a	C	MI2					IF									B9FD	7-307
MULTIPLY SINGLE (32)	MSY	RXY-a		LD		A											B_2	E351	7-307
MULTIPLY SINGLE (64)	MSG	RXY-a		N		A											B_2	E30C	7-307
MULTIPLY SINGLE (64)	MSGC	RXY-a	C	MI2		A			IF								B_2	E383	7-307

Figure B-1. Instructions Arranged by Name (Part 14 of 23)

Name	Mnemonic	Characteristics															Op-code	Page
MULTIPLY SINGLE (64)	MSGR	RRE		N													B90C	7-307
MULTIPLY SINGLE (64)	MSGRKC	RRF-a	C	MI2					IF								B9ED	7-307
MULTIPLY SINGLE (64←32)	MSGF	RXY-a		N		A										B_2	E31C	7-307
MULTIPLY SINGLE (64←32)	MSGFR	RRE		N													B91C	7-307
MULTIPLY SINGLE IMMEDIATE (32)	MSFI	RIL-a		GE													C21	7-307
MULTIPLY SINGLE IMMEDIATE (64←32)	MSGFI	RIL-a		GE													C20	7-307
MULTIPLY UNNORM. (long to ext. high HFP)	MYH	RXF		UE	¤[7,9]	A			Da							B_2	ED3D	18-22
MULTIPLY UNNORM. (long to ext. high HFP)	MYHR	RRD		UE	¤[7,9]				Da								B33D	18-22
MULTIPLY UNNORM. (long to ext. low HFP)	MYL	RXF		UE	¤[7,9]	A			Da							B_2	ED39	18-22
MULTIPLY UNNORM. (long to ext. low HFP)	MYLR	RRD		UE	¤[7,9]				Da								B339	18-22
MULTIPLY UNNORMALIZED (long to ext. HFP)	MY	RXF		UE	¤[7,9]	A	SP		Da							B_2	ED3B	18-22
MULTIPLY UNNORMALIZED (long to ext. HFP)	MYR	RRD		UE	¤[7,9]		SP		Da								B33B	18-22
NEXT INSTRUCTION ACCESS INTENT	NIAI	IE		EH													B2FA	7-308
NONTRANSACTIONAL STORE (64)	NTSTG	RXY-a		TX	¤[9]	A	SP							ST		B_2	E325	7-310
OR (32)	O	RX-a	C			A										B_2	56	7-310
OR (32)	OR	RR	C														16	7-310
OR (32)	ORK	RRF-a	C	DO													B9F6	7-310
OR (32)	OY	RXY-a	C	LD		A										B_2	E356	7-310
OR (64)	OG	RXY-a	C	N		A										B_2	E381	7-311
OR (64)	OGR	RRE	C	N													B981	7-310
OR (64)	OGRK	RRF-a	C	DO													B9E6	7-310
OR (character)	OC	SS-a	C		¤[9]	A								ST	B_1	B_2	D6	7-311
OR (immediate)	OI	SI	C			A								ST	B_1		96	7-311
OR (immediate)	OIY	SIY	C	LD		A								ST	B_1		EB56	7-311
OR IMMEDIATE (high high)	OIHH	RI-a	C	N													A58	7-312
OR IMMEDIATE (high low)	OIHL	RI-a	C	N													A59	7-312
OR IMMEDIATE (high)	OIHF	RIL-a	C	EI													C0C	7-312
OR IMMEDIATE (low high)	OILH	RI-a	C	N													A5A	7-312
OR IMMEDIATE (low low)	OILL	RI-a	C	N													A5B	7-312
OR IMMEDIATE (low)	OILF	RIL-a	C	EI													C0D	7-312
PACK	PACK	SS-b			¤[9]	A								ST	B_1	B_2	F2	7-312
PACK ASCII	PKA	SS-f		E2	¤[9]	A	SP							ST	B_1	B_2	E9	7-313
PACK UNICODE	PKU	SS-f		E2	¤[9]	A	SP							ST	B_1	B_2	E1	7-314
PAGE IN	PGIN	RRE	C	ES	P	A[1]				¢							B22E	10-72
PAGE OUT	PGOUT	RRE	C	ES	P	A[1]				¢							B22F	10-73
PERFORM CRYPTOGRAPHIC COMPUTATION	PCC	RRE	C	M4	¤[5,9]	A	SP		IC			GM	I1	ST			B92C	7-315
PERFORM CRYPTOGRAPHIC KEY MGMT. OPERATIONS	PCKMO	RRE		M3	P	A	SP					GM		ST			B928	10-74
PERFORM FLOATING-POINT OPERATION	PFPO	E		PF	¤[7-9]		SP		Da	Xi	X0	GM	Xu Xx Xq				010A	9-35
PERFORM FRAME MANAGEMENT FUNCTION	PFMF	RRE		ED1	P	A[1]	SP		II		¢[3]						B9AF	10-77
PERFORM LOCKED OPERATION	PLO	SS-e	C		¤[1]	A	SP				$	GM		ST		FC	EE	7-331
PERFORM PROCESSOR ASSIST	PPA	RRF-c		PA	¤[1]												B2E8	7-345
PERFORM RANDOM NUMBER OPERATION	PRNO	RRE	C	M5	¤[5,9]	A	SP		IC	Dg		GM	I1	ST	R_1	R_2	B93C	7-346
PERFORM TIMING FACILITY FUNCTION	PTFF	E	C	TS	Q	A	SP					GM		ST			0104	10-81
PERFORM TOPOLOGY FUNCTION	PTF	RRE	C	CT	P		SP										B9A2	10-89
POPULATION COUNT	POPCNT	RRE	C	PK													B9E1	7-359
PREFETCH DATA	PFD	RXY-b		GE	¤[9,11]											B_2	E336	7-359
PREFETCH DATA RELATIVE LONG	PFDRL	RIL-c		GE	¤[9,11]												C62	7-359
PROGRAM CALL	PC	S			Q	A[1]*			Z[1]	T	¢	GM		B	ST		B218	10-90
PROGRAM RETURN	PR	E	L		¤[1]	A[1]*	SP		Z[4]	T	¢[2]			B	ST		0101	10-103
PROGRAM TRANSFER	PT	RRE			Q	A[1]*	SP		Z[2]	T	¢			B			B228	10-107
PROGRAM TRANSFER WITH INSTANCE	PTI	RRE		RA	Q	A[1]*	SP		Z[6]	T	¢			B			B99E	10-107
PURGE ALB	PALB	RRE			P						$						B248	10-116
PURGE TLB	PTLB	S			P						$						B20D	10-116
QUANTIZE (extended DFP)	QAXTR	RRF-b		TF	¤[7,9]		SP		Dt	Xi			Xx Xq				B3FD	20-49

Figure B-1. Instructions Arranged by Name (Part 15 of 23)

Name	Mnemonic	Characteristics																Opcode	Page
QUANTIZE (long DFP)	QADTR	RRF-b		TF	¤[7,9]			Dt	Xi			Xx	Xq					B3F5	20-49
REROUND (extended DFP)	RRXTR	RRF-b		TF	¤[7,9]		SP	Dt	Xi			Xx	Xq					B3FF	20-52
REROUND (long DFP)	RRDTR	RRF-b		TF	¤[7,9]			Dt	Xi			Xx	Xq					B3F7	20-52
RESET CHANNEL PATH	RCHP	S	C		P													B23B	14-9
RESET REFERENCE BIT EXTENDED	RRBE	RRE	C		P	A[1]*												B22A	10-116
RESET REFERENCE BITS MULTIPLE	RRBM	RRE		RB	P	A[1]*												B9AE	10-117
RESUME PROGRAM	RP	S	L		Q	A	SP	WE	T					B		B_2		B277	10-117
RESUME SUBCHANNEL	RSCH	S	C		P			OP		¢	GS							B238	14-10
ROTATE LEFT SINGLE LOGICAL (32)	RLL	RSY-a		N3														EB1D	7-361
ROTATE LEFT SINGLE LOGICAL (64)	RLLG	RSY-a		N														EB1C	7-361
ROTATE THEN AND SELECTED BITS (64)	RNSBG	RIE-f	C	GE														EC54	7-361
ROTATE THEN EXCLUSIVE OR SELECT. BITS (64)	RXSBG	RIE-f	C	GE														EC57	7-362
ROTATE THEN INSERT SELECTED BITS (64)	RISBG	RIE-f	C	GE														EC55	7-363
ROTATE THEN INSERT SELECTED BITS (64)	RISBGN	RIE-f		MI1														EC59	7-363
ROTATE THEN INSERT SELECTED BITS HIGH (64)	RISBHG	RIE-f		HW														EC5D	7-364
ROTATE THEN INSERT SELECTED BITS LOW (64)	RISBLG	RIE-f		HW														EC51	7-365
ROTATE THEN OR SELECTED BITS (64)	ROSBG	RIE-f	C	GE														EC56	7-362
SEARCH STRING	SRST	RRE	C		¤[9]	A	SP	IC			G0					R_2		B25E	7-366
SEARCH STRING UNICODE	SRSTU	RRE	C	E3	¤[9]	A	SP	IC			G0				R_1	R_2		B9BE	7-368
SET ACCESS	SAR	RRE			¤[6]										U_1			B24E	7-370
SET ADDRESS LIMIT	SAL	S			P			OP		¢	G1							B237	14-12
SET ADDRESS SPACE CONTROL	SAC	S			Q		SP	SW		¢								B219	10-120
SET ADDRESS SPACE CONTROL FAST	SACF	S			Q		SP	SW										B279	10-120
SET ADDRESSING MODE (24)	SAM24	E		N3	¤[3,9]		SP		T									010C	7-370
SET ADDRESSING MODE (31)	SAM31	E		N3	¤[3,9]		SP		T									010D	7-370
SET ADDRESSING MODE (64)	SAM64	E		N	¤[3,9]				T									010E	7-370
SET BFP ROUNDING MODE (2 bit)	SRNM	S			¤[7,9]			Db										B299	9-47
SET BFP ROUNDING MODE (3 bit)	SRNMB	S		F	¤[7,9]		SP	Db										B2B8	9-47
SET CHANNEL MONITOR	SCHM	S			P			OP		¢	GM							B23C	14-13
SET CLOCK	SCK	S	C		P	A	SP									B_2		B204	10-121
SET CLOCK COMPARATOR	SCKC	S			P	A	SP									B_2		B206	10-122
SET CLOCK PROGRAMMABLE FIELD	SCKPF	E			P		SP				G0							0107	10-123
SET CPU TIMER	SPT	S			P	A	SP									B_2		B208	10-123
SET DFP ROUNDING MODE	SRNMT	S		TR	¤[7,9]			Dt										B2B9	9-47
SET FPC	SFPC	RRE			¤[7,9]		SP	Db										B384	9-47
SET FPC AND SIGNAL	SFASR	RRE		XF	¤[7,9]		SP	Dt	Xg									B385	9-48
SET PREFIX	SPX	S			P	A	SP			$						B_2		B210	10-123
SET PROGRAM MASK	SPM	RR	L															04	7-371
SET PSW KEY FROM ADDRESS	SPKA	S			Q													B20A	10-124
SET SECONDARY ASN	SSAR	RRE			¤[1]	A[1]*		Z[3]	T	¢								B225	10-125
SET SECONDARY ASN WITH INSTANCE	SSAIR	RRE		RA	¤[1]	A[1]*		Z[7]	T	¢								B99F	10-125
SET STORAGE KEY EXTENDED	SSKE	RRF-c	C[1]		P	A[1]*		II		¢								B22B	10-130
SET SYSTEM MASK	SSM	SI			P	A	SP	SO								B_2		80	10-133
SHIFT AND ROUND DECIMAL	SRP	SS-c	C		¤[9]	A		Dg	DF					ST	B_1	B_2		F0	8-12
SHIFT LEFT DOUBLE (64)	SLDA	RS-a	C				SP		IF									8F	7-371
SHIFT LEFT DOUBLE LOGICAL (64)	SLDL	RS-a					SP											8D	7-372
SHIFT LEFT SINGLE (32)	SLA	RS-a	C						IF									8B	7-372
SHIFT LEFT SINGLE (32)	SLAK	RSY-a	C	DO					IF									EBDD	7-372
SHIFT LEFT SINGLE (64)	SLAG	RSY-a	C	N					IF									EB0B	7-372
SHIFT LEFT SINGLE LOGICAL (32)	SLL	RS-a																89	7-373
SHIFT LEFT SINGLE LOGICAL (32)	SLLK	RSY-a		DO														EBDF	7-373
SHIFT LEFT SINGLE LOGICAL (64)	SLLG	RSY-a		N														EB0D	7-373
SHIFT RIGHT DOUBLE (64)	SRDA	RS-a	C				SP											8E	7-374
SHIFT RIGHT DOUBLE LOGICAL (64)	SRDL	RS-a					SP											8C	7-374

Figure B-1. Instructions Arranged by Name (Part 16 of 23)

Name	Mne-monic	Characteristics																Op-code	Page
SHIFT RIGHT SINGLE (32)	SRA	RS-a	C															8A	7-375
SHIFT RIGHT SINGLE (32)	SRAK	RSY-a	C	DO														EBDC	7-375
SHIFT RIGHT SINGLE (64)	SRAG	RSY-a	C	N														EB0A	7-375
SHIFT RIGHT SINGLE LOGICAL (32)	SRL	RS-a																88	7-376
SHIFT RIGHT SINGLE LOGICAL (32)	SRLK	RSY-a		DO														EBDE	7-376
SHIFT RIGHT SINGLE LOGICAL (64)	SRLG	RSY-a		N														EB0C	7-376
SHIFT SIGNIFICAND LEFT (extended DFP)	SLXT	RXF		TF	¤[7,9]		SP	Dt										ED48	20-54
SHIFT SIGNIFICAND LEFT (long DFP)	SLDT	RXF		TF	¤[7,9]			Dt										ED40	20-54
SHIFT SIGNIFICAND RIGHT (extended DFP)	SRXT	RXF		TF	¤[7,9]		SP	Dt										ED49	20-54
SHIFT SIGNIFICAND RIGHT (long DFP)	SRDT	RXF		TF	¤[7,9]			Dt										ED41	20-54
SIGNAL PROCESSOR	SIGP	RS-a	C		P					$								AE	10-133
SQUARE ROOT (extended BFP)	SQXBR	RRE			¤[7,9]		SP	Db	Xi				Xx					B316	19-40
SQUARE ROOT (extended HFP)	SQXR	RRE			¤[7,9]		SP	Da			SQ							B336	18-23
SQUARE ROOT (long BFP)	SQDB	RXE			¤[7,9]	A		Db	Xi				Xx				B_2	ED15	19-40
SQUARE ROOT (long BFP)	SQDBR	RRE			¤[7,9]			Db	Xi				Xx					B315	19-40
SQUARE ROOT (long HFP)	SQD	RXE			¤[7,9]	A		Da			SQ						B_2	ED35	18-23
SQUARE ROOT (long HFP)	SQDR	RRE			¤[7,9]			Da			SQ							B244	18-23
SQUARE ROOT (short BFP)	SQEB	RXE			¤[7,9]	A		Db	Xi				Xx				B_2	ED14	19-40
SQUARE ROOT (short BFP)	SQEBR	RRE			¤[7,9]			Db	Xi				Xx					B314	19-40
SQUARE ROOT (short HFP)	SQE	RXE			¤[7,9]	A		Da			SQ						B_2	ED34	18-23
SQUARE ROOT (short HFP)	SQER	RRE			¤[7,9]			Da			SQ							B245	18-23
START SUBCHANNEL	SSCH	S	C		P	A	SP	OP		¢	GS						B_2	B233	14-15
STORE (32)	ST	RX-a				A										ST	B_2	50	7-376
STORE (32)	STY	RXY-a		LD		A										ST	B_2	E350	7-377
STORE (64)	STG	RXY-a		N		A										ST	B_2	E324	7-377
STORE (long)	STD	RX-a			¤[7,9]	A		Da								ST	B_2	60	9-48
STORE (long)	STDY	RXY-a		LD	¤[7,9]	A		Da								ST	B_2	ED67	9-49
STORE (short)	STE	RX-a			¤[7,9]	A		Da								ST	B_2	70	9-48
STORE (short)	STEY	RXY-a		LD	¤[7,9]	A		Da								ST	B_2	ED66	9-49
STORE ACCESS MULTIPLE	STAM	RS-a				A	SP									ST	UB	9B	7-377
STORE ACCESS MULTIPLE	STAMY	RSY-a		LD		A	SP									ST	UB	EB9B	7-377
STORE CHANNEL PATH STATUS	STCPS	S			P	A	SP			¢						ST	B_2	B23A	14-16
STORE CHANNEL REPORT WORD	STCRW	S	C		P	A	SP			¢						ST	B_2	B239	14-17
STORE CHARACTER	STC	RX-a				A										ST	B_2	42	7-378
STORE CHARACTER	STCY	RXY-a		LD		A										ST	B_2	E372	7-378
STORE CHARACTER HIGH (8)	STCH	RXY-a		HW		A										ST	B_2	E3C3	7-378
STORE CHARACTERS UNDER MASK (high)	STCMH	RSY-b		N	¤[9,11]	A										ST	B_2	EB2C	7-378
STORE CHARACTERS UNDER MASK (low)	STCM	RS-b				A										ST	B_2	BE	7-378
STORE CHARACTERS UNDER MASK (low)	STCMY	RSY-b		LD		A										ST	B_2	EB2D	7-378
STORE CLOCK	STCK	S	C		¤[8,9]	A				$						ST	B_2	B205	7-379
STORE CLOCK COMPARATOR	STCKC	S			P	A	SP									ST	B_2	B207	10-135
STORE CLOCK EXTENDED	STCKE	S	C		¤[8,9]	A				$						ST	B_2	B278	7-380
STORE CLOCK FAST	STCKF	S	C	SC	¤[8,9]	A										ST	B_2	B27C	7-379
STORE CONTROL (32)	STCTL	RS-a			P	A	SP									ST	B_2	B6	10-135
STORE CONTROL (64)	STCTG	RSY-a		N	P	A	SP									ST	B_2	EB25	10-135
STORE CPU ADDRESS	STAP	S			P	A	SP									ST	B_2	B212	10-136
STORE CPU ID	STIDP	S			P	A	SP									ST	B_2	B202	10-136
STORE CPU TIMER	STPT	S			P	A	SP									ST	B_2	B209	10-138
STORE FACILITY LIST	STFL	S		N3	P													B2B1	10-138
STORE FACILITY LIST EXTENDED	STFLE	S	C	FL	¤[1]	A	SP				G0					ST	B_2	B2B0	7-382
STORE FPC	STFPC	S			¤[7,9]	A		Db								ST	B_2	B29C	9-49
STORE GUARDED STORAGE CONTROLS	STGSC	RXY-a		GF	¤[1]	A		SO								ST	B_2	E349	7-383
STORE HALFWORD (16)	STH	RX-a				A										ST	B_2	40	7-383
STORE HALFWORD (16)	STHY	RXY-a		LD		A										ST	B_2	E370	7-384
STORE HALFWORD HIGH (16)	STHH	RXY-a		HW		A										ST	B_2	E3C7	7-384

Figure B-1. Instructions Arranged by Name (Part 17 of 23)

Name	Mne-monic	Characteristics															Op-code	Page
STORE HALFWORD RELATIVE LONG (16)	STHRL	RIL-b		GE		A*								ST			C47	7-384
STORE HIGH (32)	STFH	RXY-a		HW		A								ST		B_2	E3CB	7-384
STORE HIGH ON CONDITION	STOCFH	RSY-b		L2		A								ST		B_2	EBE1	7-386
STORE MULTIPLE (32)	STM	RS-a				A								ST		B_2	90	7-385
STORE MULTIPLE (32)	STMY	RSY-a		LD		A								ST		B_2	EB90	7-385
STORE MULTIPLE (64)	STMG	RSY-a		N		A								ST		B_2	EB24	7-385
STORE MULTIPLE HIGH (32)	STMH	RSY-a		N		A								ST		B_2	EB26	7-385
STORE ON CONDITION (32)	STOC	RSY-b		L1		A								ST		B_2	EBF3	7-385
STORE ON CONDITION (64)	STOCG	RSY-b		L1		A								ST		B_2	EBE3	7-385
STORE PAIR TO QUADWORD	STPQ	RXY-a		N	¤[9]	A	SP							ST		B_2	E38E	7-386
STORE PREFIX	STPX	S			P	A	SP							ST		B_2	B211	10-139
STORE REAL ADDRESS	STRAG	SSE		N	P	A[1]	SP							ST	B_1	BP	E502	10-139
STORE RELATIVE LONG (32)	STRL	RIL-b		GE		A*	SP							ST			C4F	7-377
STORE RELATIVE LONG (64)	STGRL	RIL-b		GE		A*	SP							ST			C4B	7-377
STORE REVERSED (16)	STRVH	RXY-a		N3		A								ST		B_2	E33F	7-387
STORE REVERSED (32)	STRV	RXY-a		N3		A								ST		B_2	E33E	7-387
STORE REVERSED (64)	STRVG	RXY-a		N		A								ST		B_2	E32F	7-387
STORE SUBCHANNEL	STSCH	S	C		P	A	SP	OP	¢	GS				ST		B_2	B234	14-18
STORE SYSTEM INFORMATION	STSI	S	C		P	A	SP			GM				ST		B_2	B27D	10-140
STORE THEN AND SYSTEM MASK	STNSM	SI			P	A								ST	B_1		AC	10-164
STORE THEN OR SYSTEM MASK	STOSM	SI			P	A	SP							ST	B_1		AD	10-164
STORE USING REAL ADDRESS (32)	STURA	RRE			P	A[1]	SP							SU			B246	10-165
STORE USING REAL ADDRESS (64)	STURG	RRE		N	P	A[1]	SP							SU			B925	10-165
SUBTRACT (32)	S	RX-a	C			A			IF							B_2	5B	7-388
SUBTRACT (32)	SR	RR	C						IF								1B	7-387
SUBTRACT (32)	SRK	RRF-a	C	DO					IF								B9F9	7-387
SUBTRACT (32)	SY	RXY-a	C	LD		A			IF							B_2	E35B	7-388
SUBTRACT (64)	SG	RXY-a	C	N		A			IF							B_2	E309	7-388
SUBTRACT (64)	SGR	RRE	C	N					IF								B909	7-387
SUBTRACT (64)	SGRK	RRF-a	C	DO					IF								B9E9	7-388
SUBTRACT (64←32)	SGF	RXY-a	C	N		A			IF							B_2	E319	7-388
SUBTRACT (64←32)	SGFR	RRE	C	N					IF								B919	7-387
SUBTRACT (extended BFP)	SXBR	RRE	C		¤[7,9]		SP	Db	Xi		Xo	Xu	Xx				B34B	19-40
SUBTRACT (extended DFP)	SXTR	RRF-a	C	TF	¤[7,9]		SP	Dt	Xi		Xo	Xu	Xx				B3DB	20-55
SUBTRACT (extended DFP)	SXTRA	RRF-a	C	F	¤[7,9]		SP	Dt	Xi		Xo	Xu	Xx	Xq			B3DB	20-55
SUBTRACT (long BFP)	SDB	RXE	C		¤[7,9]	A		Db	Xi		Xo	Xu	Xx			B_2	ED1B	19-40
SUBTRACT (long BFP)	SDBR	RRE	C		¤[7,9]			Db	Xi		Xo	Xu	Xx				B31B	19-40
SUBTRACT (long DFP)	SDTR	RRF-a	C	TF	¤[7,9]			Dt	Xi		Xo	Xu	Xx				B3D3	20-55
SUBTRACT (long DFP)	SDTRA	RRF-a	C	F	¤[7,9]			Dt	Xi		Xo	Xu	Xx	Xq			B3D3	20-55
SUBTRACT (short BFP)	SEB	RXE	C		¤[7,9]	A		Db	Xi		Xo	Xu	Xx			B_2	ED0B	19-40
SUBTRACT (short BFP)	SEBR	RRE	C		¤[7,9]			Db	Xi		Xo	Xu	Xx				B30B	19-40
SUBTRACT DECIMAL	SP	SS-b	C		¤[9]	A		Dg	DF						ST	B_1 B_2	FB	8-13
SUBTRACT HALFWORD (32←16)	SH	RX-a	C			A			IF							B_2	4B	7-388
SUBTRACT HALFWORD (32←16)	SHY	RXY-a	C	LD		A			IF							B_2	E37B	7-388
SUBTRACT HALFWORD (64←16)	SGH	RXY-a	C	MI2		A			IF							B_2	E339	7-388
SUBTRACT HIGH (32)	SHHHR	RRF-a	C	HW					IF								B9C9	7-389
SUBTRACT HIGH (32)	SHHLR	RRF-a	C	HW					IF								B9D9	7-389
SUBTRACT LOGICAL (32)	SL	RX-a	C			A										B_2	5F	7-390
SUBTRACT LOGICAL (32)	SLR	RR	C														1F	7-389
SUBTRACT LOGICAL (32)	SLRK	RRF-a	C	DO													B9FB	7-389
SUBTRACT LOGICAL (32)	SLY	RXY-a	C	LD		A										B_2	E35F	7-390
SUBTRACT LOGICAL (64)	SLG	RXY-a	C	N		A										B_2	E30B	7-390
SUBTRACT LOGICAL (64)	SLGR	RRE	C	N													B90B	7-389
SUBTRACT LOGICAL (64)	SLGRK	RRF-a	C	DO													B9EB	7-389
SUBTRACT LOGICAL (64←32)	SLGF	RXY-a	C	N		A										B_2	E31B	7-390

Figure B-1. Instructions Arranged by Name (Part 18 of 23)

Name	Mnemonic	Characteristics												Opcode	Page
SUBTRACT LOGICAL (64←32)	SLGFR	RRE	C	N										B91B	7-389
SUBTRACT LOGICAL HIGH (32)	SLHHHR	RRF-a	C	HW										B9CB	7-391
SUBTRACT LOGICAL HIGH (32)	SLHHLR	RRF-a	C	HW										B9DB	7-391
SUBTRACT LOGICAL IMMEDIATE (32)	SLFI	RIL-a	C	EI										C25	7-390
SUBTRACT LOGICAL IMMEDIATE (64←32)	SLGFI	RIL-a	C	EI										C24	7-390
SUBTRACT LOGICAL WITH BORROW (32)	SLB	RXY-a	C	N3		A							B_2	E399	7-391
SUBTRACT LOGICAL WITH BORROW (32)	SLBR	RRE	C	N3										B999	7-391
SUBTRACT LOGICAL WITH BORROW (64)	SLBG	RXY-a	C	N		A							B_2	E389	7-391
SUBTRACT LOGICAL WITH BORROW (64)	SLBGR	RRE	C	N										B989	7-391
SUBTRACT NORMALIZED (extended HFP)	SXR	RR	C		¤[7,9]		SP	Da	EU	EO	LS			37	18-24
SUBTRACT NORMALIZED (long HFP)	SD	RX-a	C		¤[7,9]	A		Da	EU	EO	LS		B_2	6B	18-24
SUBTRACT NORMALIZED (long HFP)	SDR	RR	C		¤[7,9]			Da	EU	EO	LS			2B	18-24
SUBTRACT NORMALIZED (short HFP)	SE	RX-a	C		¤[7,9]	A		Da	EU	EO	LS		B_2	7B	18-24
SUBTRACT NORMALIZED (short HFP)	SER	RR	C		¤[7,9]			Da	EU	EO	LS			3B	18-24
SUBTRACT UNNORMALIZED (long HFP)	SW	RX-a	C		¤[7,9]	A		Da		EO	LS		B_2	6F	18-25
SUBTRACT UNNORMALIZED (long HFP)	SWR	RR	C		¤[7,9]			Da		EO	LS			2F	18-25
SUBTRACT UNNORMALIZED (short HFP)	SU	RX-a	C		¤[7,9]	A		Da		EO	LS		B_2	7F	18-25
SUBTRACT UNNORMALIZED (short HFP)	SUR	RR	C		¤[7,9]			Da		EO	LS			3F	18-25
SUPERVISOR CALL	SVC	I			¤[1]					¢				0A	7-392
TEST ACCESS	TAR	RRE	C		¤[1]	A[1]*							U_1	B24C	10-165
TEST ADDRESSING MODE	TAM	E	C	N3	¤[9]									010B	7-392
TEST AND SET	TS	SI	C		¤[9]	A				$		ST	B_2	93	7-392
TEST BLOCK	TB	RRE	C		P	A[1]*		II		$	G0			B22C	10-167
TEST DATA CLASS (extended BFP)	TCXB	RXE	C		¤[7,9]		SP	Db						ED12	19-41
TEST DATA CLASS (extended DFP)	TDCXT	RXE	C	TF	¤[7,9]		SP	Dt						ED58	20-56
TEST DATA CLASS (long BFP)	TCDB	RXE	C		¤[7,9]			Db						ED11	19-41
TEST DATA CLASS (long DFP)	TDCDT	RXE	C	TF	¤[7,9]			Dt						ED54	20-56
TEST DATA CLASS (short BFP)	TCEB	RXE	C		¤[7,9]			Db						ED10	19-41
TEST DATA CLASS (short DFP)	TDCET	RXE	C	TF	¤[7,9]			Dt						ED50	20-56
TEST DATA GROUP (extended DFP)	TDGXT	RXE	C	TF	¤[7,9]		SP	Dt						ED59	20-57
TEST DATA GROUP (long DFP)	TDGDT	RXE	C	TF	¤[7,9]			Dt						ED55	20-57
TEST DATA GROUP (short DFP)	TDGET	RXE	C	TF	¤[7,9]			Dt						ED51	20-57
TEST DECIMAL	TP	RSL-a	C	E2	¤[9]	A							B_1 B_2	EBC0	8-14
TEST PENDING EXTERNAL INTERRUPTION	TPEI	RRE	C	TE	P									B9A1	10-169
TEST PENDING INTERRUPTION	TPI	S	C		P	A[1]*	SP			¢		ST	B_2	B236	14-19
TEST PROTECTION	TPROT	SSE	C		P	A[1]*							B_1	E501	10-170
TEST SUBCHANNEL	TSCH	S	C		P	A	SP	OP		¢	GS	ST	B_2	B235	14-21
TEST UNDER MASK	TM	SI	C			A							B_1	91	7-393
TEST UNDER MASK	TMY	SIY	C	LD		A							B_1	EB51	7-393
TEST UNDER MASK (high high)	TMHH	RI-a	C	N										A72	7-393
TEST UNDER MASK (high low)	TMHL	RI-a	C	N										A73	7-393
TEST UNDER MASK (low high)	TMLH	RI-a	C	N										A70	7-393
TEST UNDER MASK (low low)	TMLL	RI-a	C	N										A71	7-393
TEST UNDER MASK HIGH	TMH	RI-a	C											A70	7-393
TEST UNDER MASK LOW	TML	RI-a	C											A71	7-393
TRACE (32)	TRACE	RS-a			P	A	SP		T	¢			B_2	99	10-173
TRACE (64)	TRACG	RSY-a		N	P	A	SP		T	¢			B_2	EB0F	10-173
TRANSACTION ABORT	TABORT	S		TX	¤[9]		SP	SO		$	EX			B2FC	7-394
TRANSACTION BEGIN (constrained)	TBEGINC	SIL	C	CX	¤[9]		SP	SO		$	EX			E561	7-399
TRANSACTION BEGIN (nonconstrained)	TBEGIN	SIL	C	TX	¤[9]	A	SP	SO		$	EX	ST		E560	7-395
TRANSACTION END	TEND	S	C	TX				SO		$	EX			B2F8	7-401
TRANSLATE	TR	SS-a			¤[9]	A						ST	B_1 B_2	DC	7-402
TRANSLATE AND TEST	TRT	SS-a	C		¤[9]	A					GM		B_1 B_2	DD	7-403
TRANSLATE AND TEST EXTENDED	TRTE	RRF-c	C	PE	¤[9]	A	SP	IC				ST	RM	B9BF	7-403
TRANSLATE AND TEST REVERSE	TRTR	SS-a	C	E3	¤[9]	A					GM		B_1 B_2	D0	7-408

Figure B-1. Instructions Arranged by Name (Part 19 of 23)

Name	Mnemonic	Characteristics													Opcode	Page
TRANSLATE AND TEST REVERSE EXTENDED	TRTRE	RRF-c	C	PE	α[9]	A	SP	IC				ST	RM		B9BD	7-404
TRANSLATE EXTENDED	TRE	RRE	C		α[9]	A	SP	IC				ST	R_1	R_2	B2A5	7-409
TRANSLATE ONE TO ONE	TROO	RRF-c	C	E2	α[9]	A	SP	IC		GM		ST	RM	R_2	B993	7-411
TRANSLATE ONE TO TWO	TROT	RRF-c	C	E2	α[9]	A	SP	IC		GM		ST	RM	R_2	B992	7-411
TRANSLATE TWO TO ONE	TRTO	RRF-c	C	E2	α[9]	A	SP	IC		GM		ST	RM	R_2	B991	7-411
TRANSLATE TWO TO TWO	TRTT	RRF-c	C	E2	α[9]	A	SP	IC		GM		ST	RM	R_2	B990	7-411
TRAP	TRAP2	E			α[1]	A*		SO	T			B ST			01FF	10-174
TRAP	TRAP4	S			α[1]	A*		SO	T			B ST			B2FF	10-174
UNPACK	UNPK	SS-b			α[9]	A						ST	B_1	B_2	F3	7-416
UNPACK ASCII	UNPKA	SS-a	C	E2	α[9]	A	SP					ST	B_1	B_2	EA	7-417
UNPACK UNICODE	UNPKU	SS-a	C	E2	α[9]	A	SP					ST	B_1	B_2	E2	7-418
UPDATE TREE	UPT	E	C		α[9]	A	SP	II		GM	I4	ST			0102	7-419
VECTOR ADD	VA	VRR-c		VF	α[7,9]		SP	Dv							E7F3	22-3
VECTOR ADD COMPUTE CARRY	VACC	VRR-c		VF	α[7,9]		SP	Dv							E7F1	22-4
VECTOR ADD DECIMAL	VAP	VRI-f	C*	VD	α[7,9]		SP	Dv	Dg	DF					E671	25-3
VECTOR ADD WITH CARRY	VAC	VRR-d		VF	α[7,9]		SP	Dv							E7BB	22-4
VECTOR ADD WITH CARRY COMPUTE CARRY	VACCC	VRR-d		VF	α[7,9]		SP	Dv							E7B9	22-5
VECTOR AND	VN	VRR-c		VF	α[7,9]			Dv							E768	22-5
VECTOR AND WITH COMPLEMENT	VNC	VRR-c		VF	α[7,9]			Dv							E769	22-5
VECTOR AVERAGE	VAVG	VRR-c		VF	α[7,9]		SP	Dv							E7F2	22-6
VECTOR AVERAGE LOGICAL	VAVGL	VRR-c		VF	α[7,9]		SP	Dv							E7F0	22-6
VECTOR BIT PERMUTE	VBPERM	VRR-c		V1	α[7,9]			Dv							E785	21-4
VECTOR CHECKSUM	VCKSM	VRR-c		VF	α[7,9]			Dv							E766	22-6
VECTOR COMPARE DECIMAL	VCP	VRR-h	C	VD	α[7,9]			Dv	Dg						E677	25-4
VECTOR COMPARE EQUAL	VCEQ	VRR-b	C*	VF	α[7,9]		SP	Dv							E7F8	22-7
VECTOR COMPARE HIGH	VCH	VRR-b	C*	VF	α[7,9]		SP	Dv							E7FB	22-8
VECTOR COMPARE HIGH LOGICAL	VCHL	VRR-b	C*	VF	α[7,9]		SP	Dv							E7F9	22-9
VECTOR CONVERT TO BINARY	VCVB	VRR-i	C*	VD	α[7,9]			Dv	Dg		IF				E650	25-5
VECTOR CONVERT TO BINARY	VCVBG	VRR-i	C*	VD	α[7,9]			Dv	Dg		IF				E652	25-5
VECTOR CONVERT TO DECIMAL	VCVD	VRI-i	C*	VD	α[7,9]		SP	Dv		DF					E658	25-6
VECTOR CONVERT TO DECIMAL	VCVDG	VRI-i	C*	VD	α[7,9]		SP	Dv		DF					E65A	25-6
VECTOR COUNT LEADING ZEROS	VCLZ	VRR-a		VF	α[7,9]		SP	Dv							E753	22-10
VECTOR COUNT TRAILING ZEROS	VCTZ	VRR-a		VF	α[7,9]		SP	Dv							E752	22-10
VECTOR DIVIDE DECIMAL	VDP	VRI-f	C*	VD	α[7,9]		SP	Dv	Dg	DF	DK				E67A	25-8
VECTOR ELEMENT COMPARE	VEC	VRR-a	C	VF	α[7,9]		SP	Dv							E7DB	22-7
VECTOR ELEMENT COMPARE LOGICAL	VECL	VRR-a	C	VF	α[7,9]		SP	Dv							E7D9	22-7
VECTOR ELEMENT ROTATE AND INSERT UNDER MASK	VERIM	VRI-d		VF	α[7,9]		SP	Dv							E772	22-22
VECTOR ELEMENT ROTATE LEFT LOGICAL	VERLL	VRS-a		VF	α[7,9]		SP	Dv							E733	22-21
VECTOR ELEMENT ROTATE LEFT LOGICAL	VERLLV	VRR-c		VF	α[7,9]		SP	Dv							E773	22-21
VECTOR ELEMENT SHIFT LEFT	VESLV	VRR-c		VF	α[7,9]		SP	Dv							E770	22-23
VECTOR ELEMENT SHIFT LEFT	VESL	VRS-a		VF	α[7,9]		SP	Dv							E730	22-23
VECTOR ELEMENT SHIFT RIGHT ARITHMETIC	VESRA	VRS-a		VF	α[7,9]		SP	Dv							E73A	22-23
VECTOR ELEMENT SHIFT RIGHT ARITHMETIC	VESRAV	VRR-c		VF	α[7,9]		SP	Dv							E77A	22-23
VECTOR ELEMENT SHIFT RIGHT LOGICAL	VESRL	VRS-a		VF	α[7,9]		SP	Dv							E738	22-24
VECTOR ELEMENT SHIFT RIGHT LOGICAL	VESRLV	VRR-c		VF	α[7,9]		SP	Dv							E778	22-24
VECTOR EXCLUSIVE OR	VX	VRR-c		VF	α[7,9]			Dv							E76D	22-11
VECTOR FIND ANY ELEMENT EQUAL	VFAE	VRR-b	C*	VF	α[7,9]		SP	Dv							E782	23-1
VECTOR FIND ELEMENT EQUAL	VFEE	VRR-b	C*	VF	α[7,9]		SP	Dv							E780	23-3
VECTOR FIND ELEMENT NOT EQUAL	VFENE	VRR-b	C*	VF	α[7,9]		SP	Dv							E781	23-4
VECTOR FP ADD	VFA	VRR-c		VF	α[7,9]		SP	Dv	Xi		Xo	Xu	Xx		E7E3	24-3
VECTOR FP COMPARE AND SIGNAL SCALAR	WFK	VRR-a	C	VF	α[7,9]		SP	Dv	Xi						E7CA	24-8
VECTOR FP COMPARE EQUAL	VFCE	VRR-c	C*	VF	α[7,9]		SP	Dv	Xi						E7E8	24-9
VECTOR FP COMPARE HIGH	VFCH	VRR-c	C*	VF	α[7,9]		SP	Dv	Xi						E7EB	24-11
VECTOR FP COMPARE HIGH OR EQUAL	VFCHE	VRR-c	C*	VF	α[7,9]		SP	Dv	Xi						E7EA	24-13

Figure B-1. Instructions Arranged by Name (Part 20 of 23)

Name	Mnemonic	Characteristics																			Opcode	Page
VECTOR FP COMPARE SCALAR	WFC	VRR-a	C		VF		α[7,9]		SP	Dv	Xi										E7CB	24-7
VECTOR FP CONVERT FROM FIXED 64-BIT	VCDG	VRR-a			VF		α[7,9]		SP	Dv						Xx					E7C3	24-15
VECTOR FP CONVERT FROM LOGICAL 64-BIT	VCDLG	VRR-a			VF		α[7,9]		SP	Dv						Xx					E7C1	24-16
VECTOR FP CONVERT TO FIXED 64-BIT	VCGD	VRR-a			VF		α[7,9]		SP	Dv	Xi					Xx					E7C2	24-17
VECTOR FP CONVERT TO LOGICAL 64-BIT	VCLGD	VRR-a			VF		α[7,9]		SP	Dv	Xi					Xx					E7C0	24-19
VECTOR FP DIVIDE	VFD	VRR-c			VF		α[7,9]		SP	Dv	Xi	Xz	Xo	Xu	Xx						E7E5	24-21
VECTOR FP LOAD LENGTHENED	VFLL	VRR-a			VF		α[7,9]		SP	Dv	Xi										E7C4	24-25
VECTOR FP LOAD ROUNDED	VFLR	VRR-a			VF		α[7,9]		SP	Dv	Xi		Xo	Xu	Xx						E7C5	24-26
VECTOR FP MAXIMUM	VFMAX	VRR-c			V1		α[7,9]		SP	Dv	Xi										E7EF	24-27
VECTOR FP MINIMUM	VFMIN	VRR-c			V1		α[7,9]		SP	Dv	Xi										E7EE	24-33
VECTOR FP MULTIPLY	VFM	VRR-c			VF		α[7,9]		SP	Dv	Xi		Xo	Xu	Xx						E7E7	24-39
VECTOR FP MULTIPLY AND ADD	VFMA	VRR-e			VF		α[7,9]		SP	Dv	Xi		Xo	Xu	Xx						E78F	24-41
VECTOR FP MULTIPLY AND SUBTRACT	VFMS	VRR-e			VF		α[7,9]		SP	Dv	Xi		Xo	Xu	Xx						E78E	24-41
VECTOR FP NEGATIVE MULTIPLY AND ADD	VFNMA	VRR-e			V1		α[7,9]		SP	Dv	Xi		Xo	Xu	Xx						E79F	24-41
VECTOR FP NEGATIVE MULTIPLY AND SUBTRACT	VFNMS	VRR-e			V1		α[7,9]		SP	Dv	Xi		Xo	Xu	Xx						E79E	24-41
VECTOR FP PERFORM SIGN OPERATION	VFPSO	VRR-a			VF		α[7,9]		SP	Dv											E7CC	24-43
VECTOR FP SQUARE ROOT	VFSQ	VRR-a			VF		α[7,9]		SP	Dv	Xi					Xx					E7CE	24-44
VECTOR FP SUBTRACT	VFS	VRR-c			VF		α[7,9]		SP	Dv	Xi		Xo	Xu	Xx						E7E2	24-45
VECTOR FP TEST DATA CLASS IMMEDIATE	VFTCI	VRI-e	C		VF		α[7,9]		SP	Dv											E74A	24-46
VECTOR GALOIS FIELD MULTIPLY SUM	VGFM	VRR-c			VF		α[7,9]		SP	Dv											E7B4	22-11
VECTOR GALOIS FIELD MULTIPLY SUM AND ACCUMULATE	VGFMA	VRR-d			VF		α[7,9]		SP	Dv											E7BC	22-12
VECTOR GATHER ELEMENT (32)	VGEF	VRV			VF		α[7,9]	A	SP	Dv										B_2	E713	21-4
VECTOR GATHER ELEMENT (64)	VGEG	VRV			VF		α[7,9]	A	SP	Dv										B_2	E712	21-4
VECTOR GENERATE BYTE MASK	VGBM	VRI-a			VF		α[7,9]			Dv											E744	21-5
VECTOR GENERATE MASK	VGM	VRI-b			VF		α[7,9]		SP	Dv											E746	21-5
VECTOR ISOLATE STRING	VISTR	VRR-a	C*		VF		α[7,9]		SP	Dv											E75C	23-5
VECTOR LOAD	VL	VRX			VF		α[7,9]	A		Dv										B_2	E706	21-6
VECTOR LOAD	VLR	VRR-a			VF		α[7,9]			Dv											E756	21-6
VECTOR LOAD AND REPLICATE	VLREP	VRX			VF		α[7,9]	A	SP	Dv										B_2	E705	21-6
VECTOR LOAD COMPLEMENT	VLC	VRR-a			VF		α[7,9]		SP	Dv											E7DE	22-12
VECTOR LOAD ELEMENT (16)	VLEH	VRX			VF		α[7,9]	A	SP	Dv										B_2	E701	21-7
VECTOR LOAD ELEMENT (32)	VLEF	VRX			VF		α[7,9]	A	SP	Dv										B_2	E703	21-7
VECTOR LOAD ELEMENT (64)	VLEG	VRX			VF		α[7,9]	A	SP	Dv										B_2	E702	21-7
VECTOR LOAD ELEMENT (8)	VLEB	VRX			VF		α[7,9]	A	SP	Dv										B_2	E700	21-6
VECTOR LOAD ELEMENT IMMEDIATE (16)	VLEIH	VRI-a			VF		α[7,9]		SP	Dv											E741	21-7
VECTOR LOAD ELEMENT IMMEDIATE (32)	VLEIF	VRI-a			VF		α[7,9]		SP	Dv											E743	21-7
VECTOR LOAD ELEMENT IMMEDIATE (64)	VLEIG	VRI-a			VF		α[7,9]		SP	Dv											E742	21-7
VECTOR LOAD ELEMENT IMMEDIATE (8)	VLEIB	VRI-a			VF		α[7,9]		SP	Dv											E740	21-7
VECTOR LOAD FP INTEGER	VFI	VRR-a			VF		α[7,9]		SP	Dv	Xi					Xx					E7C7	24-23
VECTOR LOAD GR FROM VR ELEMENT	VLGV	VRS-c			VF		α[7,9]		SP	Dv											E721	21-7
VECTOR LOAD IMMEDIATE DECIMAL	VLIP	VRI-h			VD		α[7,9]			Dv	Dg										E649	25-9
VECTOR LOAD LOGICAL ELEMENT AND ZERO	VLLEZ	VRX			VF		α[7,9]	A	SP	Dv										B_2	E704	21-8
VECTOR LOAD MULTIPLE	VLM	VRS-a			VF		α[7,9]	A	SP	Dv										B_2	E736	21-9
VECTOR LOAD POSITIVE	VLP	VRR-a			VF		α[7,9]		SP	Dv											E7DF	22-12
VECTOR LOAD RIGHTMOST WITH LENGTH	VLRL	VSI			VD		α[7,9]	A	SP	Dv										B_2	E635	21-9
VECTOR LOAD RIGHTMOST WITH LENGTH	VLRLR	VRS-d			VD		α[7,9]	A		Dv										B_2	E637	21-9
VECTOR LOAD TO BLOCK BOUNDARY	VLBB	VRX			VF		α[7,9]	A	SP	Dv										B_2	E707	21-10
VECTOR LOAD VR ELEMENT FROM GR	VLVG	VRS-b			VF		α[7,9]		SP	Dv											E722	21-11
VECTOR LOAD VR FROM GRS DISJOINT	VLVGP	VRR-f			VF		α[7,9]			Dv											E762	21-11
VECTOR LOAD WITH LENGTH	VLL	VRS-b			VF		α[7,9]	A		Dv										B_2	E737	21-11
VECTOR MAXIMUM	VMX	VRR-c			VF		α[7,9]		SP	Dv											E7FF	22-13
VECTOR MAXIMUM LOGICAL	VMXL	VRR-c			VF		α[7,9]		SP	Dv											E7FD	22-13
VECTOR MERGE HIGH	VMRH	VRR-c			VF		α[7,9]		SP	Dv											E761	21-12

Figure B-1. Instructions Arranged by Name (Part 21 of 23)

Name	Mnemonic	Characteristics									Opcode	Page
VECTOR MERGE LOW	VMRL	VRR-c	VF	$\sigma^{7,9}$		SP	Dv				E760	21-12
VECTOR MINIMUM	VMN	VRR-c	VF	$\sigma^{7,9}$		SP	Dv				E7FE	22-13
VECTOR MINIMUM LOGICAL	VMNL	VRR-c	VF	$\sigma^{7,9}$		SP	Dv				E7FC	22-14
VECTOR MULTIPLY AND ADD EVEN	VMAE	VRR-d	VF	$\sigma^{7,9}$		SP	Dv				E7AE	22-15
VECTOR MULTIPLY AND ADD HIGH	VMAH	VRR-d	VF	$\sigma^{7,9}$		SP	Dv				E7AB	22-15
VECTOR MULTIPLY AND ADD LOGICAL EVEN	VMALE	VRR-d	VF	$\sigma^{7,9}$		SP	Dv				E7AC	22-15
VECTOR MULTIPLY AND ADD LOGICAL HIGH	VMALH	VRR-d	VF	$\sigma^{7,9}$		SP	Dv				E7A9	22-15
VECTOR MULTIPLY AND ADD LOGICAL ODD	VMALO	VRR-d	VF	$\sigma^{7,9}$		SP	Dv				E7AD	22-16
VECTOR MULTIPLY AND ADD LOW	VMAL	VRR-d	VF	$\sigma^{7,9}$		SP	Dv				E7AA	22-14
VECTOR MULTIPLY AND ADD ODD	VMAO	VRR-d	VF	$\sigma^{7,9}$		SP	Dv				E7AF	22-16
VECTOR MULTIPLY AND SHIFT DECIMAL	VMSP	VRI-f	C* VD	$\sigma^{7,9}$		SP	Dv	Dg	DF		E679	25-11
VECTOR MULTIPLY DECIMAL	VMP	VRI-f	C* VD	$\sigma^{7,9}$		SP	Dv	Dg	DF		E678	25-10
VECTOR MULTIPLY EVEN	VME	VRR-c	VF	$\sigma^{7,9}$		SP	Dv				E7A6	22-18
VECTOR MULTIPLY HIGH	VMH	VRR-c	VF	$\sigma^{7,9}$		SP	Dv				E7A3	22-16
VECTOR MULTIPLY LOGICAL EVEN	VMLE	VRR-c	VF	$\sigma^{7,9}$		SP	Dv				E7A4	22-18
VECTOR MULTIPLY LOGICAL HIGH	VMLH	VRR-c	VF	$\sigma^{7,9}$		SP	Dv				E7A1	22-17
VECTOR MULTIPLY LOGICAL ODD	VMLO	VRR-c	VF	$\sigma^{7,9}$		SP	Dv				E7A5	22-18
VECTOR MULTIPLY LOW	VML	VRR-c	VF	$\sigma^{7,9}$		SP	Dv				E7A2	22-17
VECTOR MULTIPLY ODD	VMO	VRR-c	VF	$\sigma^{7,9}$		SP	Dv				E7A7	22-18
VECTOR MULTIPLY SUM LOGICAL	VMSL	VRR-d	V1	$\sigma^{7,9}$		SP	Dv				E7B8	22-19
VECTOR NAND	VNN	VRR-c	V1	$\sigma^{7,9}$			DV				E76E	22-20
VECTOR NOR	VNO	VRR-c	VF	$\sigma^{7,9}$			Dv				E76B	22-20
VECTOR NOT EXCLUSIVE OR	VNX	VRR-c	V1	$\sigma^{7,9}$			Dv				E76C	22-20
VECTOR OR	VO	VRR-c	VF	$\sigma^{7,9}$			Dv				E76A	22-20
VECTOR OR WITH COMPLEMENT	VOC	VRR-c	V1	$\sigma^{7,9}$			Dv				E76F	22-21
VECTOR PACK	VPK	VRR-c	VF	$\sigma^{7,9}$		SP	Dv				E794	21-13
VECTOR PACK LOGICAL SATURATE	VPKLS	VRR-b	C* VF	$\sigma^{7,9}$		SP	Dv				E795	21-14
VECTOR PACK SATURATE	VPKS	VRR-b	C* VF	$\sigma^{7,9}$		SP	Dv				E797	21-13
VECTOR PACK ZONED	VPKZ	VSI	VD	$\sigma^{7,9}$	A	SP	Dv			B_2	E634	25-12
VECTOR PERFORM SIGN OPERATION DECIMAL	VPSOP	VRI-g	C* VD	$\sigma^{7,9}$		SP	Dv	Dg	DF		E65B	25-13
VECTOR PERMUTE	VPERM	VRR-e	VF	$\sigma^{7,9}$			Dv				E78C	21-15
VECTOR PERMUTE DOUBLEWORD IMMEDIATE	VPDI	VRR-c	VF	$\sigma^{7,9}$			Dv				E784	21-15
VECTOR POPULATION COUNT	VPOPCT	VRR-a	VF	$\sigma^{7,9}$		SP	Dv				E750	22-21
VECTOR REMAINDER DECIMAL	VRP	VRI-f	C* VD	$\sigma^{7,9}$		SP	Dv	Dg	DF	DK	E67B	25-14
VECTOR REPLICATE	VREP	VRI-c	VF	$\sigma^{7,9}$		SP	Dv				E74D	21-15
VECTOR REPLICATE IMMEDIATE	VREPI	VRI-a	VF	$\sigma^{7,9}$		SP	Dv				E745	21-16
VECTOR SCATTER ELEMENT (32)	VSCEF	VRV	VF	$\sigma^{7,9}$	A	SP	Dv		ST	B_2	E71B	21-16
VECTOR SCATTER ELEMENT (64)	VSCEG	VRV	VF	$\sigma^{7,9}$	A	SP	Dv		ST	B_2	E71A	21-16
VECTOR SELECT	VSEL	VRR-e	VF	$\sigma^{7,9}$			Dv				E78D	21-17
VECTOR SHIFT AND DIVIDE DECIMAL	VSDP	VRI-f	C* VD	$\sigma^{7,9}$		SP	Dv	Dg	DF	DK	E67E	25-16
VECTOR SHIFT AND ROUND DECIMAL	VSRP	VRI-g	C* VD	$\sigma^{7,9}$		SP	Dv	Dg	DF		E659	25-17
VECTOR SHIFT LEFT	VSL	VRR-c	VF	$\sigma^{7,9}$			Dv				E774	22-25
VECTOR SHIFT LEFT BY BYTE	VSLB	VRR-c	VF	$\sigma^{7,9}$			Dv				E775	22-25
VECTOR SHIFT LEFT DOUBLE BY BYTE	VSLDB	VRI-d	VF	$\sigma^{7,9}$			Dv				E777	22-25
VECTOR SHIFT RIGHT ARITHMETIC	VSRA	VRR-c	VF	$\sigma^{7,9}$			Dv				E77E	22-25
VECTOR SHIFT RIGHT ARITHMETIC BY BYTE	VSRAB	VRR-c	VF	$\sigma^{7,9}$			Dv				E77F	22-26
VECTOR SHIFT RIGHT LOGICAL	VSRL	VRR-c	VF	$\sigma^{7,9}$			Dv				E77C	22-26
VECTOR SHIFT RIGHT LOGICAL BY BYTE	VSRLB	VRR-c	VF	$\sigma^{7,9}$			Dv				E77D	22-26
VECTOR SIGN EXTEND TO DOUBLEWORD	VSEG	VRR-a	VF	$\sigma^{7,9}$		SP	Dv				E75F	21-17
VECTOR STORE	VST	VRX	VF	$\sigma^{7,9}$	A		Dv		ST	B_2	E70E	21-18
VECTOR STORE ELEMENT (16)	VSTEH	VRX	VF	$\sigma^{7,9}$	A	SP	Dv		ST	B_2	E709	21-18
VECTOR STORE ELEMENT (32)	VSTEF	VRX	VF	$\sigma^{7,9}$	A	SP	Dv		ST	B_2	E70B	21-18
VECTOR STORE ELEMENT (64)	VSTEG	VRX	VF	$\sigma^{7,9}$	A	SP	Dv		ST	B_2	E70A	21-18
VECTOR STORE ELEMENT (8)	VSTEB	VRX	VF	$\sigma^{7,9}$	A	SP	Dv		ST	B_2	E708	21-18
VECTOR STORE MULTIPLE	VSTM	VRS-a	VF	$\sigma^{7,9}$	A	SP	Dv		ST	B_2	E73E	21-19

Figure B-1. Instructions Arranged by Name (Part 22 of 23)

Name	Mne-monic	Characteristics										Op-code	Page
VECTOR STORE RIGHTMOST WITH LENGTH	VSTRL	VSI		VD	$\alpha^{7,9}$	A	SP	Dv		ST	B_2	E63D	21-19
VECTOR STORE RIGHTMOST WITH LENGTH	VSTRLR	VRS-d		VD	$\alpha^{7,9}$	A		Dv		ST	B_2	E63F	21-19
VECTOR STORE WITH LENGTH	VSTL	VRS-b		VF	$\alpha^{7,9}$	A		Dv		ST	B_2	E73F	21-20
VECTOR STRING RANGE COMPARE	VSTRC	VRR-d	C*	VF	$\alpha^{7,9}$		SP	Dv				E78A	23-6
VECTOR SUBTRACT	VS	VRR-c		VF	$\alpha^{7,9}$		SP	Dv				E7F7	22-26
VECTOR SUBTRACT COMPUTE BORROW INDICATION	VSCBI	VRR-c		VF	$\alpha^{7,9}$		SP	Dv				E7F5	22-27
VECTOR SUBTRACT DECIMAL	VSP	VRI-f	C*	VD	$\alpha^{7,9}$		SP	Dv	Dg DF			E673	25-18
VECTOR SUBTRACT WITH BORROW COMPUTE BORROW INDICATION	VSBCBI	VRR-d		VF	$\alpha^{7,9}$		SP	Dv				E7BD	22-28
VECTOR SUBTRACT WITH BORROW INDICATION	VSBI	VRR-d		VF	$\alpha^{7,9}$		SP	Dv				E7BF	22-27
VECTOR SUM ACROSS DOUBLEWORD	VSUMG	VRR-c		VF	$\alpha^{7,9}$		SP	Dv				E765	22-28
VECTOR SUM ACROSS QUADWORD	VSUMQ	VRR-c		VF	$\alpha^{7,9}$		SP	Dv				E767	22-29
VECTOR SUM ACROSS WORD	VSUM	VRR-c		VF	$\alpha^{7,9}$		SP	Dv				E764	22-29
VECTOR TEST DECIMAL	VTP	VRR-g	C	VD	$\alpha^{7,9}$			Dv				E65F	25-20
VECTOR TEST UNDER MASK	VTM	VRR-a	C	VF	$\alpha^{7,9}$			Dv				E7D8	22-30
VECTOR UNPACK HIGH	VUPH	VRR-a		VF	$\alpha^{7,9}$		SP	Dv				E7D7	21-20
VECTOR UNPACK LOGICAL HIGH	VUPLH	VRR-a		VF	$\alpha^{7,9}$		SP	Dv				E7D5	21-21
VECTOR UNPACK LOGICAL LOW	VUPLL	VRR-a		VF	$\alpha^{7,9}$		SP	Dv				E7D4	21-21
VECTOR UNPACK LOW	VUPL	VRR-a		VF	$\alpha^{7,9}$		SP	Dv				E7D6	21-22
VECTOR UNPACK ZONED	VUPKZ	VSI		VD	$\alpha^{7,9}$	A	SP	Dv		ST	B_2	E63C	25-20
ZERO AND ADD	ZAP	SS-b	C		α^{9}	A		Dg	DF	ST	B_1 B_2	F8	8-14

Figure B-1. Instructions Arranged by Name (Part 23 of 23)

Instructions Arranged by Mnemonic

Mnemonic	Name	Characteristics																	Opcode	Page
—	DIAGNOSE		DM		P	DM												MD	83	10-23
A	ADD (32)	RX-a	C			A		IF										B_2	5A	7-26
AD	ADD NORMALIZED (long HFP)	RX-a	C		α[7,9]	A	Da	EU	EO		LS							B_2	6A	18-8
ADB	ADD (long BFP)	RXE	C		α[7,9]	A	Db	Xi		Xo	Xu	Xx						B_2	ED1A	19-15
ADBR	ADD (long BFP)	RRE	C		α[7,9]		Db	Xi		Xo	Xu	Xx							B31A	19-15
ADR	ADD NORMALIZED (long HFP)	RR	C		α[7,9]		Da	EU	EO		LS								2A	18-8
ADTR	ADD (long DFP)	RRF-a	C	TF	α[7,9]		Dt	Xi		Xo	Xu	Xx							B3D2	20-19
ADTRA	ADD (long DFP)	RRF-a	C	F	α[7,9]		Dt	Xi		Xo	Xu	Xx	Xq						B3D2	20-19
AE	ADD NORMALIZED (short HFP)	RX-a	C		α[7,9]	A	Da	EU	EO		LS							B_2	7A	18-8
AEB	ADD (short BFP)	RXE	C		α[7,9]	A	Db	Xi		Xo	Xu	Xx						B_2	ED0A	19-15
AEBR	ADD (short BFP)	RRE	C		α[7,9]		Db	Xi		Xo	Xu	Xx							B30A	19-15
AER	ADD NORMALIZED (short HFP)	RR	C		α[7,9]		Da	EU	EO		LS								3A	18-8
AFI	ADD IMMEDIATE (32)	RIL-a	C	EI				IF											C29	7-27
AG	ADD (64)	RXY-a	C	N		A		IF										B_2	E308	7-27
AGF	ADD (64←32)	RXY-a	C	N		A		IF										B_2	E318	7-27
AGFI	ADD IMMEDIATE (64←32)	RIL-a	C	EI				IF											C28	7-27
AGFR	ADD (64←32)	RRE	C	N				IF											B918	7-26
AGH	ADD HALFWORD (64←16)	RXY-a	C	MI2		A		IF										B_2	E338	7-28
AGHI	ADD HALFWORD IMMEDIATE (64←16)	RI-a	C	N				IF											A7B	7-29
AGHIK	ADD IMMEDIATE (64←16)	RIE-d	C	DO				IF											ECD9	7-27
AGR	ADD (64)	RRE	C	N				IF											B908	7-26
AGRK	ADD (64)	RRF-a	C	DO				IF											B9E8	7-26
AGSI	ADD IMMEDIATE (64←8)	SIY	C	GE		A		IF	£[1]							ST	B_1		EB7A	7-27
AH	ADD HALFWORD (32←16)	RX-a	C			A		IF										B_2	4A	7-28
AHHHR	ADD HIGH (32)	RRF-a	C	HW				IF											B9C8	7-29
AHHLR	ADD HIGH (32)	RRF-a	C	HW				IF											B9D8	7-29
AHI	ADD HALFWORD IMMEDIATE (32←16)	RI-a	C					IF											A7A	7-28
AHIK	ADD IMMEDIATE (32←16)	RIE-d	C	DO				IF											ECD8	7-27
AHY	ADD HALFWORD (32←16)	RXY-a	C	LD		A		IF										B_2	E37A	7-28
AIH	ADD IMMEDIATE HIGH (32)	RIL-a	C	HW				IF											CC8	7-29
AL	ADD LOGICAL (32)	RX-a	C			A												B_2	5E	7-30
ALC	ADD LOGICAL WITH CARRY (32)	RXY-a	C	N3		A												B_2	E398	7-31
ALCG	ADD LOGICAL WITH CARRY (64)	RXY-a	C	N		A												B_2	E388	7-31
ALCGR	ADD LOGICAL WITH CARRY (64)	RRE	C	N															B988	7-31
ALCR	ADD LOGICAL WITH CARRY (32)	RRE	C	N3															B998	7-31
ALFI	ADD LOGICAL IMMEDIATE (32)	RIL-a	C	EI															C2B	7-30
ALG	ADD LOGICAL (64)	RXY-a	C	N		A												B_2	E30A	7-30
ALGF	ADD LOGICAL (64←32)	RXY-a	C	N		A												B_2	E31A	7-30
ALGFI	ADD LOGICAL IMMEDIATE (64←32)	RIL-a	C	EI															C2A	7-30
ALGFR	ADD LOGICAL (64←32)	RRE	C	N															B91A	7-30
ALGHSIK	ADD LOGICAL WITH SIGNED IMMEDIATE (64←16)	RIE-d	C	DO															ECDB	7-32
ALGR	ADD LOGICAL (64)	RRE	C	N															B90A	7-30
ALGRK	ADD LOGICAL (64)	RRF-a	C	DO															B9EA	7-30
ALGSI	ADD LOGICAL WITH SIGNED IMMEDIATE (64←8)	SIY	C	GE		A			£[1]							ST	B_1		EB7E	7-32
ALHHHR	ADD LOGICAL HIGH (32)	RRF-a	C	HW															B9CA	7-31
ALHHLR	ADD LOGICAL HIGH (32)	RRF-a	C	HW															B9DA	7-31
ALHSIK	ADD LOGICAL WITH SIGNED IMMEDIATE (32←16)	RIE-d	C	DO															ECDA	7-32
ALR	ADD LOGICAL (32)	RR	C																1E	7-30
ALRK	ADD LOGICAL (32)	RRF-a	C	DO															B9FA	7-30
ALSI	ADD LOGICAL WITH SIGNED IMMEDIATE (32←8)	SIY	C	GE		A			£[1]							ST	B_1		EB6E	7-32

Figure B-2. Instructions Arranged by Mnemonic (Part 1 of 23)

Mnemonic	Name	Characteristics																						Opcode	Page
ALSIH	ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (32)	RIL-a	C	HW																				CCA	7-33
ALSIHN	ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (32)	RIL-a		HW																				CCB	7-33
ALY	ADD LOGICAL (32)	RXY-a	C	LD		A																	B_2	E35E	7-30
AP	ADD DECIMAL	SS-b	C		¤[9]	A			Dg	DF										ST	B_1	B_2		FA	8-6
AR	ADD (32)	RR	C							IF														1A	7-26
ARK	ADD (32)	RRF-a	C	DO						IF														B9F8	7-26
ASI	ADD IMMEDIATE (32←8)	SIY	C	GE		A				IF	£[1]									ST	B_1			EB6A	7-27
AU	ADD UNNORMALIZED (short HFP)	RX-a	C		¤[7,9]	A			Da		EO		LS									B_2		7E	18-9
AUR	ADD UNNORMALIZED (short HFP)	RR	C		¤[7,9]				Da		EO		LS											3E	18-9
AW	ADD UNNORMALIZED (long HFP)	RX-a	C		¤[7,9]	A			Da		EO		LS									B_2		6E	18-9
AWR	ADD UNNORMALIZED (long HFP)	RR	C		¤[7,9]				Da		EO		LS											2E	18-9
AXBR	ADD (extended BFP)	RRE	C		¤[7,9]		SP		Db	Xi		Xo	Xu	Xx										B34A	19-15
AXR	ADD NORMALIZED (extended HFP)	RR	C		¤[7,9]		SP		Da	EU	EO		LS											36	18-8
AXTR	ADD (extended DFP)	RRF-a	C	TF	¤[7,9]		SP		Dt	Xi		Xo	Xu	Xx										B3DA	20-19
AXTRA	ADD (extended DFP)	RRF-a	C	F	¤[7,9]		SP		Dt	Xi		Xo	Xu	Xx	Xq									B3DA	20-19
AY	ADD (32)	RXY-a	C	LD		A				IF												B_2		E35A	7-27
BAKR	BRANCH AND STACK	RRE			¤[1]	A[1]*			Z[5]	T										B	ST			B240	10-11
BAL	BRANCH AND LINK	RX-a			¤[9]															B				45	7-35
BALR	BRANCH AND LINK	RR			¤[2,9]					T										B				05	7-35
BAS	BRANCH AND SAVE	RX-a			¤[9]															B				4D	7-36
BASR	BRANCH AND SAVE	RR			¤[2,9]					T										B				0D	7-36
BASSM	BRANCH AND SAVE AND SET MODE	RR			¤[2,3,9]					T										B				0C	7-37
BC	BRANCH ON CONDITION	RX-b			¤[9]															B				47	7-40
BCR	BRANCH ON CONDITION	RR			¤[9]						¢[1]									B				07	7-40
BCT	BRANCH ON COUNT (32)	RX-a			¤[9]															B				46	7-41
BCTG	BRANCH ON COUNT (64)	RXY-a		N	¤[9]															B				E346	7-41
BCTGR	BRANCH ON COUNT (64)	RRE		N	¤[9]															B				B946	7-41
BCTR	BRANCH ON COUNT (32)	RR			¤[9]															B				06	7-41
BIC	BRANCH INDIRECT ON CONDITION	RXY-b		MI2	¤[9]	A														B		B_2		E347	7-39
BPP	BRANCH PREDICTION PRELOAD	SMI		EH	¤[9]																			C7	7-43
BPRP	BRANCH PREDICTION RELATIVE PRELOAD	MII		EH	¤[9]																			C5	7-43
BRAS	BRANCH RELATIVE AND SAVE	RI-b			¤[9]															B				A75	7-46
BRASL	BRANCH RELATIVE AND SAVE LONG	RIL-b		N3	¤[9]															B				C05	7-46
BRC	BRANCH RELATIVE ON CONDITION	RI-c			¤[10]															B				A74	7-46
BRCL	BRANCH RELATIVE ON CONDITION LONG	RIL-c		N3	¤[10]															B				C04	7-46
BRCT	BRANCH RELATIVE ON COUNT (32)	RI-b			¤[9]															B				A76	7-47
BRCTG	BRANCH RELATIVE ON COUNT (64)	RI-b		N	¤[9]															B				A77	7-47
BRCTH	BRANCH RELATIVE ON COUNT HIGH (32)	RIL-b		HW	¤[9]															B				CC6	7-47
BRXH	BRANCH RELATIVE ON INDEX HIGH (32)	RSI			¤[9]															B				84	7-48
BRXHG	BRANCH RELATIVE ON INDEX HIGH (64)	RIE-e		N	¤[9]															B				EC44	7-48
BRXLE	BRANCH RELATIVE ON INDEX LOW OR EQ. (32)	RSI			¤[9]															B				85	7-48
BRXLG	BRANCH RELATIVE ON INDEX LOW OR EQ. (64)	RIE-e		N	¤[9]															B				EC45	7-48
BSA	BRANCH AND SET AUTHORITY	RRE			Q	A[1]*			SO	T										B				B25A	10-7
BSG	BRANCH IN SUBSPACE GROUP	RRE			¤[1]	A[1]*			SO	T										B		R_2		B258	10-13
BSM	BRANCH AND SET MODE	RR			¤[3,9]					T										B				0B	7-38
BXH	BRANCH ON INDEX HIGH (32)	RS-a			¤[9]															B				86	7-42
BXHG	BRANCH ON INDEX HIGH (64)	RSY-a		N	¤[9]															B				EB44	7-42
BXLE	BRANCH ON INDEX LOW OR EQUAL (32)	RS-a			¤[9]															B				87	7-42
BXLEG	BRANCH ON INDEX LOW OR EQUAL (64)	RSY-a		N	¤[9]															B				EB45	7-42
C	COMPARE (32)	RX-a	C			A																B_2		59	7-134
CD	COMPARE (long HFP)	RX-a	C		¤[7,9]	A			Da													B_2		69	18-10
CDB	COMPARE (long BFP)	RXE	C		¤[7,9]	A			Db	Xi												B_2		ED19	19-17
CDBR	COMPARE (long BFP)	RRE	C		¤[7,9]				Db	Xi														B319	19-17

Figure B-2. Instructions Arranged by Mnemonic (Part 2 of 23)

Mnemonic	Name	Characteristics												Opcode	Page
CDFBR	CONVERT FROM FIXED (32 to long BFP)	RRE			¤[7,9]			Db						B395	19-19
CDFBRA	CONVERT FROM FIXED (32 to long BFP)	RRF-e		F	¤[7,9]		SP	Db						B395	19-19
CDFR	CONVERT FROM FIXED (32 to long HFP)	RRE			¤[7,9]			Da						B3B5	18-11
CDFTR	CONVERT FROM FIXED (32 to long DFP)	RRE		F	¤[7,9]			Dt						B951	20-24
CDGBR	CONVERT FROM FIXED (64 to long BFP)	RRE		N	¤[7,9]			Db			Xx			B3A5	19-19
CDGBRA	CONVERT FROM FIXED (64 to long BFP)	RRF-e		F	¤[7,9]		SP	Db			Xx			B3A5	19-19
CDGR	CONVERT FROM FIXED (64 to long HFP)	RRE		N	¤[7,9]			Da						B3C5	18-11
CDGTR	CONVERT FROM FIXED (64 to long DFP)	RRE		TF	¤[7,9]			Dt			Xx			B3F1	20-24
CDGTRA	CONVERT FROM FIXED (64 to long DFP)	RRF-e		F	¤[7,9]			Dt			Xx	Xq		B3F1	20-24
CDLFBR	CONVERT FROM LOGICAL (32 to long BFP)	RRF-e		F	¤[7,9]		SP	Db						B391	19-21
CDLFTR	CONVERT FROM LOGICAL (32 to long DFP)	RRF-e		F	¤[7,9]			Dt						B953	20-25
CDLGBR	CONVERT FROM LOGICAL (64 to long BFP)	RRF-e		F	¤[7,9]		SP	Db			Xx			B3A1	19-21
CDLGTR	CONVERT FROM LOGICAL (64 to long DFP)	RRF-e		F	¤[7,9]			Dt			Xx	Xq		B952	20-25
CDPT	CONVERT FROM PACKED (to long DFP)	RSL-b		PC	¤[7,9]	A	SP	Dt	Dg				B_2	EDAE	20-26
CDR	COMPARE (long HFP)	RR	C		¤[7,9]			Da						29	18-10
CDS	COMPARE DOUBLE AND SWAP (32)	RS-a	C		¤[9]	A	SP		$			ST	B_2	BB	7-144
CDSG	COMPARE DOUBLE AND SWAP (64)	RSY-a	C	N	¤[9]	A	SP		$			ST	B_2	EB3E	7-144
CDSTR	CONVERT FROM SIGNED PACKED (64 to long DFP)	RRE		TF	¤[7,9]			Dt	Dg					B3F3	20-28
CDSY	COMPARE DOUBLE AND SWAP (32)	RSY-a	C	LD	¤[9]	A	SP		$			ST	B_2	EB31	7-144
CDTR	COMPARE (long DFP)	RRE	C	TF	¤[7,9]			Dt	Xi					B3E4	20-22
CDUTR	CONVERT FROM UNSIGNED PACKED (64 to long DFP)	RRE		TF	¤[7,9]			Dt	Dg					B3F2	20-28
CDZT	CONVERT FROM ZONED (to long DFP)	RSL-b		ZF	¤[7,9]	A	SP	Dt	Dg				B_2	EDAA	20-29
CE	COMPARE (short HFP)	RX-a	C		¤[7,9]	A		Da					B_2	79	18-10
CEB	COMPARE (short BFP)	RXE	C		¤[7,9]	A		Db	Xi				B_2	ED09	19-17
CEBR	COMPARE (short BFP)	RRE	C		¤[7,9]			Db	Xi					B309	19-17
CEDTR	COMPARE BIASED EXPONENT (long DFP)	RRE	C	TF	¤[7,9]			Dt						B3F4	20-23
CEFBR	CONVERT FROM FIXED (32 to short BFP)	RRE			¤[7,9]			Db			Xx			B394	19-19
CEFBRA	CONVERT FROM FIXED (32 to short BFP)	RRF-e		F	¤[7,9]		SP	Db			Xx			B394	19-19
CEFR	CONVERT FROM FIXED (32 to short HFP)	RRE			¤[7,9]			Da						B3B4	18-11
CEGBR	CONVERT FROM FIXED (64 to short BFP)	RRE		N	¤[7,9]			Db			Xx			B3A4	19-19
CEGBRA	CONVERT FROM FIXED (64 to short BFP)	RRF-e		F	¤[7,9]		SP	Db			Xx			B3A4	19-19
CEGR	CONVERT FROM FIXED (64 to short HFP)	RRE		N	¤[7,9]			Da						B3C4	18-11
CELFBR	CONVERT FROM LOGICAL (32 to short BFP)	RRF-e		F	¤[7,9]		SP	Db			Xx			B390	19-21
CELGBR	CONVERT FROM LOGICAL (64 to short BFP)	RRF-e		F	¤[7,9]		SP	Db			Xx			B3A0	19-21
CER	COMPARE (short HFP)	RR	C		¤[7,9]			Da						39	18-10
CEXTR	COMPARE BIASED EXPONENT (extended DFP)	RRE	C	TF	¤[7,9]		SP	Dt						B3FC	20-23
CFC	COMPARE AND FORM CODEWORD	S	C		¤[9]	A	SP	II		GM	I1			B21A	7-137
CFDBR	CONVERT TO FIXED (long BFP to 32)	RRF-e	C		¤[7,9]		SP	Db	Xi		Xx			B399	19-22
CFDBRA	CONVERT TO FIXED (long BFP to 32)	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx			B399	19-22
CFDR	CONVERT TO FIXED (long HFP to 32)	RRF-e	C		¤[7,9]		SP	Da						B3B9	18-11
CFDTR	CONVERT TO FIXED (long DFP to 32)	RRF-e	C	F	¤[7,9]			Dt	Xi		Xx			B941	20-30
CFEBR	CONVERT TO FIXED (short BFP to 32)	RRF-e	C		¤[7,9]		SP	Db	Xi		Xx			B398	19-22
CFEBRA	CONVERT TO FIXED (short BFP to 32)	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx			B398	19-22
CFER	CONVERT TO FIXED (short HFP to 32)	RRF-e	C		¤[7,9]		SP	Da						B3B8	18-11
CFI	COMPARE IMMEDIATE (32)	RIL-a	C	EI										C2D	7-134
CFXBR	CONVERT TO FIXED (extended BFP to 32)	RRF-e	C		¤[7,9]		SP	Db	Xi		Xx			B39A	19-22
CFXBRA	CONVERT TO FIXED (extended BFP to 32)	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx			B39A	19-22
CFXR	CONVERT TO FIXED (extended HFP to 32)	RRF-e	C		¤[7,9]		SP	Da						B3BA	18-11
CFXTR	CONVERT TO FIXED (extended DFP to 32)	RRF-e	C	F	¤[7,9]		SP	Dt	Xi		Xx			B949	20-30
CG	COMPARE (64)	RXY-a	C	N		A							B_2	E320	7-134
CGDBR	CONVERT TO FIXED (long BFP to 64)	RRF-e	C	N	¤[7,9]		SP	Db	Xi		Xx			B3A9	19-22
CGDBRA	CONVERT TO FIXED (long BFP to 64)	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx			B3A9	19-22
CGDR	CONVERT TO FIXED (long HFP to 64)	RRF-e	C	N	¤[7,9]		SP	Da						B3C9	18-11

Figure B-2. Instructions Arranged by Mnemonic (Part 3 of 23)

Mne-monic	Name	Characteristics												Op-code	Page
CGDTR	CONVERT TO FIXED (long DFP to 64)	RRF-e	C	TF	¤[7,9]			Dt	Xi		Xx			B3E1	20-29
CGDTRA	CONVERT TO FIXED (long DFP to 64)	RRF-e	C	F	¤[7,9]			Dt	Xi		Xx			B3E1	20-30
CGEBR	CONVERT TO FIXED (short BFP to 64)	RRF-e	C	N	¤[7,9]		SP	Db	Xi		Xx			B3A8	19-22
CGEBRA	CONVERT TO FIXED (short BFP to 64)	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx			B3A8	19-22
CGER	CONVERT TO FIXED (short HFP to 64)	RRF-e	C	N	¤[7,9]		SP	Da						B3C8	18-11
CGF	COMPARE (64←32)	RXY-a	C	N		A							B_2	E330	7-134
CGFI	COMPARE IMMEDIATE (64←32)	RIL-a	C	EI										C2C	7-135
CGFR	COMPARE (64←32)	RRE	C	N										B930	7-134
CGFRL	COMPARE RELATIVE LONG (64←32)	RIL-b	C	GE		A*	SP							C6C	7-135
CGH	COMPARE HALFWORD (64←16)	RXY-a	C	GE		A							B_2	E334	7-150
CGHI	COMPARE HALFWORD IMMEDIATE (64←16)	RI-a	C	N										A7F	7-150
CGHRL	COMPARE HALFWORD RELATIVE LONG (64←16)	RIL-b	C	GE		A*								C64	7-150
CGHSI	COMPARE HALFWORD IMMEDIATE (64←16)	SIL	C	GE		A							B_1	E558	7-150
CGIB	COMPARE IMMEDIATE AND BRANCH (64←8)	RIS		GE	¤[9]							B		ECFC	7-136
CGIJ	COMPARE IMMEDIATE AND BRANCH RELATIVE (64←8)	RIE-c		GE	¤[10]							B		EC7C	7-136
CGIT	COMPARE IMMEDIATE AND TRAP (64←16)	RIE-a		GE					Dc					EC70	7-149
CGR	COMPARE (64)	RRE	C	N										B920	7-134
CGRB	COMPARE AND BRANCH (64)	RRS		GE	¤[9]							B		ECE4	7-135
CGRJ	COMPARE AND BRANCH RELATIVE (64)	RIE-b		GE	¤[10]							B		EC64	7-136
CGRL	COMPARE RELATIVE LONG (64)	RIL-b	C	GE		A*	SP							C68	7-135
CGRT	COMPARE AND TRAP (64)	RRF-c		GE					Dc					B960	7-149
CGXBR	CONVERT TO FIXED (extended BFP to 64)	RRF-e	C	N	¤[7,9]		SP	Db	Xi		Xx			B3AA	19-22
CGXBRA	CONVERT TO FIXED (extended BFP to 64)	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx			B3AA	19-22
CGXR	CONVERT TO FIXED (extended HFP to 64)	RRF-e	C	N	¤[7,9]		SP	Da						B3CA	18-11
CGXTR	CONVERT TO FIXED (extended DFP to 64)	RRF-e	C	TF	¤[7,9]		SP	Dt	Xi		Xx			B3E9	20-29
CGXTRA	CONVERT TO FIXED (extended DFP to 64)	RRF-e	C	F	¤[7,9]		SP	Dt	Xi		Xx			B3E9	20-30
CH	COMPARE HALFWORD (32←16)	RX-a	C			A							B_2	49	7-150
CHF	COMPARE HIGH (32)	RXY-a	C	HW		A							B_2	E3CD	7-151
CHHR	COMPARE HIGH (32)	RRE	C	HW										B9CD	7-151
CHHSI	COMPARE HALFWORD IMMEDIATE (16←16)	SIL	C	GE		A							B_1	E554	7-150
CHI	COMPARE HALFWORD IMMEDIATE (32←16)	RI-a	C											A7E	7-150
CHLR	COMPARE HIGH (32)	RRE	C	HW										B9DD	7-151
CHRL	COMPARE HALFWORD RELATIVE LONG (32←16)	RIL-b	C	GE		A*								C65	7-150
CHSI	COMPARE HALFWORD IMMEDIATE (32←16)	SIL	C	GE		A							B_1	E55C	7-150
CHY	COMPARE HALFWORD (32←16)	RXY-a	C	LD		A							B_2	E379	7-150
CIB	COMPARE IMMEDIATE AND BRANCH (32←8)	RIS		GE	¤[9]							B		ECFE	7-136
CIH	COMPARE IMMEDIATE HIGH (32)	RIL-a	C	HW										CCD	7-151
CIJ	COMPARE IMMEDIATE AND BRANCH RELATIVE (32←8)	RIE-c		GE	¤[10]							B		EC7E	7-136
CIT	COMPARE IMMEDIATE AND TRAP (32←16)	RIE-a		GE					Dc					EC72	7-149
CKSM	CHECKSUM	RRE	C		¤[9]	A	SP	IC					R_2	B241	7-49
CL	COMPARE LOGICAL (32)	RX-a	C			A							B_2	55	7-152
CLC	COMPARE LOGICAL (character)	SS-a	C		¤[9]	A							B_1 B_2	D5	7-152
CLCL	COMPARE LOGICAL LONG	RR	C		¤[9]	A	SP	II					R_1 R_2	0F	7-158
CLCLE	COMPARE LOGICAL LONG EXTENDED	RS-a	C		¤[9]	A	SP	IC					R_1 R_3	A9	7-160
CLCLU	COMPARE LOGICAL LONG UNICODE	RSY-a	C	E2	¤[9]	A	SP	IC					R_1 R_2	EB8F	7-163
CLFDBR	CONVERT TO LOGICAL (long BFP to 32)	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx			B39D	19-25
CLFDTR	CONVERT TO LOGICAL (long DFP to 32)	RRF-e	C	F	¤[7,9]			Dt	Xi		Xx			B943	20-32
CLFEBR	CONVERT TO LOGICAL (short BFP to 32)	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx			B39C	19-25
CLFHSI	COMPARE LOGICAL IMMEDIATE (32←16)	SIL	C	GE		A							B_1	E55D	7-152
CLFI	COMPARE LOGICAL IMMEDIATE (32)	RIL-a	C	EI										C2F	7-152
CLFIT	COMPARE LOGICAL IMMEDIATE AND TRAP (32←16)	RIE-a		GE					Dc					EC73	7-156
CLFXBR	CONVERT TO LOGICAL (extended BFP to 32)	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx			B39E	19-25

Figure B-2. Instructions Arranged by Mnemonic (Part 4 of 23)

Mnemonic	Name	Characteristics													Opcode	Page
CLFXTR	CONVERT TO LOGICAL (extended DFP to 32)	RRF-e	C	F	¤[7,9]		SP	Dt	Xi		Xx				B94B	20-32
CLG	COMPARE LOGICAL (64)	RXY-a	C	N		A								B_2	E321	7-152
CLGDBR	CONVERT TO LOGICAL (long BFP to 64)	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx				B3AD	19-25
CLGDTR	CONVERT TO LOGICAL (long DFP to 64)	RRF-e	C	F	¤[7,9]			Dt	Xi		Xx				B942	20-32
CLGEBR	CONVERT TO LOGICAL (short BFP to 64)	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx				B3AC	19-25
CLGF	COMPARE LOGICAL (64←32)	RXY-a	C	N		A								B_2	E331	7-152
CLGFI	COMPARE LOGICAL IMMEDIATE (64←32)	RIL-a	C	EI											C2E	7-152
CLGFR	COMPARE LOGICAL (64←32)	RRE	C	N											B931	7-152
CLGFRL	COMPARE LOGICAL RELATIVE LONG (64←32)	RIL-b	C	GE		A*	SP								C6E	7-153
CLGHRL	COMPARE LOGICAL RELATIVE LONG (64←16)	RIL-b	C	GE		A*									C66	7-153
CLGHSI	COMPARE LOGICAL IMMEDIATE (64←16)	SIL	C	GE		A								B_1	E559	7-152
CLGIB	COMPARE LOGICAL IMMEDIATE AND BRANCH (64←8)	RIS		GE	¤[9]							B			ECFD	7-154
CLGIJ	COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (64←8)	RIE-c		GE	¤[10]							B			EC7D	7-154
CLGIT	COMPARE LOGICAL IMMEDIATE AND TRAP (64←16)	RIE-a		GE					Dc						EC71	7-156
CLGR	COMPARE LOGICAL (64)	RRE	C	N											B921	7-152
CLGRB	COMPARE LOGICAL AND BRANCH (64)	RRS		GE	¤[9]							B			ECE5	7-154
CLGRJ	COMPARE LOGICAL AND BRANCH RELATIVE (64)	RIE-b		GE	¤[10]							B			EC65	7-154
CLGRL	COMPARE LOGICAL RELATIVE LONG (64)	RIL-b	C	GE		A*	SP								C6A	7-153
CLGRT	COMPARE LOGICAL AND TRAP (64)	RRF-c		GE					Dc						B961	7-155
CLGT	COMPARE LOGICAL AND TRAP (64)	RSY-b		MI1		A			Dc					B_2	EB2B	7-155
CLGXBR	CONVERT TO LOGICAL (extended BFP to 64)	RRF-e	C	F	¤[7,9]		SP	Db	Xi		Xx				B3AE	19-25
CLGXTR	CONVERT TO LOGICAL (extended DFP to 64)	RRF-e	C	F	¤[7,9]		SP	Dt	Xi		Xx				B94A	20-32
CLHF	COMPARE LOGICAL HIGH (32)	RXY-a	C	HW		A								B_2	E3CF	7-157
CLHHR	COMPARE LOGICAL HIGH (32)	RRE	C	HW											B9CF	7-157
CLHHSI	COMPARE LOGICAL IMMEDIATE (16←16)	SIL	C	GE		A								B_1	E555	7-152
CLHLR	COMPARE LOGICAL HIGH (32)	RRE	C	HW											B9DF	7-157
CLHRL	COMPARE LOGICAL RELATIVE LONG (32←16)	RIL-b	C	GE		A*									C67	7-153
CLI	COMPARE LOGICAL (immediate)	SI	C			A								B_1	95	7-152
CLIB	COMPARE LOGICAL IMMEDIATE AND BRANCH (32←8)	RIS		GE	¤[9]							B			ECFF	7-154
CLIH	COMPARE LOGICAL IMMEDIATE HIGH (32)	RIL-a	C	HW											CCF	7-158
CLIJ	COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (32←8)	RIE-c		GE	¤[10]							B			EC7F	7-154
CLIY	COMPARE LOGICAL (immediate)	SIY	C	LD		A								B_1	EB55	7-152
CLM	COMPARE LOGICAL CHAR. UNDER MASK (low)	RS-b	C			A								B_2	BD	7-157
CLMH	COMPARE LOGICAL CHAR. UNDER MASK (high)	RSY-b	C	N		A								B_2	EB20	7-157
CLMY	COMPARE LOGICAL CHAR. UNDER MASK (low)	RSY-b	C	LD		A								B_2	EB21	7-157
CLR	COMPARE LOGICAL (32)	RR	C												15	7-152
CLRB	COMPARE LOGICAL AND BRANCH (32)	RRS		GE	¤[9]							B			ECF7	7-154
CLRJ	COMPARE LOGICAL AND BRANCH RELATIVE (32)	RIE-b		GE	¤[10]							B			EC77	7-154
CLRL	COMPARE LOGICAL RELATIVE LONG (32)	RIL-b	C	GE		A*	SP								C6F	7-153
CLRT	COMPARE LOGICAL AND TRAP (32)	RRF-c		GE					Dc						B973	7-155
CLST	COMPARE LOGICAL STRING	RRE	C		¤[9]	A	SP	IC		G0				R_1 R_2	B25D	7-166
CLT	COMPARE LOGICAL AND TRAP (32)	RSY-b		MI1		A			Dc					B_2	EB23	7-155
CLY	COMPARE LOGICAL (32)	RXY-a	C	LD		A								B_2	E355	7-152
CMPSC	COMPRESSION CALL	RRE	C		¤[5,9]	A	SP	II	Dg	GM			ST	R_1 R_2	B263	7-170
CP	COMPARE DECIMAL	SS-b	C		¤[9]	A		Dg						B_1 B_2	F9	8-7
CPDT	CONVERT TO PACKED (from long DFP)	RSL-b	C	PC	¤[7,9]	A	SP	Dt	DF				ST	B_2	EDAC	20-33
CPSDR	COPY SIGN (long)	RRF-b		FS	¤[7,9]			Da							B372	9-30
CPXT	CONVERT TO PACKED (from extended DFP)	RSL-b	C	PC	¤[7,9]	A	SP	Dt	DF				ST	B_2	EDAD	20-33
CPYA	COPY ACCESS	RRE			¤[6]									U_1 U_2	B24D	7-252

Figure B-2. Instructions Arranged by Mnemonic (Part 5 of 23)

Mnemonic	Name	Characteristics													Opcode	Page
CR	COMPARE (32)	RR	C												19	7-134
CRB	COMPARE AND BRANCH (32)	RRS		GE	¤[9]							B			ECF6	7-135
CRDTE	COMPARE AND REPLACE DAT TABLE ENTRY	RRF-b		ED2	P	A[1]	SP		$						B98F	10-17
CRJ	COMPARE AND BRANCH RELATIVE (32)	RIE-b		GE	¤[10]							B			EC76	7-135
CRL	COMPARE RELATIVE LONG (32)	RIL-b	C	GE		A*	SP								C6D	7-135
CRT	COMPARE AND TRAP (32)	RRF-c		GE				Dc							B972	7-149
CS	COMPARE AND SWAP (32)	RS-a	C		¤[9]	A	SP		$			ST		B_2	BA	7-144
CSCH	CLEAR SUBCHANNEL	S	C		P			OP	¢	GS					B230	14-5
CSDTR	CONVERT TO SIGNED PACKED (long DFP to 64)	RRF-d		TF	¤[7,9]			Dt							B3E3	20-35
CSG	COMPARE AND SWAP (64)	RSY-a	C	N	¤[9]	A	SP		$			ST		B_2	EB30	7-144
CSP	COMPARE AND SWAP AND PURGE (32)	RRE	C		P	A[1]	SP		$			ST		R_2	B250	10-21
CSPG	COMPARE AND SWAP AND PURGE (64)	RRE	C	DE	P	A[1]	SP		$			ST		R_2	B98A	10-21
CSST	COMPARE AND SWAP AND STORE	SSF	C	CS	¤[1]	A	SP		$	GM		ST	B_1	B_2	C82	7-146
CSXTR	CONVERT TO SIGNED PACKED (extended DFP to 128)	RRF-d		TF	¤[7,9]		SP	Dt							B3EB	20-35
CSY	COMPARE AND SWAP (32)	RSY-a	C	LD	¤[9]	A	SP		$			ST		B_2	EB14	7-144
CU12	CONVERT UTF-8 TO UTF-16	RRF-c	C		¤[5,9]	A	SP	IC				ST	R_1	R_2	B2A7	7-244
CU14	CONVERT UTF-8 TO UTF-32	RRF-c	C	E3	¤[5,9]	A	SP	IC				ST	R_1	R_2	B9B0	7-248
CU21	CONVERT UTF-16 TO UTF-8	RRF-c	C		¤[5,9]	A	SP	IC				ST	R_1	R_2	B2A6	7-234
CU24	CONVERT UTF-16 TO UTF-32	RRF-c	C	E3	¤[5,9]	A	SP	IC				ST	R_1	R_2	B9B1	7-231
CU41	CONVERT UTF-32 TO UTF-8	RRE	C	E3	¤[5,9]	A	SP	IC				ST	R_1	R_2	B9B2	7-241
CU42	CONVERT UTF-32 TO UTF-16	RRE	C	E3	¤[5,9]	A	SP	IC				ST	R_1	R_2	B9B3	7-238
CUDTR	CONVERT TO UNSIGNED PACKED (long DFP to 64)	RRE		TF	¤[7,9]			Dt							B3E2	20-35
CUSE	COMPARE UNTIL SUBSTRING EQUAL	RRE	C		¤[9]	A	SP	Il		GM			R_1	R_2	B257	7-167
CUTFU	CONVERT UTF-8 TO UNICODE	RRF-c	C		¤[5,9]	A	SP	IC				ST	R_1	R_2	B2A7	7-244
CUUTF	CONVERT UNICODE TO UTF-8	RRF-c	C		¤[5,9]	A	SP	IC				ST	R_1	R_2	B2A6	7-234
CUXTR	CONVERT TO UNSIGNED PACKED (extended DFP to 128)	RRE		TF	¤[7,9]		SP	Dt							B3EA	20-35
CVB	CONVERT TO BINARY (32)	RX-a			¤[9]	A			Dg	IK				B_2	4F	7-230
CVBG	CONVERT TO BINARY (64)	RXY-a		N	¤[9]	A			Dg	IK				B_2	E30E	7-230
CVBY	CONVERT TO BINARY (32)	RXY-a		LD	¤[9]	A			Dg	IK				B_2	E306	7-230
CVD	CONVERT TO DECIMAL (32)	RX-a			¤[9]	A						ST		B_2	4E	7-231
CVDG	CONVERT TO DECIMAL (64)	RXY-a		N	¤[9]	A						ST		B_2	E32E	7-231
CVDY	CONVERT TO DECIMAL (32)	RXY-a		LD	¤[9]	A						ST		B_2	E326	7-231
CXBR	COMPARE (extended BFP)	RRE	C		¤[7,9]		SP	Db	Xi						B349	19-17
CXFBR	CONVERT FROM FIXED (32 to extended BFP)	RRE			¤[7,9]		SP	Db							B396	19-19
CXFBRA	CONVERT FROM FIXED (32 to extended BFP)	RRF-e		F	¤[7,9]		SP	Db							B396	19-19
CXFR	CONVERT FROM FIXED (32 to extended HFP)	RRE			¤[7,9]		SP	Da							B3B6	18-11
CXFTR	CONVERT FROM FIXED (32 to extended DFP)	RRE		F	¤[7,9]		SP	Dt							B959	20-24
CXGBR	CONVERT FROM FIXED (64 to extended BFP)	RRE		N	¤[7,9]		SP	Db							B3A6	19-19
CXGBRA	CONVERT FROM FIXED (64 to extended BFP)	RRF-e		F	¤[7,9]		SP	Db							B3A6	19-19
CXGR	CONVERT FROM FIXED (64 to extended HFP)	RRE		N	¤[7,9]		SP	Da							B3C6	18-11
CXGTR	CONVERT FROM FIXED (64 to extended DFP)	RRE		TF	¤[7,9]		SP	Dt							B3F9	20-24
CXGTRA	CONVERT FROM FIXED (64 to extended DFP)	RRF-e		F	¤[7,9]		SP	Dt							B3F9	20-24
CXLFBR	CONVERT FROM LOGICAL (32 to extended BFP)	RRF-e		F	¤[7,9]		SP	Db							B392	19-21
CXLFTR	CONVERT FROM LOGICAL (32 to extended DFP)	RRF-e		F	¤[7,9]		SP	Dt							B95B	20-25
CXLGBR	CONVERT FROM LOGICAL (64 to extended BFP)	RRF-e		F	¤[7,9]		SP	Db							B3A2	19-21
CXLGTR	CONVERT FROM LOGICAL (64 to extended DFP)	RRF-e		F	¤[7,9]		SP	Dt							B95A	20-25
CXPT	CONVERT FROM PACKED (to extended DFP)	RSL-b		PC	¤[7,9]	A	SP	Dt	Dg					B_2	EDAF	20-26
CXR	COMPARE (extended HFP)	RRE	C		¤[7,9]		SP	Da							B369	18-10
CXSTR	CONVERT FROM SIGNED PACKED (128 to extended DFP)	RRE		TF	¤[7,9]		SP	Dt	Dg						B3FB	20-28
CXTR	COMPARE (extended DFP)	RRE	C	TF	¤[7,9]		SP	Dt	Xi						B3EC	20-22
CXUTR	CONVERT FROM UNSIGNED PACKED (128 to ext. DFP)	RRE		TF	¤[7,9]		SP	Dt	Dg						B3FA	20-28

Figure B-2. Instructions Arranged by Mnemonic (Part 6 of 23)

Mnemonic	Name	Characteristics																																Opcode	Page
CXZT	CONVERT FROM ZONED (to extended DFP)	RSL-b		ZF	¤[7,9]		A		SP	Dt		Dg																		B_2				EDAB	20-29
CY	COMPARE (32)	RXY-a	C	LD			A																							B_2				E359	7-134
CZDT	CONVERT TO ZONED (from long DFP)	RSL-b	C	ZF	¤[7,9]		A		SP																					B_2				EDA8	20-36
CZXT	CONVERT TO ZONED (from extended DFP)	RSL-b	C	ZF	¤[7,9]		A		SP																					B_2				EDA9	20-36
D	DIVIDE (32←64)	RX-a			¤[9]		A		SP					IK															B_2				5D	7-252
DD	DIVIDE (long HFP)	RX-a			¤[7,9]		A			Da		EU		EO		FK													B_2				6D	18-12
DDB	DIVIDE (long BFP)	RXE			¤[7,9]		A			Db		Xi		Xz		Xo		Xu		Xx									B_2				ED1D	19-27
DDBR	DIVIDE (long BFP)	RRE			¤[7,9]					Db		Xi		Xz		Xo		Xu		Xx													B31D	19-27
DDR	DIVIDE (long HFP)	RR			¤[7,9]					Da		EU		EO		FK																	2D	18-12
DDTR	DIVIDE (long DFP)	RRF-a		TF	¤[7,9]					Dt		Xi		Xz		Xo		Xu		Xx													B3D1	20-37
DDTRA	DIVIDE (long DFP)	RRF-a		F	¤[7,9]					Dt		Xi		Xz		Xo		Xu		Xx		Xq											B3D1	20-37
DE	DIVIDE (short HFP)	RX-a			¤[7,9]		A			Da		EU		EO		FK													B_2				7D	18-12
DEB	DIVIDE (short BFP)	RXE			¤[7,9]		A			Db		Xi		Xz		Xo		Xu		Xx									B_2				ED0D	19-27
DEBR	DIVIDE (short BFP)	RRE			¤[7,9]					Db		Xi		Xz		Xo		Xu		Xx													B30D	19-27
DER	DIVIDE (short HFP)	RR			¤[7,9]					Da		EU		EO		FK																	3D	18-12
DIDBR	DIVIDE TO INTEGER (long BFP)	RRF-b	C		¤[7,9]				SP	Db		Xi						Xu		Xx													B35B	19-28
DIEBR	DIVIDE TO INTEGER (short BFP)	RRF-b	C		¤[7,9]				SP	Db		Xi						Xu		Xx													B353	19-28
DL	DIVIDE LOGICAL (32←64)	RXY-a		N3	¤[9]		A		SP					IK															B_2				E397	7-253
DLG	DIVIDE LOGICAL (64←128)	RXY-a		N	¤[9]		A		SP					IK															B_2				E387	7-253
DLGR	DIVIDE LOGICAL (64←128)	RRE		N	¤[9]				SP					IK																			B987	7-253
DLR	DIVIDE LOGICAL (32←64)	RRE		N3	¤[9]				SP					IK																			B997	7-253
DP	DIVIDE DECIMAL	SS-b			¤[9]		A		SP	Dg				DK														ST	B_1	B_2			FD	8-7
DR	DIVIDE (32←64)	RR			¤[9]				SP					IK																			1D	7-252
DSG	DIVIDE SINGLE (64)	RXY-a		N	¤[9]		A		SP					IK															B_2				E30D	7-254
DSGF	DIVIDE SINGLE (64←32)	RXY-a		N	¤[9]		A		SP					IK															B_2				E31D	7-254
DSGFR	DIVIDE SINGLE (64←32)	RRE		N	¤[9]				SP					IK																			B91D	7-254
DSGR	DIVIDE SINGLE (64)	RRE		N	¤[9]				SP					IK																			B90D	7-254
DXBR	DIVIDE (extended BFP)	RRE			¤[7,9]				SP	Db		Xi		Xz		Xo		Xu		Xx													B34D	19-27
DXR	DIVIDE (extended HFP)	RRE			¤[7,9]				SP	Da		EU		EO		FK																	B22D	18-12
DXTR	DIVIDE (extended DFP)	RRF-a		TF	¤[7,9]				SP	Dt		Xi		Xz		Xo		Xu		Xx													B3D9	20-37
DXTRA	DIVIDE (extended DFP)	RRF-a		F	¤[7,9]				SP	Dt		Xi		Xz		Xo		Xu		Xx		Xq											B3D9	20-37
EAR	EXTRACT ACCESS	RRE																												U_2				B24F	7-257
ECAG	EXTRACT CPU ATTRIBUTE	RSY-a		GE	¤[9]																													EB4C	7-257
ECTG	EXTRACT CPU TIME	SSF		ET	¤[8,9]		A									GM													R_3	B_1	B_2		C81	7-260
ED	EDIT	SS-a	C		¤[9]		A			Dg																			ST	B_1	B_2		DE	8-8
EDMK	EDIT AND MARK	SS-a	C		¤[9]		A			Dg						G1													ST	B_1	B_2		DF	8-11
EEDTR	EXTRACT BIASED EXPONENT (long DFP to 64)	RRE		TF	¤[7,9]					Dt																							B3E5	20-39
EEXTR	EXTRACT BIASED EXPONENT (extended DFP to 64)	RRE		TF	¤[7,9]				SP	Dt																							B3ED	20-39
EFPC	EXTRACT FPC	RRE			¤[7,9]					Db																							B38C	9-30
EPAIR	EXTRACT PRIMARY ASN AND INSTANCE	RRE		RA	Q					SO																							B99A	10-24
EPAR	EXTRACT PRIMARY ASN	RRE			Q					SO																							B226	10-24
EPSW	EXTRACT PSW	RRE		N3	¤[8,9]																												B98D	7-261
EREG	EXTRACT STACKED REGISTERS (32)	RRE			¤[1]		A[1]*			SE																				U_1	U_2		B249	10-25
EREGG	EXTRACT STACKED REGISTERS (64)	RRE		N	¤[1]		A[1]*			SE																				U_1	U_2		B90E	10-25
ESAIR	EXTRACT SECONDARY ASN AND INSTANCE	RRE		RA	Q					SO																							B99B	10-25
ESAR	EXTRACT SECONDARY ASN	RRE			Q					SO																							B227	10-24
ESDTR	EXTRACT SIGNIFICANCE (long DFP to 64)	RRE		TF	¤[7,9]					Dt																							B3E7	20-39
ESEA	EXTRACT AND SET EXTENDED AUTHORITY	RRE		N	P																												B99D	10-24
ESTA	EXTRACT STACKED STATE	RRE	C		¤[1]		A[1]*		SP	SE																							B24A	10-26
ESXTR	EXTRACT SIGNIFICANCE (extended DFP to 64)	RRE		TF	¤[7,9]				SP	Dt																							B3EF	20-39
ETND	EXTRACT TRANSACTION NESTING DEPTH	RRE		TX	¤[9]					SO																							B2EC	7-261
EX	EXECUTE	RX-a			¤[9]		AI		SP							EX																	44	7-256
EXRL	EXECUTE RELATIVE LONG	RIL-b		XX	¤[9]		AI*									EX																	C60	7-256
FIDBR	LOAD FP INTEGER (long BFP)	RRF-e			¤[7,9]				SP	Db		Xi								Xx													B35F	19-32

Figure B-2. Instructions Arranged by Mnemonic (Part 7 of 23)

Mnemonic	Name	Characteristics																							Opcode	Page
FIDBRA	LOAD FP INTEGER (long BFP)	RRF-e		F				¤[7,9]		SP	Db		Xi				Xx								B35F	19-32
FIDR	LOAD FP INTEGER (long HFP)	RRE						¤[7,9]			Da														B37F	18-15
FIDTR	LOAD FP INTEGER (long DFP)	RRF-e		TF				¤[7,9]			Dt		Xi				Xx	Xq							B3D7	20-42
FIEBR	LOAD FP INTEGER (short BFP)	RRF-e						¤[7,9]		SP	Db		Xi				Xx								B357	19-32
FIEBRA	LOAD FP INTEGER (short BFP)	RRF-e		F				¤[7,9]		SP	Db		Xi				Xx								B357	19-32
FIER	LOAD FP INTEGER (short HFP)	RRE						¤[7,9]			Da														B377	18-15
FIXBR	LOAD FP INTEGER (extended BFP)	RRF-e						¤[7,9]		SP	Db		Xi				Xx								B347	19-32
FIXBRA	LOAD FP INTEGER (extended BFP)	RRF-e		F				¤[7,9]		SP	Db		Xi				Xx								B347	19-32
FIXR	LOAD FP INTEGER (extended HFP)	RRE						¤[7,9]		SP	Da														B367	18-15
FIXTR	LOAD FP INTEGER (extended DFP)	RRF-e		TF				¤[7,9]		SP	Dt		Xi				Xx	Xq							B3DF	20-42
FLOGR	FIND LEFTMOST ONE	RRE	C	EI						SP															B983	7-262
HDR	HALVE (long HFP)	RR						¤[7,9]			Da		EU												24	18-13
HER	HALVE (short HFP)	RR						¤[7,9]			Da		EU												34	18-13
HSCH	HALT SUBCHANNEL	S	C					P			OP			¢		GS									B231	14-6
IAC	INSERT ADDRESS SPACE CONTROL	RRE	C					Q			SO														B224	10-29
IC	INSERT CHARACTER	RX-a							A															B_2	43	7-262
ICM	INSERT CHARACTERS UNDER MASK (low)	RS-b	C						A															B_2	BF	7-262
ICMH	INSERT CHARACTERS UNDER MASK (high)	RSY-b	C	N					A															B_2	EB80	7-262
ICMY	INSERT CHARACTERS UNDER MASK (low)	RSY-b	C	LD					A															B_2	EB81	7-262
ICY	INSERT CHARACTER	RXY-a		LD					A															B_2	E373	7-262
IDTE	INVALIDATE DAT TABLE ENTRY	RRF-b	U	DE				P	A[1]	SP				$											B98E	10-32
IEDTR	INSERT BIASED EXPONENT (64 to long DFP)	RRF-b		TF				¤[7,9]			Dt														B3F6	20-40
IEXTR	INSERT BIASED EXPONENT (64 to extended DFP)	RRF-b		TF				¤[7,9]		SP	Dt														B3FE	20-40
IIHF	INSERT IMMEDIATE (high)	RIL-a		EI																					C08	7-263
IIHH	INSERT IMMEDIATE (high high)	RI-a		N																					A50	7-263
IIHL	INSERT IMMEDIATE (high low)	RI-a		N																					A51	7-263
IILF	INSERT IMMEDIATE (low)	RIL-a		EI																					C09	7-263
IILH	INSERT IMMEDIATE (low high)	RI-a		N																					A52	7-263
IILL	INSERT IMMEDIATE (low low)	RI-a		N																					A53	7-263
IPK	INSERT PSW KEY	S						Q								G2									B20B	10-30
IPM	INSERT PROGRAM MASK	RRE																							B222	7-264
IPTE	INVALIDATE PAGE TABLE ENTRY	RRF-a						P	A[1]	SP				$											B221	10-37
IRBM	INSERT REFERENCE BITS MULTIPLE	RRE		IM				P	A[1]*																B9AC	10-30
ISKE	INSERT STORAGE KEY EXTENDED	RRE						P	A[1]*																B229	10-30
IVSK	INSERT VIRTUAL STORAGE KEY	RRE						Q	A[1]*		SO													R_2	B223	10-31
KDB	COMPARE AND SIGNAL (long BFP)	RXE	C					¤[7,9]	A		Db		Xi											B_2	ED18	19-18
KDBR	COMPARE AND SIGNAL (long BFP)	RRE	C					¤[7,9]			Db		Xi												B318	19-18
KDTR	COMPARE AND SIGNAL (long DFP)	RRE	C	TF				¤[7,9]			Dt		Xi												B3E0	20-23
KEB	COMPARE AND SIGNAL (short BFP)	RXE	C					¤[7,9]	A		Db		Xi											B_2	ED08	19-18
KEBR	COMPARE AND SIGNAL (short BFP)	RRE	C					¤[7,9]			Db		Xi												B308	19-18
KIMD	COMPUTE INTERMEDIATE MESSAGE DIGEST	RRE	C	MS				¤[5,9]	A	SP	IC					GM	I1				ST			R_2	B93E	7-188
KLMD	COMPUTE LAST MESSAGE DIGEST	RRE	C	MS				¤[5,9]	A	SP	IC					GM	I1				ST			R_2	B93F	7-201
KM	CIPHER MESSAGE	RRE	C	MS				¤[5,9]	A	SP	IC					GM	I1				ST		R_1	R_2	B92E	7-53
KMA	CIPHER MESSAGE WITH AUTHENTICATION	RRF-b	C	M8				¤[5,9]	A	SP	IC					GM	I1				ST		R_1	R_2 R_3	B929	7-78
KMAC	COMPUTE MESSAGE AUTHENTICATION CODE	RRE	C	MS				¤[5,9]	A	SP	IC					GM	I1				ST			R_2	B91E	7-219
KMC	CIPHER MESSAGE WITH CHAINING	RRE	C	MS				¤[5,9]	A	SP	IC					GM	I1				ST		R_1	R_2	B92F	7-53
KMCTR	CIPHER MESSAGE WITH COUNTER	RRF-b	C	M4				¤[5,9]	A	SP	IC					GM	I1				ST		R_1,R_2,R_3		B92D	7-107
KMF	CIPHER MESSAGE WITH CIPHER FEEDBACK	RRE	C	M4				¤[5,9]	A	SP	IC					GM	I1				ST		R_1	R_2	B92A	7-92
KMO	CIPHER MESSAGE WITH OUTPUT FEEDBACK	RRE	C	M4				¤[5,9]	A	SP	IC					GM	I1				ST		R_1	R_2	B92B	7-120
KXBR	COMPARE AND SIGNAL (extended BFP)	RRE	C					¤[7,9]		SP	Db		Xi												B348	19-18
KXTR	COMPARE AND SIGNAL (extended DFP)	RRE	C	TF				¤[7,9]		SP	Dt		Xi												B3E8	20-23
L	LOAD (32)	RX-a							A															B_2	58	7-264
LA	LOAD ADDRESS	RX-a																							41	7-266
LAA	LOAD AND ADD (32)	RSY-a	C	IA				¤[9]	A	SP			IF	£							ST			B_2	EBF8	7-268
LAAG	LOAD AND ADD (64)	RSY-a	C	IA				¤[9]	A	SP			IF	£							ST			B_2	EBE8	7-268

Figure B-2. Instructions Arranged by Mnemonic (Part 8 of 23)

Mnemonic	Name	Characteristics															Op-code	Page
LAAL	LOAD AND ADD LOGICAL (32)	RSY-a	C	IA	¤[9]	A	SP		£			ST		B_2			EBFA	7-268
LAALG	LOAD AND ADD LOGICAL (64)	RSY-a	C	IA	¤[9]	A	SP		£			ST		B_2			EBEA	7-268
LAE	LOAD ADDRESS EXTENDED	RX-a			¤[6]								U_1	BP			51	7-266
LAEY	LOAD ADDRESS EXTENDED	RXY-a		GE	¤[6]								U_1	BP			E375	7-266
LAM	LOAD ACCESS MULTIPLE	RS-a			¤[6]	A	SP							UB			9A	7-265
LAMY	LOAD ACCESS MULTIPLE	RSY-a		LD	¤[6]	A	SP							UB			EB9A	7-265
LAN	LOAD AND AND (32)	RSY-a	C	IA	¤[9]	A	SP		£			ST		B_2			EBF4	7-269
LANG	LOAD AND AND (64)	RSY-a	C	IA	¤[9]	A	SP		£			ST		B_2			EBE4	7-269
LAO	LOAD AND OR (32)	RSY-a	C	IA	¤[9]	A	SP		£			ST		B_2			EBF6	7-270
LAOG	LOAD AND OR (64)	RSY-a	C	IA	¤[9]	A	SP		£			ST		B_2			EBE6	7-270
LARL	LOAD ADDRESS RELATIVE LONG	RIL-b		N3													C00	7-267
LASP	LOAD ADDRESS SPACE PARAMETERS	SSE	C		P	A[1]	SP	SO					B_1				E500	10-41
LAT	LOAD AND TRAP (32L←32)	RXY-a		LT		A			Dc					B_2			E39F	7-271
LAX	LOAD AND EXCLUSIVE OR (32)	RSY-a	C	IA	¤[9]	A	SP		£			ST		B_2			EBF7	7-269
LAXG	LOAD AND EXCLUSIVE OR (64)	RSY-a	C	IA	¤[9]	A	SP		£			ST		B_2			EBE7	7-269
LAY	LOAD ADDRESS	RXY-a		LD													E371	7-266
LB	LOAD BYTE (32←8)	RXY-a		LD		A											E376	7-272
LBH	LOAD BYTE HIGH (32←8)	RXY-a		HW		A								B_2			E3C0	7-272
LBR	LOAD BYTE (32←8)	RRE		EI													B926	7-272
LCBB	LOAD COUNT TO BLOCK BOUNDARY	RXE	C	VF			SP										E727	7-273
LCDBR	LOAD COMPLEMENT (long BFP)	RRE	C		¤[7,9]			Db									B313	19-31
LCDFR	LOAD COMPLEMENT (long)	RRE		FS	¤[7,9]			Da									B373	9-31
LCDR	LOAD COMPLEMENT (long HFP)	RR	C		¤[7,9]			Da									23	18-14
LCEBR	LOAD COMPLEMENT (short BFP)	RRE	C		¤[7,9]			Db									B303	19-31
LCER	LOAD COMPLEMENT (short HFP)	RR	C		¤[7,9]			Da									33	18-14
LCGFR	LOAD COMPLEMENT (64←32)	RRE	C	N													B913	7-273
LCGR	LOAD COMPLEMENT (64)	RRE	C	N					IF								B903	7-273
LCR	LOAD COMPLEMENT (32)	RR	C						IF								13	7-272
LCTL	LOAD CONTROL (32)	RS-a			P	A	SP							B_2			B7	10-50
LCTLG	LOAD CONTROL (64)	RSY-a		N	P	A	SP							B_2			EB2F	10-50
LCXBR	LOAD COMPLEMENT (extended BFP)	RRE	C		¤[7,9]		SP	Db									B343	19-31
LCXR	LOAD COMPLEMENT (extended HFP)	RRE	C		¤[7,9]		SP	Da									B363	18-14
LD	LOAD (long)	RX-a			¤[7,9]	A		Da						B_2			68	9-31
LDE	LOAD LENGTHENED (short to long HFP)	RXE			¤[7,9]	A		Da						B_2			ED24	18-15
LDEB	LOAD LENGTHENED (short to long BFP)	RXE			¤[7,9]	A		Db	Xi					B_2			ED04	19-34
LDEBR	LOAD LENGTHENED (short to long BFP)	RRE			¤[7,9]			Db	Xi								B304	19-33
LDER	LOAD LENGTHENED (short to long HFP)	RRE			¤[7,9]			Da									B324	18-15
LDETR	LOAD LENGTHENED (short to long DFP)	RRF-d		TF	¤[7,9]			Dt	Xi								B3D4	20-45
LDGR	LOAD FPR FROM GR (64 to long)	RRE		FG	¤[7,9]			Da									B3C1	9-34
LDR	LOAD (long)	RR			¤[7,9]			Da									28	9-31
LDXBR	LOAD ROUNDED (extended to long BFP)	RRE			¤[7,9]		SP	Db	Xi		Xo	Xu	Xx				B345	19-35
LDXBRA	LOAD ROUNDED (extended to long BFP)	RRF-e		F	¤[7,9]		SP	Db	Xi		Xo	Xu	Xx				B345	19-35
LDXR	LOAD ROUNDED (extended to long HFP)	RR			¤[7,9]		SP	Da		EO							25	18-17
LDXTR	LOAD ROUNDED (extended to long DFP)	RRF-e		TF	¤[7,9]		SP	Dt	Xi		Xo	Xu	Xx	Xq			B3DD	20-46
LDY	LOAD (long)	RXY-a		LD	¤[7,9]	A		Da						B_2			ED65	9-31
LE	LOAD (short)	RX-a			¤[7,9]	A		Da						B_2			78	9-31
LEDBR	LOAD ROUNDED (long to short BFP)	RRE			¤[7,9]			Db	Xi		Xo	Xu	Xx				B344	19-35
LEDBRA	LOAD ROUNDED (long to short BFP)	RRF-e		F	¤[7,9]		SP	Db	Xi		Xo	Xu	Xx				B344	19-35
LEDR	LOAD ROUNDED (long to short HFP)	RR			¤[7,9]			Da		EO							35	18-17
LEDTR	LOAD ROUNDED (long to short DFP)	RRF-e		TF	¤[7,9]			Dt	Xi		Xo	Xu	Xx	Xq			B3D5	20-46
LER	LOAD (short)	RR			¤[7,9]			Da									38	9-31
LEXBR	LOAD ROUNDED (extended to short BFP)	RRE			¤[7,9]		SP	Db	Xi		Xo	Xu	Xx				B346	19-35
LEXBRA	LOAD ROUNDED (extended to short BFP)	RRF-e		F	¤[7,9]		SP	Db	Xi		Xo	Xu	Xx				B346	19-35
LEXR	LOAD ROUNDED (extended to short HFP)	RRE			¤[7,9]		SP	Da		EO							B366	18-17
LEY	LOAD (short)	RXY-a		LD	¤[7,9]	A		Da						B_2			ED64	9-31

Figure B-2. Instructions Arranged by Mnemonic (Part 9 of 23)

Mnemonic	Name	Characteristics									Opcode	Page
LFAS	LOAD FPC AND SIGNAL	S	XF	¤[7,9]	A	SP	Dt	Xg		B_2	B2BD	9-32
LFH	LOAD HIGH (32)	RXY-a	HW		A					B_2	E3CA	7-278
LFHAT	LOAD HIGH AND TRAP (32H←32)	RXY-a	LT		A			Dc		B_2	E3C8	7-278
LFPC	LOAD FPC	S		¤[7,9]	A	SP	Db			B_2	B29D	9-31
LG	LOAD (64)	RXY-a	N		A					B_2	E304	7-264
LGAT	LOAD AND TRAP (64)	RXY-a	LT		A			Dc		B_2	E385	7-271
LGB	LOAD BYTE (64←8)	RXY-a	LD		A						E377	7-272
LGBR	LOAD BYTE (64←8)	RRE	EI								B906	7-272
LGDR	LOAD GR FROM FPR (long to 64)	RRE	FG	¤[7,9]			Da				B3CD	9-34
LGF	LOAD (64←32)	RXY-a	N		A					B_2	E314	7-264
LGFI	LOAD IMMEDIATE (64←32)	RIL-a	EI								C01	7-264
LGFR	LOAD (64←32)	RRE	N								B914	7-264
LGFRL	LOAD RELATIVE LONG (64←32)	RIL-b	GE		A*	SP					C4C	7-264
LGG	LOAD GUARDED (64)	RXY-a	GF	¤[12]	A	SP			B ST	B_2	E34C	7-274
LGH	LOAD HALFWORD (64←16)	RXY-a	N		A					B_2	E315	7-276
LGHI	LOAD HALFWORD IMMEDIATE (64←16)	RI-a	N								A79	7-276
LGHR	LOAD HALFWORD (64←16)	RRE	EI								B907	7-276
LGHRL	LOAD HALFWORD RELATIVE LONG (64←16)	RIL-b	GE		A*						C44	7-276
LGR	LOAD (64)	RRE	N								B904	7-264
LGRL	LOAD RELATIVE LONG (64)	RIL-b	GE		A*	SP					C48	7-264
LGSC	LOAD GUARDED STORAGE CONTROLS	RXY-a	GF	¤[1]	A		SO			B_2	E34D	7-275
LH	LOAD HALFWORD (32←16)	RX-a			A					B_2	48	7-276
LHH	LOAD HALFWORD HIGH (32←16)	RXY-a	HW		A					B_2	E3C4	7-277
LHI	LOAD HALFWORD IMMEDIATE (32)←16	RI-a									A78	7-276
LHR	LOAD HALFWORD (32←16)	RRE	EI								B927	7-276
LHRL	LOAD HALFWORD RELATIVE LONG (32←16)	RIL-b	GE		A*						C45	7-276
LHY	LOAD HALFWORD (32←16)	RXY-a	LD		A					B_2	E378	7-276
LLC	LOAD LOGICAL CHARACTER (32←8)	RXY-a	EI		A					B_2	E394	7-279
LLCH	LOAD LOGICAL CHARACTER HIGH (32←8)	RXY-a	HW		A					B_2	E3C2	7-280
LLCR	LOAD LOGICAL CHARACTER (32←8)	RRE	EI								B994	7-279
LLGC	LOAD LOGICAL CHARACTER (64←8)	RXY-a	N		A					B_2	E390	7-279
LLGCR	LOAD LOGICAL CHARACTER (64←8)	RRE	EI								B984	7-279
LLGF	LOAD LOGICAL (64←32)	RXY-a	N		A					B_2	E316	7-278
LLGFAT	LOAD LOGICAL AND TRAP (64←32)	RXY-a	LT		A			Dc		B_2	E39D	7-279
LLGFR	LOAD LOGICAL (64←32)	RRE	N								B916	7-278
LLGFRL	LOAD LOGICAL RELATIVE LONG (64←32)	RIL-b	GE		A*	SP					C4E	7-278
LLGFSG	LOAD LOGICAL AND SHIFT GUARDED (64←32)	RXY-a	GF	¤[12]	A	SP			B ST	B_2	E348	7-274
LLGH	LOAD LOGICAL HALFWORD (64←16)	RXY-a	N		A					B_2	E391	7-280
LLGHR	LOAD LOGICAL HALFWORD (64←16)	RRE	EI								B985	7-280
LLGHRL	LOAD LOGICAL HALFWORD RELATIVE LONG (64←16)	RIL-b	GE		A*						C46	7-280
LLGT	LOAD LOGICAL THIRTY ONE BITS (64←31)	RXY-a	N		A					B_2	E317	7-282
LLGTAT	LOAD LOGICAL THIRTY ONE BITS AND TRAP (64←31)	RXY-a	LT		A			Dc		B_2	E39C	7-282
LLGTR	LOAD LOGICAL THIRTY ONE BITS (64←31)	RRE	N								B917	7-281
LLH	LOAD LOGICAL HALFWORD (32←16)	RXY-a	EI		A					B_2	E395	7-280
LLHH	LOAD LOGICAL HALFWORD HIGH (32←16)	RXY-a	HW		A					B_2	E3C6	7-281
LLHR	LOAD LOGICAL HALFWORD (32←16)	RRE	EI								B995	7-280
LLHRL	LOAD LOGICAL HALFWORD RELATIVE LONG (32←16)	RIL-b	GE		A*						C42	7-280
LLIHF	LOAD LOGICAL IMMEDIATE (high)	RIL-a	EI								C0E	7-281
LLIHH	LOAD LOGICAL IMMEDIATE (high high)	RI-a	N								A5C	7-281
LLIHL	LOAD LOGICAL IMMEDIATE (high low)	RI-a	N								A5D	7-281
LLILF	LOAD LOGICAL IMMEDIATE (low)	RIL-a	EI								C0F	7-281
LLILH	LOAD LOGICAL IMMEDIATE (low high)	RI-a	N								A5E	7-281

Figure B-2. Instructions Arranged by Mnemonic (Part 10 of 23)

Mnemonic	Name	Characteristics											Op-code	Page
LLILL	LOAD LOGICAL IMMEDIATE (low low)	RI-a		N									A5F	7-281
LLZRGF	LOAD LOGICAL AND ZERO RIGHTMOST BYTE (64←32)	RXY-a		LZ		A						B_2	E33A	7-279
LM	LOAD MULTIPLE (32)	RS-a				A						B_2	98	7-282
LMD	LOAD MULTIPLE DISJOINT (64←32&32)	SS-e		N	¤[9]	A					B_2	B_4	EF	7-283
LMG	LOAD MULTIPLE (64)	RSY-a		N		A						B_2	EB04	7-282
LMH	LOAD MULTIPLE HIGH (32)	RSY-a		N		A						B_2	EB96	7-283
LMY	LOAD MULTIPLE (32)	RSY-a		LD		A						B_2	EB98	7-282
LNDBR	LOAD NEGATIVE (long BFP)	RRE	C		¤[7,9]			Db					B311	19-34
LNDFR	LOAD NEGATIVE (long)	RRE		FS	¤[7,9]			Da					B371	9-34
LNDR	LOAD NEGATIVE (long HFP)	RR	C		¤[7,9]			Da					21	18-16
LNEBR	LOAD NEGATIVE (short BFP)	RRE	C		¤[7,9]			Db					B301	19-34
LNER	LOAD NEGATIVE (short HFP)	RR	C		¤[7,9]			Da					31	18-16
LNGFR	LOAD NEGATIVE (64←32)	RRE	C	N									B911	7-284
LNGR	LOAD NEGATIVE (64)	RRE	C	N									B901	7-283
LNR	LOAD NEGATIVE (32)	RR	C										11	7-283
LNXBR	LOAD NEGATIVE (extended BFP)	RRE	C		¤[7,9]		SP	Db					B341	19-34
LNXR	LOAD NEGATIVE (extended HFP)	RRE	C		¤[7,9]		SP	Da					B361	18-16
LOC	LOAD ON CONDITION (32)	RSY-b		L1		A						B_2	EBF2	7-284
LOCFH	LOAD HIGH ON CONDITION (32)	RSY-b		L2		A						B_2	EBE0	7-284
LOCFHR	LOAD HIGH ON CONDITION (32)	RRF-c		L2									B9E0	7-284
LOCG	LOAD ON CONDITION (64)	RSY-b		L1		A						B_2	EBE2	7-284
LOCGHI	LOAD HALFWORD IMMEDIATE ON CONDITION (64←16)	RIE-g		L2									EC46	7-277
LOCGR	LOAD ON CONDITION (64)	RRF-c		L1		A							B9E2	7-284
LOCHHI	LOAD HALFWORD HIGH IMMEDIATE ON CONDITION (32←16)	RIE-g		L2									EC4E	7-277
LOCHI	LOAD HALFWORD IMMEDIATE ON CONDITION (32←16)	RIE-g		L2									EC42	7-277
LOCR	LOAD ON CONDITION (32)	RRF-c		L1		A							B9F2	7-284
LPD	LOAD PAIR DISJOINT (32)	SSF	C	IA	¤[9]	A	SP				B_1	B_2	C84	7-285
LPDBR	LOAD POSITIVE (long BFP)	RRE	C		¤[7,9]			Db					B310	19-35
LPDFR	LOAD POSITIVE (long)	RRE		FS	¤[7,9]			Da					B370	9-34
LPDG	LOAD PAIR DISJOINT (64)	SSF	C	IA	¤[9]	A	SP				B_1	B_2	C85	7-285
LPDR	LOAD POSITIVE (long HFP)	RR	C		¤[7,9]			Da					20	18-16
LPEBR	LOAD POSITIVE (short BFP)	RRE	C		¤[7,9]			Db					B300	19-35
LPER	LOAD POSITIVE (short HFP)	RR	C		¤[7,9]			Da					30	18-16
LPGFR	LOAD POSITIVE (64←32)	RRE	C	N									B910	7-287
LPGR	LOAD POSITIVE (64)	RRE	C	N					IF				B900	7-287
LPQ	LOAD PAIR FROM QUADWORD (64&64←128)	RXY-a		N	¤[9]	A	SP					B_2	E38F	7-286
LPR	LOAD POSITIVE (32)	RR	C						IF				10	7-287
LPSW	LOAD PSW	SI	L		P	A	SP			¢		B_2	82	10-54
LPSWE	LOAD PSW EXTENDED	S	L	N	P	A	SP			¢		B_2	B2B2	10-55
LPTEA	LOAD PAGE TABLE ENTRY ADDRESS	RRF-b	C	D2	P	A[1]*	SP	SO				R_2	B9AA	10-50
LPXBR	LOAD POSITIVE (extended BFP)	RRE	C		¤[7,9]		SP	Db					B340	19-35
LPXR	LOAD POSITIVE (extended HFP)	RRE	C		¤[7,9]		SP	Da					B360	18-16
LR	LOAD (32)	RR											18	7-264
LRA	LOAD REAL ADDRESS (32)	RX-a	C		P	A[1]*		SO				BP	B1	10-56
LRAG	LOAD REAL ADDRESS (64)	RXY-a	C	N	P	A[1]*						BP	E303	10-56
LRAY	LOAD REAL ADDRESS (32)	RXY-a	C	LD	P	A[1]*		SO				BP	E313	10-56
LRDR	LOAD ROUNDED (extended to long HFP)	RR			¤[7,9]		SP	Da		EO			25	18-17
LRER	LOAD ROUNDED (long to short HFP)	RR			¤[7,9]			Da		EO			35	18-17
LRL	LOAD RELATIVE LONG (32)	RIL-b		GE		A	SP						C4D	7-264
LRV	LOAD REVERSED (32)	RXY-a		N3		A						B_2	E31E	7-287
LRVG	LOAD REVERSED (64)	RXY-a		N		A						B_2	E30F	7-287
LRVGR	LOAD REVERSED (64)	RRE		N									B90F	7-287

Figure B-2. Instructions Arranged by Mnemonic (Part 11 of 23)

Mnemonic	Name	Characteristics												Opcode	Page
LRVH	LOAD REVERSED (16)	RXY-a		N3		A							B_2	E31F	7-287
LRVR	LOAD REVERSED (32)	RRE		N3										B91F	7-287
LT	LOAD AND TEST (32)	RXY-a	C	EI		A							B_2	E312	7-271
LTDBR	LOAD AND TEST (long BFP)	RRE	C		¤[7,9]			Db	Xi					B312	19-31
LTDR	LOAD AND TEST (long HFP)	RR	C		¤[7,9]			Da						22	18-13
LTDTR	LOAD AND TEST (long DFP)	RRE	C	TF	¤[7,9]			Dt	Xi					B3D6	20-41
LTEBR	LOAD AND TEST (short BFP)	RRE	C		¤[7,9]			Db	Xi					B302	19-31
LTER	LOAD AND TEST (short HFP)	RR	C		¤[7,9]			Da						32	18-13
LTG	LOAD AND TEST (64)	RXY-a	C	EI		A							B_2	E302	7-271
LTGF	LOAD AND TEST (64←32)	RXY-a	C	GE		A							B_2	E332	7-271
LTGFR	LOAD AND TEST (64←32)	RRE	C	N										B912	7-270
LTGR	LOAD AND TEST (64)	RRE	C	N										B902	7-270
LTR	LOAD AND TEST (32)	RR	C											12	7-270
LTXBR	LOAD AND TEST (extended BFP)	RRE	C		¤[7,9]		SP	Db	Xi					B342	19-31
LTXR	LOAD AND TEST (extended HFP)	RRE	C		¤[7,9]		SP	Da						B362	18-14
LTXTR	LOAD AND TEST (extended DFP)	RRE	C	TF	¤[7,9]		SP	Dt	Xi					B3DE	20-41
LURA	LOAD USING REAL ADDRESS (32)	RRE			P	A[1]	SP							B24B	10-60
LURAG	LOAD USING REAL ADDRESS (64)	RRE		N	P	A[1]	SP							B905	10-60
LXD	LOAD LENGTHENED (long to extended HFP)	RXE			¤[7,9]	A	SP	Da					B_2	ED25	18-15
LXDB	LOAD LENGTHENED (long to extended BFP)	RXE			¤[7,9]	A	SP	Db	Xi				B_2	ED05	19-34
LXDBR	LOAD LENGTHENED (long to extended BFP)	RRE			¤[7,9]		SP	Db	Xi					B305	19-33
LXDR	LOAD LENGTHENED (long to extended HFP)	RRE			¤[7,9]		SP	Da						B325	18-15
LXDTR	LOAD LENGTHENED (long to extended DFP)	RRF-d		TF	¤[7,9]		SP	Dt	Xi					B3DC	20-45
LXE	LOAD LENGTHENED (short to extended HFP)	RXE			¤[7,9]	A	SP	Da					B_2	ED26	18-15
LXEB	LOAD LENGTHENED (short to extended BFP)	RXE			¤[7,9]	A	SP	Db	Xi				B_2	ED06	19-34
LXEBR	LOAD LENGTHENED (short to extended BFP)	RRE			¤[7,9]		SP	Db	Xi					B306	19-33
LXER	LOAD LENGTHENED (short to extended HFP)	RRE			¤[7,9]		SP	Da						B326	18-15
LXR	LOAD (extended)	RRE			¤[7,9]		SP	Da						B365	9-31
LY	LOAD (32)	RXY-a		LD		A							B_2	E358	7-264
LZDR	LOAD ZERO (long)	RRE			¤[7,9]			Da						B375	9-35
LZER	LOAD ZERO (short)	RRE			¤[7,9]			Da						B374	9-35
LZRF	LOAD AND ZERO RIGHTMOST BYTE (32)	RXY-a		LZ		A							B_2	E33B	7-271
LZRG	LOAD AND ZERO RIGHTMOST BYTE (64)	RXY-a		LZ		A							B_2	E32A	7-271
LZXR	LOAD ZERO (extended)	RRE			¤[7,9]		SP	Da						B376	9-35
M	MULTIPLY (64←32)	RX-a				A	SP						B_2	5C	7-304
MAD	MULTIPLY AND ADD (long HFP)	RXF		HM	¤[7,9]	A		Da	EU	EO			B_2	ED3E	18-19
MADB	MULTIPLY AND ADD (long BFP)	RXF			¤[7,9]	A		Db	Xi	Xo	Xu	Xx	B_2	ED1E	19-38
MADBR	MULTIPLY AND ADD (long BFP)	RRD			¤[7,9]			Db	Xi	Xo	Xu	Xx		B31E	19-38
MADR	MULTIPLY AND ADD (long HFP)	RRD		HM	¤[7,9]			Da	EU	EO				B33E	18-19
MAE	MULTIPLY AND ADD (short HFP)	RXF		HM	¤[7,9]	A		Da	EU	EO			B_2	ED2E	18-19
MAEB	MULTIPLY AND ADD (short BFP)	RXF			¤[7,9]	A		Db	Xi	Xo	Xu	Xx	B_2	ED0E	19-38
MAEBR	MULTIPLY AND ADD (short BFP)	RRD			¤[7,9]			Db	Xi	Xo	Xu	Xx		B30E	19-38
MAER	MULTIPLY AND ADD (short HFP)	RRD		HM	¤[7,9]			Da	EU	EO				B32E	18-19
MAY	MULTIPLY & ADD UNNORMALIZED (long to ext. HFP)	RXF		UE	¤[7,9]	A		Da					B_2	ED3A	18-20
MAYH	MULTIPLY AND ADD UNNRM. (long to ext. high HFP)	RXF		UE	¤[7,9]	A		Da					B_2	ED3C	18-20
MAYHR	MULTIPLY AND ADD UNNRM. (long to ext. high HFP)	RRD		UE	¤[7,9]			Da						B33C	18-20
MAYL	MULTIPLY AND ADD UNNRM. (long to ext. low HFP)	RXF		UE	¤[7,9]	A		Da					B_2	ED38	18-20
MAYLR	MULTIPLY AND ADD UNNRM. (long to ext. low HFP)	RRD		UE	¤[7,9]			Da						B338	18-20
MAYR	MULTIPLY & ADD UNNORMALIZED (long to ext. HFP)	RRD		UE	¤[7,9]			Da						B33A	18-20
MC	MONITOR CALL	SI			¤[4,8,9]		SP	ME						AF	7-288

Figure B-2. Instructions Arranged by Mnemonic (Part 12 of 23)

Mnemonic	Name	Characteristics																				Opcode	Page
MD	MULTIPLY (long HFP)	RX-a				¤[7,9]	A			Da	EU	EO								B_2		6C	18-18
MDB	MULTIPLY (long BFP)	RXE				¤[7,9]	A			Db	Xi			Xo	Xu	Xx				B_2		ED1C	19-37
MDBR	MULTIPLY (long BFP)	RRE				¤[7,9]				Db	Xi			Xo	Xu	Xx						B31C	19-37
MDE	MULTIPLY (short to long HFP)	RX-a				¤[7,9]	A			Da	EU	EO								B_2		7C	18-18
MDEB	MULTIPLY (short to long BFP)	RXE				¤[7,9]	A			Db	Xi									B_2		ED0C	19-37
MDEBR	MULTIPLY (short to long BFP)	RRE				¤[7,9]				Db	Xi											B30C	19-37
MDER	MULTIPLY (short to long HFP)	RR				¤[7,9]				Da	EU	EO										3C	18-17
MDR	MULTIPLY (long HFP)	RR				¤[7,9]				Da	EU	EO										2C	18-17
MDTR	MULTIPLY (long DFP)	RRF-a	TF			¤[7,9]				Dt	Xi			Xo	Xu	Xx						B3D0	20-47
MDTRA	MULTIPLY (long DFP)	RRF-a	F			¤[7,9]				Dt	Xi			Xo	Xu	Xx	Xq					B3D0	20-48
ME	MULTIPLY (short to long HFP)	RX-a				¤[7,9]	A			Da	EU	EO								B_2		7C	18-18
MEE	MULTIPLY (short HFP)	RXE				¤[7,9]	A			Da	EU	EO								B_2		ED37	18-18
MEEB	MULTIPLY (short BFP)	RXE				¤[7,9]	A			Db	Xi			Xo	Xu	Xx				B_2		ED17	19-37
MEEBR	MULTIPLY (short BFP)	RRE				¤[7,9]				Db	Xi			Xo	Xu	Xx						B317	19-37
MEER	MULTIPLY (short HFP)	RRE				¤[7,9]				Da	EU	EO										B337	18-17
MER	MULTIPLY (short to long HFP)	RR				¤[7,9]				Da	EU	EO										3C	18-18
MFY	MULTIPLY (64←32)	RXY-a	GE				A	SP												B_2		E35C	7-304
MG	MULTIPLY (128←64)	RXY-a	MI2				A	SP												B_2		E384	7-304
MGH	MULTIPLY HALFWORD (64←16)	RXY-a	MI2				A													B_2		E33C	7-305
MGHI	MULTIPLY HALFWORD IMMEDIATE (64←16)	RI-a	N																			A7D	7-305
MGRK	MULTIPLY (128←64)	RRF-a	MI2					SP														B9EC	7-304
MH	MULTIPLY HALFWORD (32←16)	RX-a					A													B_2		4C	7-305
MHI	MULTIPLY HALFWORD IMMEDIATE (32←16)	RI-a																				A7C	7-305
MHY	MULTIPLY HALFWORD (32←16)	RXY-a	GE				A													B_2		E37C	7-305
ML	MULTIPLY LOGICAL (64←32)	RXY-a	N3				A	SP												B_2		E396	7-306
MLG	MULTIPLY LOGICAL (128←64)	RXY-a	N				A	SP												B_2		E386	7-306
MLGR	MULTIPLY LOGICAL (128←64)	RRE	N					SP														B986	7-306
MLR	MULTIPLY LOGICAL (64←32)	RRE	N3					SP														B996	7-305
MP	MULTIPLY DECIMAL	SS-b				¤[9]	A	SP		Dg									ST	B_1	B_2	FC	8-12
MR	MULTIPLY (64←32)	RR						SP														1C	7-304
MS	MULTIPLY SINGLE (32)	RX-a					A													B_2		71	7-307
MSC	MULTIPLY SINGLE (32)	RXY-a	C	MI2			A				IF									B_2		E353	7-307
MSCH	MODIFY SUBCHANNEL	S	C			P	A	SP		OP		¢		GS						B_2		B232	14-7
MSD	MULTIPLY AND SUBTRACT (long HFP)	RXF		HM		¤[7,9]	A			Da	EU	EO								B_2		ED3F	18-19
MSDB	MULTIPLY AND SUBTRACT (long BFP)	RXF				¤[7,9]	A			Db	Xi			Xo	Xu	Xx				B_2		ED1F	19-38
MSDBR	MULTIPLY AND SUBTRACT (long BFP)	RRD				¤[7,9]				Db	Xi			Xo	Xu	Xx						B31F	19-38
MSDR	MULTIPLY AND SUBTRACT (long HFP)	RRD		HM		¤[7,9]				Da	EU	EO										B33F	18-19
MSE	MULTIPLY AND SUBTRACT (short HFP)	RXF		HM		¤[7,9]	A			Da	EU	EO								B_2		ED2F	18-19
MSEB	MULTIPLY AND SUBTRACT (short BFP)	RXF				¤[7,9]	A			Db	Xi			Xo	Xu	Xx				B_2		ED0F	19-38
MSEBR	MULTIPLY AND SUBTRACT (short BFP)	RRD				¤[7,9]				Db	Xi			Xo	Xu	Xx						B30F	19-38
MSER	MULTIPLY AND SUBTRACT (short HFP)	RRD		HM		¤[7,9]				Da	EU	EO										B32F	18-19
MSFI	MULTIPLY SINGLE IMMEDIATE (32)	RIL-a		GE																		C21	7-307
MSG	MULTIPLY SINGLE (64)	RXY-a		N			A													B_2		E30C	7-307
MSGC	MULTIPLY SINGLE (64)	RXY-a	C	MI2			A				IF									B_2		E383	7-307
MSGF	MULTIPLY SINGLE (64←32)	RXY-a		N			A													B_2		E31C	7-307
MSGFI	MULTIPLY SINGLE IMMEDIATE (64←32)	RIL-a		GE																		C20	7-307
MSGFR	MULTIPLY SINGLE (64←32)	RRE		N																		B91C	7-307
MSGR	MULTIPLY SINGLE (64)	RRE		N																		B90C	7-307
MSGRKC	MULTIPLY SINGLE (64)	RRF-a	C	MI2							IF											B9ED	7-307
MSR	MULTIPLY SINGLE (32)	RRE																				B252	7-307
MSRKC	MULTIPLY SINGLE (32)	RRF-a	C	MI2							IF											B9FD	7-307
MSTA	MODIFY STACKED STATE	RRE				¤[1]	A[1]*	SP		SE									ST			B247	10-61
MSY	MULTIPLY SINGLE (32)	RXY-a		LD			A													B_2		E351	7-307
MVC	MOVE (character)	SS-a				¤[9]	A												ST	B_1	B_2	D2	7-289
MVCDK	MOVE WITH DESTINATION KEY	SSE				Q	A							GM					ST	B_1	B_2	E50F	10-66

Figure B-2. Instructions Arranged by Mnemonic (Part 13 of 23)

Mne-monic	Name	Characteristics																Op-code	Page
MVCIN	MOVE INVERSE	SS-a			¤[9]	A								ST	B_1	B_2		E8	7-290
MVCK	MOVE WITH KEY	SS-d	C		Q	A								ST	B_1	B_2		D9	10-67
MVCL	MOVE LONG	RR	C		¤[9]	A	SP	II						ST	R_1	R_2		0E	7-290
MVCLE	MOVE LONG EXTENDED	RS-a	C		¤[9]	A	SP	IC						ST	R_1	R_3		A8	7-294
MVCLU	MOVE LONG UNICODE	RSY-a	C	E2	¤[9]	A	SP	IC						ST	R_1	R_3		EB8E	7-297
MVCOS	MOVE WITH OPTIONAL SPECIFICATIONS	SSF	C	MO	Q	A		SO			G0			ST	B†	B‡		C80	10-68
MVCP	MOVE TO PRIMARY	SS-d	C		Q	A		SO	¢					ST				DA	10-64
MVCS	MOVE TO SECONDARY	SS-d	C		Q	A		SO	¢					ST				DB	10-64
MVCSK	MOVE WITH SOURCE KEY	SSE			Q	A					GM			ST	B_1	B_2		E50E	10-71
MVGHI	MOVE (64←16)	SIL		GE		A								ST	B_1			E548	7-289
MVHHI	MOVE (16←16)	SIL		GE		A								ST	B_1			E544	7-289
MVHI	MOVE (32←16)	SIL		GE		A								ST	B_1			E54C	7-289
MVI	MOVE (immediate)	SI				A								ST	B_1			92	7-289
MVIY	MOVE (immediate)	SIY		LD		A								ST	B_1			EB52	7-289
MVN	MOVE NUMERICS	SS-a			¤[9]	A								ST	B_1	B_2		D1	7-301
MVO	MOVE WITH OFFSET	SS-b			¤[9]	A								ST	B_1	B_2		F1	7-302
MVPG	MOVE PAGE	RRE	C		Q	A	SP				G0			ST	R_1	R_2		B254	10-62
MVST	MOVE STRING	RRE	C		¤[9]	A	SP	IC			G0			ST	R_1	R_2		B255	7-301
MVZ	MOVE ZONES	SS-a			¤[9]	A								ST	B_1	B_2		D3	7-303
MXBR	MULTIPLY (extended BFP)	RRE			¤[7,9]		SP	Db	Xi		Xo	Xu	Xx					B34C	19-37
MXD	MULTIPLY (long to extended HFP)	RX-a			¤[7,9]	A	SP	Da	EU	EO						B_2		67	18-18
MXDB	MULTIPLY (long to extended BFP)	RXE			¤[7,9]	A	SP	Db	Xi							B_2		ED07	19-37
MXDBR	MULTIPLY (long to extended BFP)	RRE			¤[7,9]		SP	Db	Xi									B307	19-37
MXDR	MULTIPLY (long to extended HFP)	RR			¤[7,9]		SP	Da	EU	EO								27	18-17
MXR	MULTIPLY (extended HFP)	RR			¤[7,9]		SP	Da	EU	EO								26	18-17
MXTR	MULTIPLY (extended DFP)	RRF-a		TF	¤[7,9]		SP	Dt	Xi		Xo	Xu	Xx					B3D8	20-47
MXTRA	MULTIPLY (extended DFP)	RRF-a		F	¤[7,9]		SP	Dt	Xi		Xo	Xu	Xx Xq					B3D8	20-48
MY	MULTIPLY UNNORMALIZED (long to ext. HFP)	RXF		UE	¤[7,9]	A	SP	Da								B_2		ED3B	18-22
MYH	MULTIPLY UNNORM. (long to ext. high HFP)	RXF		UE	¤[7,9]	A		Da								B_2		ED3D	18-22
MYHR	MULTIPLY UNNORM. (long to ext. high HFP)	RRD		UE	¤[7,9]			Da										B33D	18-22
MYL	MULTIPLY UNNORM. (long to ext. low HFP)	RXF		UE	¤[7,9]	A		Da								B_2		ED39	18-22
MYLR	MULTIPLY UNNORM. (long to ext. low HFP)	RRD		UE	¤[7,9]			Da										B339	18-22
MYR	MULTIPLY UNNORMALIZED (long to ext. HFP)	RRD		UE	¤[7,9]		SP	Da										B33B	18-22
N	AND (32)	RX-a	C			A										B_2		54	7-33
NC	AND (character)	SS-a	C		¤[9]	A								ST	B_1	B_2		D4	7-34
NG	AND (64)	RXY-a	C	N		A										B_2		E380	7-33
NGR	AND (64)	RRE	C	N														B980	7-33
NGRK	AND (64)	RRF-a	C	DO														B9E4	7-33
NI	AND (immediate)	SI	C			A			£[2]					ST	B_1			94	7-33
NIAI	NEXT INSTRUCTION ACCESS INTENT	IE		EH														B2FA	7-308
NIHF	AND IMMEDIATE (high)	RIL-a	C	EI														C0A	7-34
NIHH	AND IMMEDIATE (high high)	RI-a	C	N														A54	7-34
NIHL	AND IMMEDIATE (high low)	RI-a	C	N														A55	7-34
NILF	AND IMMEDIATE (low)	RIL-a	C	EI														C0B	7-35
NILH	AND IMMEDIATE (low high)	RI-a	C	N														A56	7-35
NILL	AND IMMEDIATE (low low)	RI-a	C	N														A57	7-35
NIY	AND (immediate)	SIY	C	LD		A			£[2]					ST	B_1			EB54	7-34
NR	AND (32)	RR	C															14	7-33
NRK	AND (32)	RRF-a	C	DO														B9F4	7-33
NTSTG	NONTRANSACTIONAL STORE (64)	RXY-a		TX	¤[9]	A	SP							ST		B_2		E325	7-310
NY	AND (32)	RXY-a	C	LD		A										B_2		E354	7-33
O	OR (32)	RX-a	C			A										B_2		56	7-310
OC	OR (character)	SS-a	C		¤[9]	A								ST	B_1	B_2		D6	7-311
OG	OR (64)	RXY-a	C	N		A										B_2		E381	7-311
OGR	OR (64)	RRE	C	N														B981	7-310

Figure B-2. Instructions Arranged by Mnemonic (Part 14 of 23)

Mnemonic	Name	Characteristics																						Opcode	Page
OGRK	OR (64)	RRF-a	C	DO																				B9E6	7-310
OI	OR (immediate)	SI	C			A																ST	B_1	96	7-311
OIHF	OR IMMEDIATE (high)	RIL-a	C	EI																				C0C	7-312
OIHH	OR IMMEDIATE (high high)	RI-a	C	N																				A58	7-312
OIHL	OR IMMEDIATE (high low)	RI-a	C	N																				A59	7-312
OILF	OR IMMEDIATE (low)	RIL-a	C	EI																				C0D	7-312
OILH	OR IMMEDIATE (low high)	RI-a	C	N																				A5A	7-312
OILL	OR IMMEDIATE (low low)	RI-a	C	N																				A5B	7-312
OIY	OR (immediate)	SIY	C	LD		A																ST	B_1	EB56	7-311
OR	OR (32)	RR	C																					16	7-310
ORK	OR (32)	RRF-a	C	DO																				B9F6	7-310
OY	OR (32)	RXY-a	C	LD		A																	B_2	E356	7-310
PACK	PACK	SS-b			[9]	A																ST	B_1 B_2	F2	7-312
PALB	PURGE ALB	RRE			P							$												B248	10-116
PC	PROGRAM CALL	S			Q	A[1]*		Z[1]	T	¢		GM									B	ST		B218	10-90
PCC	PERFORM CRYPTOGRAPHIC COMPUTATION	RRE	C	M4	[5,9]	A	SP	IC				GM	I1									ST		B92C	7-315
PCKMO	PERFORM CRYPTOGRAPHIC KEY MGMT. OPERATIONS	RRE		M3	P	A	SP					GM										ST		B928	10-74
PFD	PREFETCH DATA	RXY-b		GE	[9,11]																		B_2	E336	7-359
PFDRL	PREFETCH DATA RELATIVE LONG	RIL-c		GE	[9,11]																			C62	7-359
PFMF	PERFORM FRAME MANAGEMENT FUNCTION	RRE		ED1	P	A[1]	SP	II		¢[3]														B9AF	10-77
PFPO	PERFORM FLOATING-POINT OPERATION	E		PF	[7-9]		SP	Da	Xi	X0		GM	Xu	Xx	Xq									010A	9-35
PGIN	PAGE IN	RRE	C	ES	P	A[1]				¢														B22E	10-72
PGOUT	PAGE OUT	RRE	C	ES	P	A[1]				¢														B22F	10-73
PKA	PACK ASCII	SS-f		E2	[9]	A	SP															ST	B_1 B_2	E9	7-313
PKU	PACK UNICODE	SS-f		E2	[9]	A	SP															ST	B_1 B_2	E1	7-314
PLO	PERFORM LOCKED OPERATION	SS-e	C		[1]	A	SP			$		GM										ST	FC	EE	7-331
POPCNT	POPULATION COUNT	RRE	C	PK																				B9E1	7-359
PPA	PERFORM PROCESSOR ASSIST	RRF-c		PA	[1]																			B2E8	7-345
PR	PROGRAM RETURN	E	L		[1]	A[1]*	SP	Z[4]	T	¢[2]											B	ST		0101	10-103
PRNO	PERFORM RANDOM NUMBER OPERATION	RRE	C	M5	[5,9]	A	SP	IC	Dg			GM	I1									ST	R_1 R_2	B93C	7-346
PT	PROGRAM TRANSFER	RRE			Q	A[1]*	SP	Z[2]	T	¢											B			B228	10-107
PTF	PERFORM TOPOLOGY FUNCTION	RRE	C	CT	P		SP																	B9A2	10-89
PTFF	PERFORM TIMING FACILITY FUNCTION	E	C	TS	Q	A	SP					GM										ST		0104	10-81
PTI	PROGRAM TRANSFER WITH INSTANCE	RRE		RA	Q	A[1]*	SP	Z[6]	T	¢											B			B99E	10-107
PTLB	PURGE TLB	S			P					$														B20D	10-116
QADTR	QUANTIZE (long DFP)	RRF-b		TF	[7,9]			Dt	Xi					Xx	Xq									B3F5	20-49
QAXTR	QUANTIZE (extended DFP)	RRF-b		TF	[7,9]		SP	Dt	Xi					Xx	Xq									B3FD	20-49
RCHP	RESET CHANNEL PATH	S	C		P																			B23B	14-9
RISBG	ROTATE THEN INSERT SELECTED BITS (64)	RIE-f	C	GE																				EC55	7-363
RISBGN	ROTATE THEN INSERT SELECTED BITS (64)	RIE-f		MI1																				EC59	7-363
RISBHG	ROTATE THEN INSERT SELECTED BITS HIGH (64)	RIE-f		HW																				EC5D	7-364
RISBLG	ROTATE THEN INSERT SELECTED BITS LOW (64)	RIE-f		HW																				EC51	7-365
RLL	ROTATE LEFT SINGLE LOGICAL (32)	RSY-a		N3																				EB1D	7-361
RLLG	ROTATE LEFT SINGLE LOGICAL (64)	RSY-a		N																				EB1C	7-361
RNSBG	ROTATE THEN AND SELECTED BITS (64)	RIE-f	C	GE																				EC54	7-361
ROSBG	ROTATE THEN OR SELECTED BITS (64)	RIE-f	C	GE																				EC56	7-362
RP	RESUME PROGRAM	S	L		Q	A	SP	WE	T												B		B_2	B277	10-117
RRBE	RESET REFERENCE BIT EXTENDED	RRE	C		P	A[1]*																		B22A	10-116
RRBM	RESET REFERENCE BITS MULTIPLE	RRE		RB	P	A[1]*																		B9AE	10-117
RRDTR	REROUND (long DFP)	RRF-b		TF	[7,9]			Dt	Xi					Xx	Xq									B3F7	20-52
RRXTR	REROUND (extended DFP)	RRF-b		TF	[7,9]		SP	Dt	Xi					Xx	Xq									B3FF	20-52
RSCH	RESUME SUBCHANNEL	S	C		P			OP		¢		GS												B238	14-10
RXSBG	ROTATE THEN EXCLUSIVE OR SELECT. BITS (64)	RIE-f	C	GE																				EC57	7-362

Figure B-2. Instructions Arranged by Mnemonic (Part 15 of 23)

Mne-monic	Name	Characteristics																			Op-code	Page
S	SUBTRACT (32)	RX-a	C			A			IF									B_2			5B	7-388
SAC	SET ADDRESS SPACE CONTROL	S				Q		SP	SW		¢										B219	10-120
SACF	SET ADDRESS SPACE CONTROL FAST	S				Q		SP	SW												B279	10-120
SAL	SET ADDRESS LIMIT	S				P			OP		¢	G1									B237	14-12
SAM24	SET ADDRESSING MODE (24)	E			N3	¤[3,9]		SP		T											010C	7-370
SAM31	SET ADDRESSING MODE (31)	E			N3	¤[3,9]		SP		T											010D	7-370
SAM64	SET ADDRESSING MODE (64)	E			N	¤[3,9]				T											010E	7-370
SAR	SET ACCESS	RRE				¤[6]												U_1			B24E	7-370
SCHM	SET CHANNEL MONITOR	S				P			OP		¢	GM									B23C	14-13
SCK	SET CLOCK	S	C			P	A	SP										B_2			B204	10-121
SCKC	SET CLOCK COMPARATOR	S				P	A	SP										B_2			B206	10-122
SCKPF	SET CLOCK PROGRAMMABLE FIELD	E				P		SP				G0									0107	10-123
SD	SUBTRACT NORMALIZED (long HFP)	RX-a	C			¤[7,9]	A		Da		EU	EO		LS				B_2			6B	18-24
SDB	SUBTRACT (long BFP)	RXE	C			¤[7,9]	A		Db	Xi			Xo	Xu	Xx			B_2			ED1B	19-40
SDBR	SUBTRACT (long BFP)	RRE	C			¤[7,9]			Db	Xi			Xo	Xu	Xx						B31B	19-40
SDR	SUBTRACT NORMALIZED (long HFP)	RR	C			¤[7,9]			Da		EU	EO		LS							2B	18-24
SDTR	SUBTRACT (long DFP)	RRF-a	C	TF		¤[7,9]			Dt	Xi			Xo	Xu	Xx						B3D3	20-55
SDTRA	SUBTRACT (long DFP)	RRF-a	C	F		¤[7,9]			Dt	Xi			Xo	Xu	Xx	Xq					B3D3	20-55
SE	SUBTRACT NORMALIZED (short HFP)	RX-a	C			¤[7,9]	A		Da		EU	EO		LS				B_2			7B	18-24
SEB	SUBTRACT (short BFP)	RXE	C			¤[7,9]	A		Db	Xi			Xo	Xu	Xx			B_2			ED0B	19-40
SEBR	SUBTRACT (short BFP)	RRE	C			¤[7,9]			Db	Xi			Xo	Xu	Xx						B30B	19-40
SER	SUBTRACT NORMALIZED (short HFP)	RR	C			¤[7,9]			Da		EU	EO		LS							3B	18-24
SFASR	SET FPC AND SIGNAL	RRE		XF		¤[7,9]		SP	Dt	Xg											B385	9-48
SFPC	SET FPC	RRE				¤[7,9]		SP	Db												B384	9-47
SG	SUBTRACT (64)	RXY-a	C	N			A			IF								B_2			E309	7-388
SGF	SUBTRACT (64←32)	RXY-a	C	N			A			IF								B_2			E319	7-388
SGFR	SUBTRACT (64←32)	RRE	C	N						IF											B919	7-387
SGH	SUBTRACT HALFWORD (64←16)	RXY-a	C	MI2			A			IF								B_2			E339	7-388
SGR	SUBTRACT (64)	RRE	C	N						IF											B909	7-387
SGRK	SUBTRACT (64)	RRF-a	C	DO						IF											B9E9	7-388
SH	SUBTRACT HALFWORD (32←16)	RX-a	C				A			IF								B_2			4B	7-388
SHHHR	SUBTRACT HIGH (32)	RRF-a	C	HW						IF											B9C9	7-389
SHHLR	SUBTRACT HIGH (32)	RRF-a	C	HW						IF											B9D9	7-389
SHY	SUBTRACT HALFWORD (32←16)	RXY-a	C	LD			A			IF								B_2			E37B	7-388
SIGP	SIGNAL PROCESSOR	RS-a	C			P					$										AE	10-133
SL	SUBTRACT LOGICAL (32)	RX-a	C				A											B_2			5F	7-390
SLA	SHIFT LEFT SINGLE (32)	RS-a	C							IF											8B	7-372
SLAG	SHIFT LEFT SINGLE (64)	RSY-a	C	N						IF											EB0B	7-372
SLAK	SHIFT LEFT SINGLE (32)	RSY-a	C	DO						IF											EBDD	7-372
SLB	SUBTRACT LOGICAL WITH BORROW (32)	RXY-a	C	N3			A											B_2			E399	7-391
SLBG	SUBTRACT LOGICAL WITH BORROW (64)	RXY-a	C	N			A											B_2			E389	7-391
SLBGR	SUBTRACT LOGICAL WITH BORROW (64)	RRE	C	N																	B989	7-391
SLBR	SUBTRACT LOGICAL WITH BORROW (32)	RRE	C	N3																	B999	7-391
SLDA	SHIFT LEFT DOUBLE (64)	RS-a	C					SP		IF											8F	7-371
SLDL	SHIFT LEFT DOUBLE LOGICAL (64)	RS-a						SP													8D	7-372
SLDT	SHIFT SIGNIFICAND LEFT (long DFP)	RXF		TF		¤[7,9]			Dt												ED40	20-54
SLFI	SUBTRACT LOGICAL IMMEDIATE (32)	RIL-a	C	EI																	C25	7-390
SLG	SUBTRACT LOGICAL (64)	RXY-a	C	N			A											B_2			E30B	7-390
SLGF	SUBTRACT LOGICAL (64←32)	RXY-a	C	N			A											B_2			E31B	7-390
SLGFI	SUBTRACT LOGICAL IMMEDIATE (64←32)	RIL-a	C	EI																	C24	7-390
SLGFR	SUBTRACT LOGICAL (64←32)	RRE	C	N																	B91B	7-389
SLGR	SUBTRACT LOGICAL (64)	RRE	C	N																	B90B	7-389
SLGRK	SUBTRACT LOGICAL (64)	RRF-a	C	DO																	B9EB	7-389
SLHHHR	SUBTRACT LOGICAL HIGH (32)	RRF-a	C	HW																	B9CB	7-391
SLHHLR	SUBTRACT LOGICAL HIGH (32)	RRF-a	C	HW																	B9DB	7-391

Figure B-2. Instructions Arranged by Mnemonic (Part 16 of 23)

Mnemonic	Name	Characteristics															Op-code	Page
SLL	SHIFT LEFT SINGLE LOGICAL (32)	RS-a															89	7-373
SLLG	SHIFT LEFT SINGLE LOGICAL (64)	RSY-a		N													EB0D	7-373
SLLK	SHIFT LEFT SINGLE LOGICAL (32)	RSY-a		DO													EBDF	7-373
SLR	SUBTRACT LOGICAL (32)	RR	C														1F	7-389
SLRK	SUBTRACT LOGICAL (32)	RRF-a	C	DO													B9FB	7-389
SLXT	SHIFT SIGNIFICAND LEFT (extended DFP)	RXF		TF	¤[7,9]		SP	Dt									ED48	20-54
SLY	SUBTRACT LOGICAL (32)	RXY-a	C	LD		A										B_2	E35F	7-390
SP	SUBTRACT DECIMAL	SS-b	C		¤[9]	A		Dg	DF					ST	B_1	B_2	FB	8-13
SPKA	SET PSW KEY FROM ADDRESS	S			Q												B20A	10-124
SPM	SET PROGRAM MASK	RR	L														04	7-371
SPT	SET CPU TIMER	S			P	A	SP									B_2	B208	10-123
SPX	SET PREFIX	S			P	A	SP			$						B_2	B210	10-123
SQD	SQUARE ROOT (long HFP)	RXE			¤[7,9]	A		Da			SQ					B_2	ED35	18-23
SQDB	SQUARE ROOT (long BFP)	RXE			¤[7,9]	A		Db	Xi			Xx				B_2	ED15	19-40
SQDBR	SQUARE ROOT (long BFP)	RRE			¤[7,9]			Db	Xi			Xx					B315	19-40
SQDR	SQUARE ROOT (long HFP)	RRE			¤[7,9]			Da			SQ						B244	18-23
SQE	SQUARE ROOT (short HFP)	RXE			¤[7,9]	A		Da			SQ					B_2	ED34	18-23
SQEB	SQUARE ROOT (short BFP)	RXE			¤[7,9]	A		Db	Xi			Xx				B_2	ED14	19-40
SQEBR	SQUARE ROOT (short BFP)	RRE			¤[7,9]			Db	Xi			Xx					B314	19-40
SQER	SQUARE ROOT (short HFP)	RRE			¤[7,9]			Da			SQ						B245	18-23
SQXBR	SQUARE ROOT (extended BFP)	RRE			¤[7,9]		SP	Db	Xi			Xx					B316	19-40
SQXR	SQUARE ROOT (extended HFP)	RRE			¤[7,9]		SP	Da			SQ						B336	18-23
SR	SUBTRACT (32)	RR	C						IF								1B	7-387
SRA	SHIFT RIGHT SINGLE (32)	RS-a	C														8A	7-375
SRAG	SHIFT RIGHT SINGLE (64)	RSY-a	C	N													EB0A	7-375
SRAK	SHIFT RIGHT SINGLE (32)	RSY-a	C	DO													EBDC	7-375
SRDA	SHIFT RIGHT DOUBLE (64)	RS-a	C				SP										8E	7-374
SRDL	SHIFT RIGHT DOUBLE LOGICAL (64)	RS-a					SP										8C	7-374
SRDT	SHIFT SIGNIFICAND RIGHT (long DFP)	RXF		TF	¤[7,9]			Dt									ED41	20-54
SRK	SUBTRACT (32)	RRF-a	C	DO					IF								B9F9	7-387
SRL	SHIFT RIGHT SINGLE LOGICAL (32)	RS-a															88	7-376
SRLG	SHIFT RIGHT SINGLE LOGICAL (64)	RSY-a		N													EB0C	7-376
SRLK	SHIFT RIGHT SINGLE LOGICAL (32)	RSY-a		DO													EBDE	7-376
SRNM	SET BFP ROUNDING MODE (2 bit)	S			¤[7,9]			Db									B299	9-47
SRNMB	SET BFP ROUNDING MODE (3 bit)	S		F	¤[7,9]		SP	Db									B2B8	9-47
SRNMT	SET DFP ROUNDING MODE	S		TR	¤[7,9]			Dt									B2B9	9-47
SRP	SHIFT AND ROUND DECIMAL	SS-c	C		¤[9]	A		Dg	DF					ST	B_1	B_2	F0	8-12
SRST	SEARCH STRING	RRE	C		¤[9]	A	SP	IC			G0					R_2	B25E	7-366
SRSTU	SEARCH STRING UNICODE	RRE	C	E3	¤[9]	A	SP	IC			G0				R_1	R_2	B9BE	7-368
SRXT	SHIFT SIGNIFICAND RIGHT (extended DFP)	RXF		TF	¤[7,9]		SP	Dt									ED49	20-54
SSAIR	SET SECONDARY ASN WITH INSTANCE	RRE		RA	¤[1]	A[1]*		Z[7]	T	¢							B99F	10-125
SSAR	SET SECONDARY ASN	RRE			¤[1]	A[1]*		Z[3]	T	¢							B225	10-125
SSCH	START SUBCHANNEL	S	C		P	A	SP	OP		¢	GS					B_2	B233	14-15
SSKE	SET STORAGE KEY EXTENDED	RRF-c	C[1]		P	A[1]*		II		¢							B22B	10-130
SSM	SET SYSTEM MASK	SI			P	A	SP	SO								B_2	80	10-133
ST	STORE (32)	RX-a				A								ST		B_2	50	7-376
STAM	STORE ACCESS MULTIPLE	RS-a				A	SP							ST		UB	9B	7-377
STAMY	STORE ACCESS MULTIPLE	RSY-a		LD		A	SP							ST		UB	EB9B	7-377
STAP	STORE CPU ADDRESS	S			P	A	SP							ST		B_2	B212	10-136
STC	STORE CHARACTER	RX-a				A								ST		B_2	42	7-378
STCH	STORE CHARACTER HIGH (8)	RXY-a		HW		A								ST		B_2	E3C3	7-378
STCK	STORE CLOCK	S	C		¤[8,9]	A				$				ST		B_2	B205	7-379
STCKC	STORE CLOCK COMPARATOR	S			P	A	SP							ST		B_2	B207	10-135
STCKE	STORE CLOCK EXTENDED	S	C		¤[8,9]	A				$				ST		B_2	B278	7-380
STCKF	STORE CLOCK FAST	S	C	SC	¤[8,9]	A								ST		B_2	B27C	7-379

Figure B-2. Instructions Arranged by Mnemonic (Part 17 of 23)

Mnemonic	Name	Characteristics													Opcode	Page
STCM	STORE CHARACTERS UNDER MASK (low)	RS-b				A						ST		B_2	BE	7-378
STCMH	STORE CHARACTERS UNDER MASK (high)	RSY-b		N	¤[9,11]	A						ST		B_2	EB2C	7-378
STCMY	STORE CHARACTERS UNDER MASK (low)	RSY-b		LD		A						ST		B_2	EB2D	7-378
STCPS	STORE CHANNEL PATH STATUS	S			P	A	SP		¢			ST		B_2	B23A	14-16
STCRW	STORE CHANNEL REPORT WORD	S	C		P	A	SP		¢			ST		B_2	B239	14-17
STCTG	STORE CONTROL (64)	RSY-a		N	P	A	SP					ST		B_2	EB25	10-135
STCTL	STORE CONTROL (32)	RS-a			P	A	SP					ST		B_2	B6	10-135
STCY	STORE CHARACTER	RXY-a		LD		A						ST		B_2	E372	7-378
STD	STORE (long)	RX-a			¤[7,9]	A		Da				ST		B_2	60	9-48
STDY	STORE (long)	RXY-a		LD	¤[7,9]	A		Da				ST		B_2	ED67	9-49
STE	STORE (short)	RX-a			¤[7,9]	A		Da				ST		B_2	70	9-48
STEY	STORE (short)	RXY-a		LD	¤[7,9]	A		Da				ST		B_2	ED66	9-49
STFH	STORE HIGH (32)	RXY-a		HW		A						ST		B_2	E3CB	7-384
STFL	STORE FACILITY LIST	S		N3	P										B2B1	10-138
STFLE	STORE FACILITY LIST EXTENDED	S	C	FL	¤[1]	A	SP			G0		ST		B_2	B2B0	7-382
STFPC	STORE FPC	S			¤[7,9]	A		Db				ST		B_2	B29C	9-49
STG	STORE (64)	RXY-a		N		A						ST		B_2	E324	7-377
STGRL	STORE RELATIVE LONG (64)	RIL-b		GE		A*	SP					ST			C4B	7-377
STGSC	STORE GUARDED STORAGE CONTROLS	RXY-a		GF	¤[1]	A		SO				ST	B_2		E349	7-383
STH	STORE HALFWORD (16)	RX-a				A						ST		B_2	40	7-383
STHH	STORE HALFWORD HIGH (16)	RXY-a		HW		A						ST		B_2	E3C7	7-384
STHRL	STORE HALFWORD RELATIVE LONG (16)	RIL-b		GE		A*						ST			C47	7-384
STHY	STORE HALFWORD (16)	RXY-a		LD		A						ST		B_2	E370	7-384
STIDP	STORE CPU ID	S			P	A	SP					ST		B_2	B202	10-136
STM	STORE MULTIPLE (32)	RS-a				A						ST		B_2	90	7-385
STMG	STORE MULTIPLE (64)	RSY-a		N		A						ST		B_2	EB24	7-385
STMH	STORE MULTIPLE HIGH (32)	RSY-a		N		A						ST		B_2	EB26	7-385
STMY	STORE MULTIPLE (32)	RSY-a		LD		A						ST		B_2	EB90	7-385
STNSM	STORE THEN AND SYSTEM MASK	SI			P	A						ST	B_1		AC	10-164
STOC	STORE ON CONDITION (32)	RSY-b		L1		A						ST		B_2	EBF3	7-385
STOCFH	STORE HIGH ON CONDITION	RSY-b		L2		A						ST		B_2	EBE1	7-386
STOCG	STORE ON CONDITION (64)	RSY-b		L1		A						ST		B_2	EBE3	7-385
STOSM	STORE THEN OR SYSTEM MASK	SI			P	A	SP					ST	B_1		AD	10-164
STPQ	STORE PAIR TO QUADWORD	RXY-a		N	¤[9]	A	SP					ST		B_2	E38E	7-386
STPT	STORE CPU TIMER	S			P	A	SP					ST		B_2	B209	10-138
STPX	STORE PREFIX	S			P	A	SP					ST		B_2	B211	10-139
STRAG	STORE REAL ADDRESS	SSE		N	P	A[1]	SP					ST	B_1	BP	E502	10-139
STRL	STORE RELATIVE LONG (32)	RIL-b		GE		A*	SP					ST			C4F	7-377
STRV	STORE REVERSED (32)	RXY-a		N3		A						ST		B_2	E33E	7-387
STRVG	STORE REVERSED (64)	RXY-a		N		A						ST		B_2	E32F	7-387
STRVH	STORE REVERSED (16)	RXY-a		N3		A						ST		B_2	E33F	7-387
STSCH	STORE SUBCHANNEL	S	C		P	A	SP	OP	¢	GS		ST		B_2	B234	14-18
STSI	STORE SYSTEM INFORMATION	S	C		P	A	SP			GM		ST		B_2	B27D	10-140
STURA	STORE USING REAL ADDRESS (32)	RRE			P	A[1]	SP					SU			B246	10-165
STURG	STORE USING REAL ADDRESS (64)	RRE		N	P	A[1]	SP					SU			B925	10-165
STY	STORE (32)	RXY-a		LD		A						ST		B_2	E350	7-377
SU	SUBTRACT UNNORMALIZED (short HFP)	RX-a	C		¤[7,9]	A		Da		EO	LS			B_2	7F	18-25
SUR	SUBTRACT UNNORMALIZED (short HFP)	RR	C		¤[7,9]			Da		EO	LS				3F	18-25
SVC	SUPERVISOR CALL	I			¤[1]				¢						0A	7-392
SW	SUBTRACT UNNORMALIZED (long HFP)	RX-a	C		¤[7,9]	A		Da		EO	LS			B_2	6F	18-25
SWR	SUBTRACT UNNORMALIZED (long HFP)	RR	C		¤[7,9]			Da		EO	LS				2F	18-25
SXBR	SUBTRACT (extended BFP)	RRE	C		¤[7,9]		SP	Db	Xi	Xo	Xu Xx				B34B	19-40
SXR	SUBTRACT NORMALIZED (extended HFP)	RR	C		¤[7,9]		SP	Da	EU EO		LS				37	18-24
SXTR	SUBTRACT (extended DFP)	RRF-a	C	TF	¤[7,9]		SP	Dt	Xi	Xo	Xu Xx				B3DB	20-55
SXTRA	SUBTRACT (extended DFP)	RRF-a	C	F	¤[7,9]		SP	Dt	Xi	Xo	Xu Xx Xq				B3DB	20-55

Figure B-2. Instructions Arranged by Mnemonic (Part 18 of 23)

Mnemonic	Name	Characteristics															Op-code	Page
SY	SUBTRACT (32)	RXY-a	C	LD		A		IF							B_2		E35B	7-388
TABORT	TRANSACTION ABORT	S		TX	¤[9]		SP	SO	$	EX			ST				B2FC	7-394
TAM	TEST ADDRESSING MODE	E	C	N3	¤[9]												010B	7-392
TAR	TEST ACCESS	RRE	C		¤[1]	A[1]*									U_1		B24C	10-165
TB	TEST BLOCK	RRE	C		P	A[1]*		II	$	G0							B22C	10-167
TBDR	CONVERT HFP TO BFP (long)	RRF-e	C		¤[7,9]		SP	Da									B351	9-28
TBEDR	CONVERT HFP TO BFP (long to short)	RRF-e	C		¤[7,9]		SP	Da									B350	9-28
TBEGIN	TRANSACTION BEGIN (nonconstrained)	SIL	C	TX	¤[9]	A	SP	SO	$	EX			ST				E560	7-395
TBEGINC	TRANSACTION BEGIN (constrained)	SIL	C	CX	¤[9]		SP	SO	$	EX							E561	7-399
TCDB	TEST DATA CLASS (long BFP)	RXE	C		¤[7,9]			Db									ED11	19-41
TCEB	TEST DATA CLASS (short BFP)	RXE	C		¤[7,9]			Db									ED10	19-41
TCXB	TEST DATA CLASS (extended BFP)	RXE	C		¤[7,9]		SP	Db									ED12	19-41
TDCDT	TEST DATA CLASS (long DFP)	RXE	C	TF	¤[7,9]			Dt									ED54	20-56
TDCET	TEST DATA CLASS (short DFP)	RXE	C	TF	¤[7,9]			Dt									ED50	20-56
TDCXT	TEST DATA CLASS (extended DFP)	RXE	C	TF	¤[7,9]		SP	Dt									ED58	20-56
TDGDT	TEST DATA GROUP (long DFP)	RXE	C	TF	¤[7,9]			Dt									ED55	20-57
TDGET	TEST DATA GROUP (short DFP)	RXE	C	TF	¤[7,9]			Dt									ED51	20-57
TDGXT	TEST DATA GROUP (extended DFP)	RXE	C	TF	¤[7,9]		SP	Dt									ED59	20-57
TEND	TRANSACTION END	S	C	TX				SO	$	EX							B2F8	7-401
THDER	CONVERT BFP TO HFP (short to long)	RRE	C		¤[7,9]			Da									B358	9-27
THDR	CONVERT BFP TO HFP (long)	RRE	C		¤[7,9]			Da									B359	9-27
TM	TEST UNDER MASK	SI	C			A									B_1		91	7-393
TMH	TEST UNDER MASK HIGH	RI-a	C														A70	7-393
TMHH	TEST UNDER MASK (high high)	RI-a	C	N													A72	7-393
TMHL	TEST UNDER MASK (high low)	RI-a	C	N													A73	7-393
TML	TEST UNDER MASK LOW	RI-a	C														A71	7-393
TMLH	TEST UNDER MASK (low high)	RI-a	C	N													A70	7-393
TMLL	TEST UNDER MASK (low low)	RI-a	C	N													A71	7-393
TMY	TEST UNDER MASK	SIY	C	LD		A									B_1		EB51	7-393
TP	TEST DECIMAL	RSL-a	C	E2	¤[9]	A									B_1	B_2	EBC0	8-14
TPEI	TEST PENDING EXTERNAL INTERRUPTION	RRE	C	TE	P												B9A1	10-169
TPI	TEST PENDING INTERRUPTION	S	C		P	A[1]*	SP		¢				ST			B_2	B236	14-19
TPROT	TEST PROTECTION	SSE	C		P	A[1]*									B_1		E501	10-170
TR	TRANSLATE	SS-a			¤[9]	A							ST		B_1	B_2	DC	7-402
TRACE	TRACE (32)	RS-a			P	A	SP		T	¢						B_2	99	10-173
TRACG	TRACE (64)	RSY-a		N	P	A	SP		T	¢						B_2	EB0F	10-173
TRAP2	TRAP	E			¤[1]	A*		SO	T				B	ST			01FF	10-174
TRAP4	TRAP	S			¤[1]	A*		SO	T				B	ST			B2FF	10-174
TRE	TRANSLATE EXTENDED	RRE	C		¤[9]	A	SP	IC						ST	R_1	R_2	B2A5	7-409
TROO	TRANSLATE ONE TO ONE	RRF-c	C	E2	¤[9]	A	SP	IC			GM			ST	RM	R_2	B993	7-411
TROT	TRANSLATE ONE TO TWO	RRF-c	C	E2	¤[9]	A	SP	IC			GM			ST	RM	R_2	B992	7-411
TRT	TRANSLATE AND TEST	SS-a	C		¤[9]	A					GM				B_1	B_2	DD	7-403
TRTE	TRANSLATE AND TEST EXTENDED	RRF-c	C	PE	¤[9]	A	SP	IC						ST	RM		B9BF	7-403
TRTO	TRANSLATE TWO TO ONE	RRF-c	C	E2	¤[9]	A	SP	IC			GM			ST	RM	R_2	B991	7-411
TRTR	TRANSLATE AND TEST REVERSE	SS-a	C	E3	¤[9]	A					GM				B_1	B_2	D0	7-408
TRTRE	TRANSLATE AND TEST REVERSE EXTENDED	RRF-c	C	PE	¤[9]	A	SP	IC						ST	RM		B9BD	7-404
TRTT	TRANSLATE TWO TO TWO	RRF-c	C	E2	¤[9]	A	SP	IC			GM			ST	RM	R_2	B990	7-411
TS	TEST AND SET	SI	C		¤[9]	A			$					ST		B_2	93	7-392
TSCH	TEST SUBCHANNEL	S	C		P	A	SP	OP	¢	GS				ST		B_2	B235	14-21
UNPK	UNPACK	SS-b			¤[9]	A								ST	B_1	B_2	F3	7-416
UNPKA	UNPACK ASCII	SS-a	C	E2	¤[9]	A	SP							ST	B_1	B_2	EA	7-417
UNPKU	UNPACK UNICODE	SS-a	C	E2	¤[9]	A	SP							ST	B_1	B_2	E2	7-418
UPT	UPDATE TREE	E	C		¤[9]	A	SP	II			GM	I4		ST			0102	7-419
VA	VECTOR ADD	VRR-c		VF	¤[7,9]		SP	Dv									E7F3	22-3
VAC	VECTOR ADD WITH CARRY	VRR-d		VF	¤[7,9]		SP	Dv									E7BB	22-4

Figure B-2. Instructions Arranged by Mnemonic (Part 19 of 23)

Mnemonic	Name	Characteristics																			Opcode	Page
VACC	VECTOR ADD COMPUTE CARRY	VRR-c		VF		$\alpha^{7,9}$	SP	Dv													E7F1	22-4
VACCC	VECTOR ADD WITH CARRY COMPUTE CARRY	VRR-d		VF		$\alpha^{7,9}$	SP	Dv													E7B9	22-5
VAP	VECTOR ADD DECIMAL	VRI-f	C*	VD		$\alpha^{7,9}$	SP	Dv	Dg	DF											E671	25-3
VAVG	VECTOR AVERAGE	VRR-c		VF		$\alpha^{7,9}$	SP	Dv													E7F2	22-6
VAVGL	VECTOR AVERAGE LOGICAL	VRR-c		VF		$\alpha^{7,9}$	SP	Dv													E7F0	22-6
VBPERM	VECTOR BIT PERMUTE	VRR-c		V1		$\alpha^{7,9}$		Dv													E785	21-4
VCDG	VECTOR FP CONVERT FROM FIXED 64-BIT	VRR-a		VF		$\alpha^{7,9}$	SP	Dv										Xx			E7C3	24-15
VCDLG	VECTOR FP CONVERT FROM LOGICAL 64-BIT	VRR-a		VF		$\alpha^{7,9}$	SP	Dv										Xx			E7C1	24-16
VCEQ	VECTOR COMPARE EQUAL	VRR-b	C*	VF		$\alpha^{7,9}$	SP	Dv													E7F8	22-7
VCGD	VECTOR FP CONVERT TO FIXED 64-BIT	VRR-a		VF		$\alpha^{7,9}$	SP	Dv	Xi									Xx			E7C2	24-17
VCH	VECTOR COMPARE HIGH	VRR-b	C*	VF		$\alpha^{7,9}$	SP	Dv													E7FB	22-8
VCHL	VECTOR COMPARE HIGH LOGICAL	VRR-b	C*	VF		$\alpha^{7,9}$	SP	Dv													E7F9	22-9
VCKSM	VECTOR CHECKSUM	VRR-c		VF		$\alpha^{7,9}$		Dv													E766	22-6
VCLGD	VECTOR FP CONVERT TO LOGICAL 64-BIT	VRR-a		VF		$\alpha^{7,9}$	SP	Dv	Xi									Xx			E7C0	24-19
VCLZ	VECTOR COUNT LEADING ZEROS	VRR-a		VF		$\alpha^{7,9}$	SP	Dv													E753	22-10
VCP	VECTOR COMPARE DECIMAL	VRR-h	C	VD		$\alpha^{7,9}$		Dv	Dg												E677	25-4
VCTZ	VECTOR COUNT TRAILING ZEROS	VRR-a		VF		$\alpha^{7,9}$	SP	Dv													E752	22-10
VCVB	VECTOR CONVERT TO BINARY	VRR-i	C*	VD		$\alpha^{7,9}$		Dv	Dg			IF									E650	25-5
VCVBG	VECTOR CONVERT TO BINARY	VRR-i	C*	VD		$\alpha^{7,9}$		Dv	Dg			IF									E652	25-5
VCVD	VECTOR CONVERT TO DECIMAL	VRI-i	C*	VD		$\alpha^{7,9}$	SP	Dv			DF										E658	25-6
VCVDG	VECTOR CONVERT TO DECIMAL	VRI-i	C*	VD		$\alpha^{7,9}$	SP	Dv			DF										E65A	25-6
VDP	VECTOR DIVIDE DECIMAL	VRI-f	C*	VD		$\alpha^{7,9}$	SP	Dv	Dg	DF	DK										E67A	25-8
VEC	VECTOR ELEMENT COMPARE	VRR-a	C	VF		$\alpha^{7,9}$	SP	Dv													E7DB	22-7
VECL	VECTOR ELEMENT COMPARE LOGICAL	VRR-a	C	VF		$\alpha^{7,9}$	SP	Dv													E7D9	22-7
VERIM	VECTOR ELEMENT ROTATE AND INSERT UNDER MASK	VRI-d		VF		$\alpha^{7,9}$	SP	Dv													E772	22-22
VERLL	VECTOR ELEMENT ROTATE LEFT LOGICAL	VRS-a		VF		$\alpha^{7,9}$	SP	Dv													E733	22-21
VERLLV	VECTOR ELEMENT ROTATE LEFT LOGICAL	VRR-c		VF		$\alpha^{7,9}$	SP	Dv													E773	22-21
VESL	VECTOR ELEMENT SHIFT LEFT	VRS-a		VF		$\alpha^{7,9}$	SP	Dv													E730	22-23
VESLV	VECTOR ELEMENT SHIFT LEFT	VRR-c		VF		$\alpha^{7,9}$	SP	Dv													E770	22-23
VESRA	VECTOR ELEMENT SHIFT RIGHT ARITHMETIC	VRS-a		VF		$\alpha^{7,9}$	SP	Dv													E73A	22-23
VESRAV	VECTOR ELEMENT SHIFT RIGHT ARITHMETIC	VRR-c		VF		$\alpha^{7,9}$	SP	Dv													E77A	22-23
VESRL	VECTOR ELEMENT SHIFT RIGHT LOGICAL	VRS-a		VF		$\alpha^{7,9}$	SP	Dv													E738	22-24
VESRLV	VECTOR ELEMENT SHIFT RIGHT LOGICAL	VRR-c		VF		$\alpha^{7,9}$	SP	Dv													E778	22-24
VFA	VECTOR FP ADD	VRR-c		VF		$\alpha^{7,9}$	SP	Dv	Xi				Xo		Xu		Xx				E7E3	24-3
VFAE	VECTOR FIND ANY ELEMENT EQUAL	VRR-b	C*	VF		$\alpha^{7,9}$	SP	Dv													E782	23-1
VFCE	VECTOR FP COMPARE EQUAL	VRR-c	C*	VF		$\alpha^{7,9}$	SP	Dv	Xi												E7E8	24-9
VFCH	VECTOR FP COMPARE HIGH	VRR-c	C*	VF		$\alpha^{7,9}$	SP	Dv	Xi												E7EB	24-11
VFCHE	VECTOR FP COMPARE HIGH OR EQUAL	VRR-c	C*	VF		$\alpha^{7,9}$	SP	Dv	Xi												E7EA	24-13
VFD	VECTOR FP DIVIDE	VRR-c		VF		$\alpha^{7,9}$	SP	Dv	Xi		Xz		Xo		Xu		Xx				E7E5	24-21
VFEE	VECTOR FIND ELEMENT EQUAL	VRR-b	C*	VF		$\alpha^{7,9}$	SP	Dv													E780	23-3
VFENE	VECTOR FIND ELEMENT NOT EQUAL	VRR-b	C*	VF		$\alpha^{7,9}$	SP	Dv													E781	23-4
VFI	VECTOR LOAD FP INTEGER	VRR-a		VF		$\alpha^{7,9}$	SP	Dv	Xi									Xx			E7C7	24-23
VFLL	VECTOR FP LOAD LENGTHENED	VRR-a		VF		$\alpha^{7,9}$	SP	Dv	Xi												E7C4	24-25
VFLR	VECTOR FP LOAD ROUNDED	VRR-a		VF		$\alpha^{7,9}$	SP	Dv	Xi				Xo		Xu		Xx				E7C5	24-26
VFM	VECTOR FP MULTIPLY	VRR-c		VF		$\alpha^{7,9}$	SP	Dv	Xi				Xo		Xu		Xx				E7E7	24-39
VFMA	VECTOR FP MULTIPLY AND ADD	VRR-e		VF		$\alpha^{7,9}$	SP	Dv	Xi				Xo		Xu		Xx				E78F	24-41
VFMAX	VECTOR FP MAXIMUM	VRR-c		V1		$\alpha^{7,9}$	SP	Dv	Xi												E7EF	24-27
VFMIN	VECTOR FP MINIMUM	VRR-c		V1		$\alpha^{7,9}$	SP	Dv	Xi												E7EE	24-33
VFMS	VECTOR FP MULTIPLY AND SUBTRACT	VRR-e		VF		$\alpha^{7,9}$	SP	Dv	Xi				Xo		Xu		Xx				E78E	24-41
VFNMA	VECTOR FP NEGATIVE MULTIPLY AND ADD	VRR-e		V1		$\alpha^{7,9}$	SP	Dv	Xi				Xo		Xu		Xx				E79F	24-41
VFNMS	VECTOR FP NEGATIVE MULTIPLY AND SUBTRACT	VRR-e		V1		$\alpha^{7,9}$	SP	Dv	Xi				Xo		Xu		Xx				E79E	24-41
VFPSO	VECTOR FP PERFORM SIGN OPERATION	VRR-a		VF		$\alpha^{7,9}$	SP	Dv													E7CC	24-43
VFS	VECTOR FP SUBTRACT	VRR-c		VF		$\alpha^{7,9}$	SP	Dv	Xi				Xo		Xu		Xx				E7E2	24-45

Figure B-2. Instructions Arranged by Mnemonic (Part 20 of 23)

Mnemonic	Name	Characteristics										Opcode	Page
VFSQ	VECTOR FP SQUARE ROOT	VRR-a	VF	α[7,9]		SP	Dv	Xi	Xx			E7CE	24-44
VFTCI	VECTOR FP TEST DATA CLASS IMMEDIATE	VRI-e C	VF	α[7,9]		SP	Dv					E74A	24-46
VGBM	VECTOR GENERATE BYTE MASK	VRI-a	VF	α[7,9]			Dv					E744	21-5
VGEF	VECTOR GATHER ELEMENT (32)	VRV	VF	α[7,9]	A	SP	Dv				B_2	E713	21-4
VGEG	VECTOR GATHER ELEMENT (64)	VRV	VF	α[7,9]	A	SP	Dv				B_2	E712	21-4
VGFM	VECTOR GALOIS FIELD MULTIPLY SUM	VRR-c	VF	α[7,9]		SP	Dv					E7B4	22-11
VGFMA	VECTOR GALOIS FIELD MULTIPLY SUM AND ACCUMULATE	VRR-d	VF	α[7,9]		SP	Dv					E7BC	22-12
VGM	VECTOR GENERATE MASK	VRI-b	VF	α[7,9]		SP	Dv					E746	21-5
VISTR	VECTOR ISOLATE STRING	VRR-a C*	VF	α[7,9]		SP	Dv					E75C	23-5
VL	VECTOR LOAD	VRX	VF	α[7,9]	A		Dv				B_2	E706	21-6
VLBB	VECTOR LOAD TO BLOCK BOUNDARY	VRX	VF	α[7,9]	A	SP	Dv				B_2	E707	21-10
VLC	VECTOR LOAD COMPLEMENT	VRR-a	VF	α[7,9]		SP	Dv					E7DE	22-12
VLEB	VECTOR LOAD ELEMENT (8)	VRX	VF	α[7,9]	A	SP	Dv				B_2	E700	21-6
VLEF	VECTOR LOAD ELEMENT (32)	VRX	VF	α[7,9]	A	SP	Dv				B_2	E703	21-7
VLEG	VECTOR LOAD ELEMENT (64)	VRX	VF	α[7,9]	A	SP	Dv				B_2	E702	21-7
VLEH	VECTOR LOAD ELEMENT (16)	VRX	VF	α[7,9]	A	SP	Dv				B_2	E701	21-7
VLEIB	VECTOR LOAD ELEMENT IMMEDIATE (8)	VRI-a	VF	α[7,9]		SP	Dv					E740	21-7
VLEIF	VECTOR LOAD ELEMENT IMMEDIATE (32)	VRI-a	VF	α[7,9]		SP	Dv					E743	21-7
VLEIG	VECTOR LOAD ELEMENT IMMEDIATE (64)	VRI-a	VF	α[7,9]		SP	Dv					E742	21-7
VLEIH	VECTOR LOAD ELEMENT IMMEDIATE (16)	VRI-a	VF	α[7,9]		SP	Dv					E741	21-7
VLGV	VECTOR LOAD GR FROM VR ELEMENT	VRS-c	VF	α[7,9]		SP	Dv					E721	21-7
VLIP	VECTOR LOAD IMMEDIATE DECIMAL	VRI-h	VD	α[7,9]			Dv	Dg				E649	25-9
VLL	VECTOR LOAD WITH LENGTH	VRS-b	VF	α[7,9]	A		Dv				B_2	E737	21-11
VLLEZ	VECTOR LOAD LOGICAL ELEMENT AND ZERO	VRX	VF	α[7,9]	A	SP	Dv				B_2	E704	21-8
VLM	VECTOR LOAD MULTIPLE	VRS-a	VF	α[7,9]	A	SP	Dv				B_2	E736	21-9
VLP	VECTOR LOAD POSITIVE	VRR-a	VF	α[7,9]		SP	Dv					E7DF	22-12
VLR	VECTOR LOAD	VRR-a	VF	α[7,9]			Dv					E756	21-6
VLREP	VECTOR LOAD AND REPLICATE	VRX	VF	α[7,9]	A	SP	Dv				B_2	E705	21-6
VLRL	VECTOR LOAD RIGHTMOST WITH LENGTH	VSI	VD	α[7,9]	A	SP	Dv				B_2	E635	21-9
VLRLR	VECTOR LOAD RIGHTMOST WITH LENGTH	VRS-d	VD	α[7,9]	A		Dv				B_2	E637	21-9
VLVG	VECTOR LOAD VR ELEMENT FROM GR	VRS-b	VF	α[7,9]		SP	Dv					E722	21-11
VLVGP	VECTOR LOAD VR FROM GRS DISJOINT	VRR-f	VF	α[7,9]			Dv					E762	21-11
VMAE	VECTOR MULTIPLY AND ADD EVEN	VRR-d	VF	α[7,9]		SP	Dv					E7AE	22-15
VMAH	VECTOR MULTIPLY AND ADD HIGH	VRR-d	VF	α[7,9]		SP	Dv					E7AB	22-15
VMAL	VECTOR MULTIPLY AND ADD LOW	VRR-d	VF	α[7,9]		SP	Dv					E7AA	22-14
VMALE	VECTOR MULTIPLY AND ADD LOGICAL EVEN	VRR-d	VF	α[7,9]		SP	Dv					E7AC	22-15
VMALH	VECTOR MULTIPLY AND ADD LOGICAL HIGH	VRR-d	VF	α[7,9]		SP	Dv					E7A9	22-15
VMALO	VECTOR MULTIPLY AND ADD LOGICAL ODD	VRR-d	VF	α[7,9]		SP	Dv					E7AD	22-16
VMAO	VECTOR MULTIPLY AND ADD ODD	VRR-d	VF	α[7,9]		SP	Dv					E7AF	22-16
VME	VECTOR MULTIPLY EVEN	VRR-c	VF	α[7,9]		SP	Dv					E7A6	22-18
VMH	VECTOR MULTIPLY HIGH	VRR-c	VF	α[7,9]		SP	Dv					E7A3	22-16
VML	VECTOR MULTIPLY LOW	VRR-c	VF	α[7,9]		SP	Dv					E7A2	22-17
VMLE	VECTOR MULTIPLY LOGICAL EVEN	VRR-c	VF	α[7,9]		SP	Dv					E7A4	22-18
VMLH	VECTOR MULTIPLY LOGICAL HIGH	VRR-c	VF	α[7,9]		SP	Dv					E7A1	22-17
VMLO	VECTOR MULTIPLY LOGICAL ODD	VRR-c	VF	α[7,9]		SP	Dv					E7A5	22-18
VMN	VECTOR MINIMUM	VRR-c	VF	α[7,9]		SP	Dv					E7FE	22-13
VMNL	VECTOR MINIMUM LOGICAL	VRR-c	VF	α[7,9]		SP	Dv					E7FC	22-14
VMO	VECTOR MULTIPLY ODD	VRR-c	VF	α[7,9]		SP	Dv					E7A7	22-18
VMP	VECTOR MULTIPLY DECIMAL	VRI-f C*	VD	α[7,9]		SP	Dv	Dg	DF			E678	25-10
VMRH	VECTOR MERGE HIGH	VRR-c	VF	α[7,9]		SP	Dv					E761	21-12
VMRL	VECTOR MERGE LOW	VRR-c	VF	α[7,9]		SP	Dv					E760	21-12
VMSL	VECTOR MULTIPLY SUM LOGICAL	VRR-d	V1	α[7,9]		SP	Dv					E7B8	22-19
VMSP	VECTOR MULTIPLY AND SHIFT DECIMAL	VRI-f C*	VD	α[7,9]		SP	Dv	Dg	DF			E679	25-11
VMX	VECTOR MAXIMUM	VRR-c	VF	α[7,9]		SP	Dv					E7FF	22-13

Figure B-2. Instructions Arranged by Mnemonic (Part 21 of 23)

Mnemonic	Name	Characteristics										Opcode	Page
VMXL	VECTOR MAXIMUM LOGICAL	VRR-c	VF	$\sigma^{7,9}$		SP	Dv					E7FD	22-13
VN	VECTOR AND	VRR-c	VF	$\sigma^{7,9}$			Dv					E768	22-5
VNC	VECTOR AND WITH COMPLEMENT	VRR-c	VF	$\sigma^{7,9}$			Dv					E769	22-5
VNN	VECTOR NAND	VRR-c	V1	$\sigma^{7,9}$			DV					E76E	22-20
VNO	VECTOR NOR	VRR-c	VF	$\sigma^{7,9}$			Dv					E76B	22-20
VNX	VECTOR NOT EXCLUSIVE OR	VRR-c	V1	$\sigma^{7,9}$			Dv					E76C	22-20
VO	VECTOR OR	VRR-c	VF	$\sigma^{7,9}$			Dv					E76A	22-20
VOC	VECTOR OR WITH COMPLEMENT	VRR-c	V1	$\sigma^{7,9}$			Dv					E76F	22-21
VPDI	VECTOR PERMUTE DOUBLEWORD IMMEDIATE	VRR-c	VF	$\sigma^{7,9}$			Dv					E784	21-15
VPERM	VECTOR PERMUTE	VRR-e	VF	$\sigma^{7,9}$			Dv					E78C	21-15
VPK	VECTOR PACK	VRR-c	VF	$\sigma^{7,9}$		SP	Dv					E794	21-13
VPKLS	VECTOR PACK LOGICAL SATURATE	VRR-b	C* VF	$\sigma^{7,9}$		SP	Dv					E795	21-14
VPKS	VECTOR PACK SATURATE	VRR-b	C* VF	$\sigma^{7,9}$		SP	Dv					E797	21-13
VPKZ	VECTOR PACK ZONED	VSI	VD	$\sigma^{7,9}$	A	SP	Dv				B_2	E634	25-12
VPOPCT	VECTOR POPULATION COUNT	VRR-a	VF	$\sigma^{7,9}$		SP	Dv					E750	22-21
VPSOP	VECTOR PERFORM SIGN OPERATION DECIMAL	VRI-g	C* VD	$\sigma^{7,9}$		SP	Dv	Dg	DF			E65B	25-13
VREP	VECTOR REPLICATE	VRI-c	VF	$\sigma^{7,9}$		SP	Dv					E74D	21-15
VREPI	VECTOR REPLICATE IMMEDIATE	VRI-a	VF	$\sigma^{7,9}$		SP	Dv					E745	21-16
VRP	VECTOR REMAINDER DECIMAL	VRI-f	C* VD	$\sigma^{7,9}$		SP	Dv	Dg	DF	DK		E67B	25-14
VS	VECTOR SUBTRACT	VRR-c	VF	$\sigma^{7,9}$		SP	Dv					E7F7	22-26
VSBCBI	VECTOR SUBTRACT WITH BORROW COMPUTE BORROW INDICATION	VRR-d	VF	$\sigma^{7,9}$		SP	Dv					E7BD	22-28
VSBI	VECTOR SUBTRACT WITH BORROW INDICATION	VRR-d	VF	$\sigma^{7,9}$		SP	Dv					E7BF	22-27
VSCBI	VECTOR SUBTRACT COMPUTE BORROW INDICATION	VRR-c	VF	$\sigma^{7,9}$		SP	Dv					E7F5	22-27
VSCEF	VECTOR SCATTER ELEMENT (32)	VRV	VF	$\sigma^{7,9}$	A	SP	Dv				ST B_2	E71B	21-16
VSCEG	VECTOR SCATTER ELEMENT (64)	VRV	VF	$\sigma^{7,9}$	A	SP	Dv				ST B_2	E71A	21-16
VSDP	VECTOR SHIFT AND DIVIDE DECIMAL	VRI-f	C* VD	$\sigma^{7,9}$		SP	Dv	Dg	DF	DK		E67E	25-16
VSEG	VECTOR SIGN EXTEND TO DOUBLEWORD	VRR-a	VF	$\sigma^{7,9}$		SP	Dv					E75F	21-17
VSEL	VECTOR SELECT	VRR-e	VF	$\sigma^{7,9}$			Dv					E78D	21-17
VSL	VECTOR SHIFT LEFT	VRR-c	VF	$\sigma^{7,9}$			Dv					E774	22-25
VSLB	VECTOR SHIFT LEFT BY BYTE	VRR-c	VF	$\sigma^{7,9}$			Dv					E775	22-25
VSLDB	VECTOR SHIFT LEFT DOUBLE BY BYTE	VRI-d	VF	$\sigma^{7,9}$			Dv					E777	22-25
VSP	VECTOR SUBTRACT DECIMAL	VRI-f	C* VD	$\sigma^{7,9}$		SP	Dv	Dg	DF			E673	25-18
VSRA	VECTOR SHIFT RIGHT ARITHMETIC	VRR-c	VF	$\sigma^{7,9}$			Dv					E77E	22-25
VSRAB	VECTOR SHIFT RIGHT ARITHMETIC BY BYTE	VRR-c	VF	$\sigma^{7,9}$			Dv					E77F	22-26
VSRL	VECTOR SHIFT RIGHT LOGICAL	VRR-c	VF	$\sigma^{7,9}$			Dv					E77C	22-26
VSRLB	VECTOR SHIFT RIGHT LOGICAL BY BYTE	VRR-c	VF	$\sigma^{7,9}$			Dv					E77D	22-26
VSRP	VECTOR SHIFT AND ROUND DECIMAL	VRI-g	C* VD	$\sigma^{7,9}$		SP	Dv	Dg	DF			E659	25-17
VST	VECTOR STORE	VRX	VF	$\sigma^{7,9}$	A		Dv				ST B_2	E70E	21-18
VSTEB	VECTOR STORE ELEMENT (8)	VRX	VF	$\sigma^{7,9}$	A	SP	Dv				ST B_2	E708	21-18
VSTEF	VECTOR STORE ELEMENT (32)	VRX	VF	$\sigma^{7,9}$	A	SP	Dv				ST B_2	E70B	21-18
VSTEG	VECTOR STORE ELEMENT (64)	VRX	VF	$\sigma^{7,9}$	A	SP	Dv				ST B_2	E70A	21-18
VSTEH	VECTOR STORE ELEMENT (16)	VRX	VF	$\sigma^{7,9}$	A	SP	Dv				ST B_2	E709	21-18
VSTL	VECTOR STORE WITH LENGTH	VRS-b	VF	$\sigma^{7,9}$	A		Dv				ST B_2	E73F	21-20
VSTM	VECTOR STORE MULTIPLE	VRS-a	VF	$\sigma^{7,9}$	A	SP	Dv				ST B_2	E73E	21-19
VSTRC	VECTOR STRING RANGE COMPARE	VRR-d	C* VF	$\sigma^{7,9}$		SP	Dv					E78A	23-6
VSTRL	VECTOR STORE RIGHTMOST WITH LENGTH	VSI	VD	$\sigma^{7,9}$	A	SP	Dv				ST B_2	E63D	21-19
VSTRLR	VECTOR STORE RIGHTMOST WITH LENGTH	VRS-d	VD	$\sigma^{7,9}$	A		Dv				ST B_2	E63F	21-19
VSUM	VECTOR SUM ACROSS WORD	VRR-c	VF	$\sigma^{7,9}$		SP	Dv					E764	22-29
VSUMG	VECTOR SUM ACROSS DOUBLEWORD	VRR-c	VF	$\sigma^{7,9}$		SP	Dv					E765	22-28
VSUMQ	VECTOR SUM ACROSS QUADWORD	VRR-c	VF	$\sigma^{7,9}$		SP	Dv					E767	22-29
VTM	VECTOR TEST UNDER MASK	VRR-a	C VF	$\sigma^{7,9}$			Dv					E7D8	22-30
VTP	VECTOR TEST DECIMAL	VRR-g	C VD	$\sigma^{7,9}$			Dv					E65F	25-20

Figure B-2. Instructions Arranged by Mnemonic (Part 22 of 23)

Mnemonic	Name	Characteristics											Op-code	Page
VUPH	VECTOR UNPACK HIGH	VRR-a		VF	¤[7,9]		SP	Dv					E7D7	21-20
VUPKZ	VECTOR UNPACK ZONED	VSI		VD	¤[7,9]	A	SP	Dv			ST	B_2	E63C	25-20
VUPL	VECTOR UNPACK LOW	VRR-a		VF	¤[7,9]		SP	Dv					E7D6	21-22
VUPLH	VECTOR UNPACK LOGICAL HIGH	VRR-a		VF	¤[7,9]		SP	Dv					E7D5	21-21
VUPLL	VECTOR UNPACK LOGICAL LOW	VRR-a		VF	¤[7,9]		SP	Dv					E7D4	21-21
VX	VECTOR EXCLUSIVE OR	VRR-c		VF	¤[7,9]			Dv					E76D	22-11
WFC	VECTOR FP COMPARE SCALAR	VRR-a	C	VF	¤[7,9]		SP	Dv	Xi				E7CB	24-7
WFK	VECTOR FP COMPARE AND SIGNAL SCALAR	VRR-a	C	VF	¤[7,9]		SP	Dv	Xi				E7CA	24-8
X	EXCLUSIVE OR (32)	RX-a	C			A						B_2	57	7-254
XC	EXCLUSIVE OR (character)	SS-a	C		¤[9]	A					ST	B_1 B_2	D7	7-255
XG	EXCLUSIVE OR (64)	RXY-a	C	N		A						B_2	E382	7-254
XGR	EXCLUSIVE OR (64)	RRE	C	N									B982	7-254
XGRK	EXCLUSIVE OR (64)	RRF-a	C	DO									B9E7	7-254
XI	EXCLUSIVE OR (immediate)	SI	C			A					ST	B_1	97	7-255
XIHF	EXCLUSIVE OR IMMEDIATE (high)	RIL-a	C	EI									C06	7-256
XILF	EXCLUSIVE OR IMMEDIATE (low)	RIL-a	C	EI									C07	7-256
XIY	EXCLUSIVE OR (immediate)	SIY	C	LD		A					ST	B_1	EB57	7-255
XR	EXCLUSIVE OR (32)	RR	C										17	7-254
XRK	EXCLUSIVE OR (32)	RRF-a	C	DO									B9F7	7-254
XSCH	CANCEL SUBCHANNEL	S	C		P			OP	¢	GS			B276	14-3
XY	EXCLUSIVE OR (32)	RXY-a	C	LD		A						B_2	E357	7-254
ZAP	ZERO AND ADD	SS-b	C		¤[9]	A		Dg	DF		ST	B_1 B_2	F8	8-14

Figure B-2. Instructions Arranged by Mnemonic (Part 23 of 23)

Instructions Arranged by Operation Code

Op-code	Name	Mnemonic	Characteristics	Page
0101	PROGRAM RETURN	PR	E L ¤[1] A[1]* SP Z[4] T ¢[2] B ST	10-103
0102	UPDATE TREE	UPT	E C ¤[9] A SP II GM I4 ST	7-419
0104	PERFORM TIMING FACILITY FUNCTION	PTFF	E C TS Q A SP GM ST	10-81
0107	SET CLOCK PROGRAMMABLE FIELD	SCKPF	E P SP G0	10-123
010A	PERFORM FLOATING-POINT OPERATION	PFPO	E PF ¤[7-9] SP Da Xi X0 GM Xu Xx Xq	9-35
010B	TEST ADDRESSING MODE	TAM	E C N3 ¤[9]	7-392
010C	SET ADDRESSING MODE (24)	SAM24	E N3 ¤[3,9] SP T	7-370
010D	SET ADDRESSING MODE (31)	SAM31	E N3 ¤[3,9] SP T	7-370
010E	SET ADDRESSING MODE (64)	SAM64	E N ¤[3,9] T	7-370
01FF	TRAP	TRAP2	E ¤[1] A* SO T B ST	10-174
04	SET PROGRAM MASK	SPM	RR L	7-371
05	BRANCH AND LINK	BALR	RR ¤[2,9] T B	7-35
06	BRANCH ON COUNT (32)	BCTR	RR ¤[9] B	7-41
07	BRANCH ON CONDITION	BCR	RR ¤[9] ¢[1] B	7-40
0A	SUPERVISOR CALL	SVC	I ¤[1] ¢	7-392
0B	BRANCH AND SET MODE	BSM	RR ¤[3,9] T B	7-38
0C	BRANCH AND SAVE AND SET MODE	BASSM	RR ¤[2,3,9] T B	7-37
0D	BRANCH AND SAVE	BASR	RR ¤[2,9] T B	7-36
0E	MOVE LONG	MVCL	RR C ¤[9] A SP II ST R_1 R_2	7-290
0F	COMPARE LOGICAL LONG	CLCL	RR C ¤[9] A SP II R_1 R_2	7-158
10	LOAD POSITIVE (32)	LPR	RR C IF	7-287
11	LOAD NEGATIVE (32)	LNR	RR C	7-283
12	LOAD AND TEST (32)	LTR	RR C	7-270
13	LOAD COMPLEMENT (32)	LCR	RR C IF	7-272
14	AND (32)	NR	RR C	7-33
15	COMPARE LOGICAL (32)	CLR	RR C	7-152
16	OR (32)	OR	RR C	7-310
17	EXCLUSIVE OR (32)	XR	RR C	7-254
18	LOAD (32)	LR	RR	7-264
19	COMPARE (32)	CR	RR C	7-134
1A	ADD (32)	AR	RR C IF	7-26
1B	SUBTRACT (32)	SR	RR C IF	7-387
1C	MULTIPLY (64←32)	MR	RR SP	7-304
1D	DIVIDE (32←64)	DR	RR ¤[9] SP IK	7-252
1E	ADD LOGICAL (32)	ALR	RR C	7-30
1F	SUBTRACT LOGICAL (32)	SLR	RR C	7-389
20	LOAD POSITIVE (long HFP)	LPDR	RR C ¤[7,9] Da	18-16
21	LOAD NEGATIVE (long HFP)	LNDR	RR C ¤[7,9] Da	18-16
22	LOAD AND TEST (long HFP)	LTDR	RR C ¤[7,9] Da	18-13
23	LOAD COMPLEMENT (long HFP)	LCDR	RR C ¤[7,9] Da	18-14
24	HALVE (long HFP)	HDR	RR ¤[7,9] Da EU	18-13
25	LOAD ROUNDED (extended to long HFP)	LDXR	RR ¤[7,9] SP Da EO	18-17
25	LOAD ROUNDED (extended to long HFP)	LRDR	RR ¤[7,9] SP Da EO	18-17
26	MULTIPLY (extended HFP)	MXR	RR ¤[7,9] SP Da EU EO	18-17
27	MULTIPLY (long to extended HFP)	MXDR	RR ¤[7,9] SP Da EU EO	18-17
28	LOAD (long)	LDR	RR ¤[7,9] Da	9-31
29	COMPARE (long HFP)	CDR	RR C ¤[7,9] Da	18-10
2A	ADD NORMALIZED (long HFP)	ADR	RR C ¤[7,9] Da EU EO LS	18-8
2B	SUBTRACT NORMALIZED (long HFP)	SDR	RR C ¤[7,9] Da EU EO LS	18-24
2C	MULTIPLY (long HFP)	MDR	RR ¤[7,9] Da EU EO	18-17
2D	DIVIDE (long HFP)	DDR	RR ¤[7,9] Da EU EO FK	18-12

Figure B-3. Instructions Arranged by Opcode (Part 1 of 23)

Op-code	Name	Mne-monic	Characteristics														Page
2E	ADD UNNORMALIZED (long HFP)	AWR	RR	C	¤[7,9]			Da		EO		LS					18-9
2F	SUBTRACT UNNORMALIZED (long HFP)	SWR	RR	C	¤[7,9]			Da		EO		LS					18-25
30	LOAD POSITIVE (short HFP)	LPER	RR	C	¤[7,9]			Da									18-16
31	LOAD NEGATIVE (short HFP)	LNER	RR	C	¤[7,9]			Da									18-16
32	LOAD AND TEST (short HFP)	LTER	RR	C	¤[7,9]			Da									18-13
33	LOAD COMPLEMENT (short HFP)	LCER	RR	C	¤[7,9]			Da									18-14
34	HALVE (short HFP)	HER	RR		¤[7,9]			Da	EU								18-13
35	LOAD ROUNDED (long to short HFP)	LEDR	RR		¤[7,9]			Da		EO							18-17
35	LOAD ROUNDED (long to short HFP)	LRER	RR		¤[7,9]			Da		EO							18-17
36	ADD NORMALIZED (extended HFP)	AXR	RR	C	¤[7,9]		SP	Da	EU	EO		LS					18-8
37	SUBTRACT NORMALIZED (extended HFP)	SXR	RR	C	¤[7,9]		SP	Da	EU	EO		LS					18-24
38	LOAD (short)	LER	RR		¤[7,9]			Da									9-31
39	COMPARE (short HFP)	CER	RR	C	¤[7,9]			Da									18-10
3A	ADD NORMALIZED (short HFP)	AER	RR	C	¤[7,9]			Da	EU	EO		LS					18-8
3B	SUBTRACT NORMALIZED (short HFP)	SER	RR	C	¤[7,9]			Da	EU	EO		LS					18-24
3C	MULTIPLY (short to long HFP)	MDER	RR		¤[7,9]			Da	EU	EO							18-17
3C	MULTIPLY (short to long HFP)	MER	RR		¤[7,9]			Da	EU	EO							18-18
3D	DIVIDE (short HFP)	DER	RR		¤[7,9]			Da	EU	EO	FK						18-12
3E	ADD UNNORMALIZED (short HFP)	AUR	RR	C	¤[7,9]			Da		EO		LS					18-9
3F	SUBTRACT UNNORMALIZED (short HFP)	SUR	RR	C	¤[7,9]			Da		EO		LS					18-25
40	STORE HALFWORD (16)	STH	RX-a			A								ST		B_2	7-383
41	LOAD ADDRESS	LA	RX-a														7-266
42	STORE CHARACTER	STC	RX-a			A								ST		B_2	7-378
43	INSERT CHARACTER	IC	RX-a			A										B_2	7-262
44	EXECUTE	EX	RX-a		¤[9]	AI	SP				EX						7-256
45	BRANCH AND LINK	BAL	RX-a		¤[9]									B			7-35
46	BRANCH ON COUNT (32)	BCT	RX-a		¤[9]									B			7-41
47	BRANCH ON CONDITION	BC	RX-b		¤[9]									B			7-40
48	LOAD HALFWORD (32←16)	LH	RX-a			A										B_2	7-276
49	COMPARE HALFWORD (32←16)	CH	RX-a	C		A										B_2	7-150
4A	ADD HALFWORD (32←16)	AH	RX-a	C		A			IF							B_2	7-28
4B	SUBTRACT HALFWORD (32←16)	SH	RX-a	C		A			IF							B_2	7-388
4C	MULTIPLY HALFWORD (32←16)	MH	RX-a			A										B_2	7-305
4D	BRANCH AND SAVE	BAS	RX-a		¤[9]									B			7-36
4E	CONVERT TO DECIMAL (32)	CVD	RX-a		¤[9]	A								ST		B_2	7-231
4F	CONVERT TO BINARY (32)	CVB	RX-a		¤[9]	A			Dg	IK						B_2	7-230
50	STORE (32)	ST	RX-a			A								ST		B_2	7-376
51	LOAD ADDRESS EXTENDED	LAE	RX-a		¤[6]										U_1	BP	7-266
54	AND (32)	N	RX-a	C		A										B_2	7-33
55	COMPARE LOGICAL (32)	CL	RX-a	C		A										B_2	7-152
56	OR (32)	O	RX-a	C		A										B_2	7-310
57	EXCLUSIVE OR (32)	X	RX-a	C		A										B_2	7-254
58	LOAD (32)	L	RX-a			A										B_2	7-264
59	COMPARE (32)	C	RX-a	C		A										B_2	7-134
5A	ADD (32)	A	RX-a	C		A			IF							B_2	7-26
5B	SUBTRACT (32)	S	RX-a	C		A			IF							B_2	7-388
5C	MULTIPLY (64←32)	M	RX-a			A	SP									B_2	7-304
5D	DIVIDE (32←64)	D	RX-a		¤[9]	A	SP			IK						B_2	7-252
5E	ADD LOGICAL (32)	AL	RX-a	C		A										B_2	7-30
5F	SUBTRACT LOGICAL (32)	SL	RX-a	C		A										B_2	7-390
60	STORE (long)	STD	RX-a		¤[7,9]	A								ST		B_2	9-48
67	MULTIPLY (long to extended HFP)	MXD	RX-a		¤[7,9]	A	SP									B_2	18-18
68	LOAD (long)	LD	RX-a		¤[7,9]	A										B_2	9-31
69	COMPARE (long HFP)	CD	RX-a	C	¤[7,9]	A										B_2	18-10
6A	ADD NORMALIZED (long HFP)	AD	RX-a	C	¤[7,9]	A										B_2	18-8

Figure B-3. Instructions Arranged by Opcode (Part 2 of 23)

Op-code	Name	Mne-monic	Characteristics																Page
6B	SUBTRACT NORMALIZED (long HFP)	SD	RX-a	C	¤[7,9]	A		Da	EU	EO		LS					B_2		18-24
6C	MULTIPLY (long HFP)	MD	RX-a		¤[7,9]	A		Da	EU	EO							B_2		18-18
6D	DIVIDE (long HFP)	DD	RX-a		¤[7,9]	A		Da	EU	EO	FK						B_2		18-12
6E	ADD UNNORMALIZED (long HFP)	AW	RX-a	C	¤[7,9]	A		Da		EO		LS					B_2		18-9
6F	SUBTRACT UNNORMALIZED (long HFP)	SW	RX-a	C	¤[7,9]	A		Da		EO		LS					B_2		18-25
70	STORE (short)	STE	RX-a		¤[7,9]	A		Da							ST		B_2		9-48
71	MULTIPLY SINGLE (32)	MS	RX-a			A											B_2		7-307
78	LOAD (short)	LE	RX-a		¤[7,9]	A		Da									B_2		9-31
79	COMPARE (short HFP)	CE	RX-a	C	¤[7,9]	A		Da									B_2		18-10
7A	ADD NORMALIZED (short HFP)	AE	RX-a	C	¤[7,9]	A		Da	EU	EO		LS					B_2		18-8
7B	SUBTRACT NORMALIZED (short HFP)	SE	RX-a	C	¤[7,9]	A		Da	EU	EO		LS					B_2		18-24
7C	MULTIPLY (short to long HFP)	MDE	RX-a		¤[7,9]	A		Da	EU	EO							B_2		18-18
7C	MULTIPLY (short to long HFP)	ME	RX-a		¤[7,9]	A		Da	EU	EO							B_2		18-18
7D	DIVIDE (short HFP)	DE	RX-a		¤[7,9]	A		Da	EU	EO	FK						B_2		18-12
7E	ADD UNNORMALIZED (short HFP)	AU	RX-a	C	¤[7,9]	A		Da		EO		LS					B_2		18-9
7F	SUBTRACT UNNORMALIZED (short HFP)	SU	RX-a	C	¤[7,9]	A		Da		EO		LS					B_2		18-25
80	SET SYSTEM MASK	SSM	SI		P	A	SP	SO									B_2		10-133
82	LOAD PSW	LPSW	SI	L	P	A	SP			¢							B_2		10-54
83	DIAGNOSE	—		DM	P	DM											MD		10-23
84	BRANCH RELATIVE ON INDEX HIGH (32)	BRXH	RSI		¤[9]										B				7-48
85	BRANCH RELATIVE ON INDEX LOW OR EQ. (32)	BRXLE	RSI		¤[9]										B				7-48
86	BRANCH ON INDEX HIGH (32)	BXH	RS-a		¤[9]										B				7-42
87	BRANCH ON INDEX LOW OR EQUAL (32)	BXLE	RS-a		¤[9]										B				7-42
88	SHIFT RIGHT SINGLE LOGICAL (32)	SRL	RS-a																7-376
89	SHIFT LEFT SINGLE LOGICAL (32)	SLL	RS-a																7-373
8A	SHIFT RIGHT SINGLE (32)	SRA	RS-a	C															7-375
8B	SHIFT LEFT SINGLE (32)	SLA	RS-a	C					IF										7-372
8C	SHIFT RIGHT DOUBLE LOGICAL (64)	SRDL	RS-a				SP												7-374
8D	SHIFT LEFT DOUBLE LOGICAL (64)	SLDL	RS-a				SP												7-372
8E	SHIFT RIGHT DOUBLE (64)	SRDA	RS-a	C			SP												7-374
8F	SHIFT LEFT DOUBLE (64)	SLDA	RS-a	C			SP		IF										7-371
90	STORE MULTIPLE (32)	STM	RS-a			A									ST		B_2		7-385
91	TEST UNDER MASK	TM	SI	C		A										B_1			7-393
92	MOVE (immediate)	MVI	SI			A									ST	B_1			7-289
93	TEST AND SET	TS	SI	C	¤[9]	A				$					ST		B_2		7-392
94	AND (immediate)	NI	SI	C		A				£[2]					ST	B_1			7-33
95	COMPARE LOGICAL (immediate)	CLI	SI	C		A										B_1			7-152
96	OR (immediate)	OI	SI	C		A									ST	B_1			7-311
97	EXCLUSIVE OR (immediate)	XI	SI	C		A									ST	B_1			7-255
98	LOAD MULTIPLE (32)	LM	RS-a			A											B_2		7-282
99	TRACE (32)	TRACE	RS-a		P	A	SP		T	¢							B_2		10-173
9A	LOAD ACCESS MULTIPLE	LAM	RS-a		¤[6]	A	SP										UB		7-265
9B	STORE ACCESS MULTIPLE	STAM	RS-a			A	SP								ST		UB		7-377
A50	INSERT IMMEDIATE (high high)	IIHH	RI-a	N															7-263
A51	INSERT IMMEDIATE (high low)	IIHL	RI-a	N															7-263
A52	INSERT IMMEDIATE (low high)	IILH	RI-a	N															7-263
A53	INSERT IMMEDIATE (low low)	IILL	RI-a	N															7-263
A54	AND IMMEDIATE (high high)	NIHH	RI-a	C N															7-34
A55	AND IMMEDIATE (high low)	NIHL	RI-a	C N															7-34
A56	AND IMMEDIATE (low high)	NILH	RI-a	C N															7-35
A57	AND IMMEDIATE (low low)	NILL	RI-a	C N															7-35
A58	OR IMMEDIATE (high high)	OIHH	RI-a	C N															7-312
A59	OR IMMEDIATE (high low)	OIHL	RI-a	C N															7-312
A5A	OR IMMEDIATE (low high)	OILH	RI-a	C N															7-312
A5B	OR IMMEDIATE (low low)	OILL	RI-a	C N															7-312

Figure B-3. Instructions Arranged by Opcode (Part 3 of 23)

Op-code	Name	Mne-monic	Characteristics	Page
A5C	LOAD LOGICAL IMMEDIATE (high high)	LLIHH	RI-a N	7-281
A5D	LOAD LOGICAL IMMEDIATE (high low)	LLIHL	RI-a N	7-281
A5E	LOAD LOGICAL IMMEDIATE (low high)	LLILH	RI-a N	7-281
A5F	LOAD LOGICAL IMMEDIATE (low low)	LLILL	RI-a N	7-281
A70	TEST UNDER MASK (low high)	TMLH	RI-a C N	7-393
A70	TEST UNDER MASK HIGH	TMH	RI-a C	7-393
A71	TEST UNDER MASK (low low)	TMLL	RI-a C N	7-393
A71	TEST UNDER MASK LOW	TML	RI-a C	7-393
A72	TEST UNDER MASK (high high)	TMHH	RI-a C N	7-393
A73	TEST UNDER MASK (high low)	TMHL	RI-a C N	7-393
A74	BRANCH RELATIVE ON CONDITION	BRC	RI-c ¤[10] B	7-46
A75	BRANCH RELATIVE AND SAVE	BRAS	RI-b ¤[9] B	7-46
A76	BRANCH RELATIVE ON COUNT (32)	BRCT	RI-b ¤[9] B	7-47
A77	BRANCH RELATIVE ON COUNT (64)	BRCTG	RI-b N ¤[9] B	7-47
A78	LOAD HALFWORD IMMEDIATE (32←16)	LHI	RI-a	7-276
A79	LOAD HALFWORD IMMEDIATE (64←16)	LGHI	RI-a N	7-276
A7A	ADD HALFWORD IMMEDIATE (32←16)	AHI	RI-a C IF	7-28
A7B	ADD HALFWORD IMMEDIATE (64←16)	AGHI	RI-a C N IF	7-29
A7C	MULTIPLY HALFWORD IMMEDIATE (32←16)	MHI	RI-a	7-305
A7D	MULTIPLY HALFWORD IMMEDIATE (64←16)	MGHI	RI-a N	7-305
A7E	COMPARE HALFWORD IMMEDIATE (32←16)	CHI	RI-a C	7-150
A7F	COMPARE HALFWORD IMMEDIATE (64←16)	CGHI	RI-a C N	7-150
A8	MOVE LONG EXTENDED	MVCLE	RS-a C ¤[9] A SP IC ST R_1 R_3	7-294
A9	COMPARE LOGICAL LONG EXTENDED	CLCLE	RS-a C ¤[9] A SP IC R_1 R_3	7-160
AC	STORE THEN AND SYSTEM MASK	STNSM	SI P A ST B_1	10-164
AD	STORE THEN OR SYSTEM MASK	STOSM	SI P A SP ST B_1	10-164
AE	SIGNAL PROCESSOR	SIGP	RS-a C P $	10-133
AF	MONITOR CALL	MC	SI ¤[4,8,9] SP ME	7-288
B1	LOAD REAL ADDRESS (32)	LRA	RX-a C P A[1]* SO BP	10-56
B202	STORE CPU ID	STIDP	S P A SP ST B_2	10-136
B204	SET CLOCK	SCK	S C P A SP B_2	10-121
B205	STORE CLOCK	STCK	S C ¤[8,9] A $ ST B_2	7-379
B206	SET CLOCK COMPARATOR	SCKC	S P A SP B_2	10-122
B207	STORE CLOCK COMPARATOR	STCKC	S P A SP ST B_2	10-135
B208	SET CPU TIMER	SPT	S P A SP B_2	10-123
B209	STORE CPU TIMER	STPT	S P A SP ST B_2	10-138
B20A	SET PSW KEY FROM ADDRESS	SPKA	S Q	10-124
B20B	INSERT PSW KEY	IPK	S Q G2	10-30
B20D	PURGE TLB	PTLB	S P $	10-116
B210	SET PREFIX	SPX	S P A SP $ B_2	10-123
B211	STORE PREFIX	STPX	S P A SP ST B_2	10-139
B212	STORE CPU ADDRESS	STAP	S P A SP ST B_2	10-136
B218	PROGRAM CALL	PC	S Q A[1]* Z^1 T ¢ GM B ST	10-90
B219	SET ADDRESS SPACE CONTROL	SAC	S Q SP SW ¢	10-120
B21A	COMPARE AND FORM CODEWORD	CFC	S C ¤[9] A SP II GM I1	7-137
B221	INVALIDATE PAGE TABLE ENTRY	IPTE	RRF-a P A[1] SP $	10-37
B222	INSERT PROGRAM MASK	IPM	RRE	7-264
B223	INSERT VIRTUAL STORAGE KEY	IVSK	RRE Q A[1]* SO R_2	10-31
B224	INSERT ADDRESS SPACE CONTROL	IAC	RRE C Q SO	10-29
B225	SET SECONDARY ASN	SSAR	RRE ¤[1] A[1]* Z^3 T ¢	10-125
B226	EXTRACT PRIMARY ASN	EPAR	RRE Q SO	10-24
B227	EXTRACT SECONDARY ASN	ESAR	RRE Q SO	10-24
B228	PROGRAM TRANSFER	PT	RRE Q A[1]* SP Z^2 T ¢ B	10-107
B229	INSERT STORAGE KEY EXTENDED	ISKE	RRE P A[1]*	10-30
B22A	RESET REFERENCE BIT EXTENDED	RRBE	RRE C P A[1]*	10-116

Figure B-3. Instructions Arranged by Opcode (Part 4 of 23)

Op-code	Name	Mnemonic	Characteristics																		Page
B22B	SET STORAGE KEY EXTENDED	SSKE	RRF-c	C[1]		P	A[1]*		II	¢											10-130
B22C	TEST BLOCK	TB	RRE	C		P	A[1]*		II	$	G0										10-167
B22D	DIVIDE (extended HFP)	DXR	RRE			¤[7,9]		SP	Da	EU	EO	FK									18-12
B22E	PAGE IN	PGIN	RRE	C	ES	P	A[1]			¢											10-72
B22F	PAGE OUT	PGOUT	RRE	C	ES	P	A[1]			¢											10-73
B230	CLEAR SUBCHANNEL	CSCH	S	C		P			OP	¢	GS										14-5
B231	HALT SUBCHANNEL	HSCH	S	C		P			OP	¢	GS										14-6
B232	MODIFY SUBCHANNEL	MSCH	S	C		P	A	SP	OP	¢	GS									B_2	14-7
B233	START SUBCHANNEL	SSCH	S	C		P	A	SP	OP	¢	GS									B_2	14-15
B234	STORE SUBCHANNEL	STSCH	S	C		P	A	SP	OP	¢	GS						ST			B_2	14-18
B235	TEST SUBCHANNEL	TSCH	S	C		P	A	SP	OP	¢	GS						ST			B_2	14-21
B236	TEST PENDING INTERRUPTION	TPI	S	C		P	A[1]*	SP		¢							ST			B_2	14-19
B237	SET ADDRESS LIMIT	SAL	S			P			OP	¢	G1										14-12
B238	RESUME SUBCHANNEL	RSCH	S	C		P			OP	¢	GS										14-10
B239	STORE CHANNEL REPORT WORD	STCRW	S	C		P	A	SP		¢							ST			B_2	14-17
B23A	STORE CHANNEL PATH STATUS	STCPS	S			P	A	SP		¢							ST			B_2	14-16
B23B	RESET CHANNEL PATH	RCHP	S	C		P															14-9
B23C	SET CHANNEL MONITOR	SCHM	S			P			OP	¢	GM										14-13
B240	BRANCH AND STACK	BAKR	RRE			¤[1]	A[1]*		Z[5]	T						B	ST				10-11
B241	CHECKSUM	CKSM	RRE	C		¤[9]	A	SP	IC											R_2	7-49
B244	SQUARE ROOT (long HFP)	SQDR	RRE			¤[7,9]			Da		SQ										18-23
B245	SQUARE ROOT (short HFP)	SQER	RRE			¤[7,9]			Da		SQ										18-23
B246	STORE USING REAL ADDRESS (32)	STURA	RRE			P	A[1]	SP									SU				10-165
B247	MODIFY STACKED STATE	MSTA	RRE			¤[1]	A[1]*	SP	SE								ST				10-61
B248	PURGE ALB	PALB	RRE			P				$											10-116
B249	EXTRACT STACKED REGISTERS (32)	EREG	RRE			¤[1]	A[1]*		SE										U_1	U_2	10-25
B24A	EXTRACT STACKED STATE	ESTA	RRE	C		¤[1]	A[1]*	SP	SE												10-26
B24B	LOAD USING REAL ADDRESS (32)	LURA	RRE			P	A[1]	SP													10-60
B24C	TEST ACCESS	TAR	RRE	C		¤[1]	A[1]*												U_1		10-165
B24D	COPY ACCESS	CPYA	RRE			¤[6]													U_1	U_2	7-252
B24E	SET ACCESS	SAR	RRE			¤[6]													U_1		7-370
B24F	EXTRACT ACCESS	EAR	RRE																	U_2	7-257
B250	COMPARE AND SWAP AND PURGE (32)	CSP	RRE	C		P	A[1]	SP		$							ST			R_2	10-21
B252	MULTIPLY SINGLE (32)	MSR	RRE																		7-307
B254	MOVE PAGE	MVPG	RRE	C		Q	A	SP			G0						ST		R_1	R_2	10-62
B255	MOVE STRING	MVST	RRE	C		¤[9]	A	SP	IC		G0						ST		R_1	R_2	7-301
B257	COMPARE UNTIL SUBSTRING EQUAL	CUSE	RRE	C		¤[9]	A	SP	II		GM								R_1	R_2	7-167
B258	BRANCH IN SUBSPACE GROUP	BSG	RRE			¤[1]	A[1]*		SO	T						B				R_2	10-13
B25A	BRANCH AND SET AUTHORITY	BSA	RRE			Q	A[1]*		SO	T						B					10-7
B25D	COMPARE LOGICAL STRING	CLST	RRE	C		¤[9]	A	SP	IC		G0								R_1	R_2	7-166
B25E	SEARCH STRING	SRST	RRE	C		¤[9]	A	SP	IC		G0									R_2	7-366
B263	COMPRESSION CALL	CMPSC	RRE	C		¤[5,9]	A	SP	II	Dg	GM						ST		R_1	R_2	7-170
B276	CANCEL SUBCHANNEL	XSCH	S	C		P			OP	¢	GS										14-3
B277	RESUME PROGRAM	RP	S	L		Q	A	SP	WE	T						B				B_2	10-117
B278	STORE CLOCK EXTENDED	STCKE	S	C		¤[8,9]	A			$							ST			B_2	7-380
B279	SET ADDRESS SPACE CONTROL FAST	SACF	S			Q		SP	SW												10-120
B27C	STORE CLOCK FAST	STCKF	S	C	SC	¤[8,9]	A										ST			B_2	7-379
B27D	STORE SYSTEM INFORMATION	STSI	S	C		P	A	SP			GM						ST			B_2	10-140
B299	SET BFP ROUNDING MODE (2 bit)	SRNM	S			¤[7,9]			Db												9-47
B29C	STORE FPC	STFPC	S			¤[7,9]	A		Db								ST			B_2	9-49
B29D	LOAD FPC	LFPC	S			¤[7,9]	A	SP	Db											B_2	9-31
B2A5	TRANSLATE EXTENDED	TRE	RRE	C		¤[9]	A	SP	IC								ST		R_1	R_2	7-409
B2A6	CONVERT UNICODE TO UTF-8	CUUTF	RRF-c	C		¤[5,9]	A	SP	IC								ST		R_1	R_2	7-234
B2A6	CONVERT UTF-16 TO UTF-8	CU21	RRF-c	C		¤[5,9]	A	SP	IC								ST		R_1	R_2	7-234
B2A7	CONVERT UTF-8 TO UNICODE	CUTFU	RRF-c	C		¤[5,9]	A	SP	IC								ST		R_1	R_2	7-244

Figure B-3. Instructions Arranged by Opcode (Part 5 of 23)

Op-code	Name	Mne-monic	Characteristics																		Page
B2A7	CONVERT UTF-8 TO UTF-16	CU12	RRF-c	C		¤[5,9]	A	SP	IC								ST	R_1	R_2	7-244
B2B0	STORE FACILITY LIST EXTENDED	STFLE	S	C	FL	¤[1]	A	SP				G0					ST		B_2	7-382
B2B1	STORE FACILITY LIST	STFL	S		N3	P														10-138
B2B2	LOAD PSW EXTENDED	LPSWE	S	L	N	P	A	SP			¢								B_2	10-55
B2B8	SET BFP ROUNDING MODE (3 bit)	SRNMB	S		F	¤[7,9]		SP	Db											9-47
B2B9	SET DFP ROUNDING MODE	SRNMT	S		TR	¤[7,9]			Dt											9-47
B2BD	LOAD FPC AND SIGNAL	LFAS	S		XF	¤[7,9]	A	SP	Dt	Xg									B_2	9-32
B2E8	PERFORM PROCESSOR ASSIST	PPA	RRF-c		PA	¤[1]														7-345
B2EC	EXTRACT TRANSACTION NESTING DEPTH	ETND	RRE		TX	¤[9]			SO											7-261
B2F8	TRANSACTION END	TEND	S	C	TX				SO		$	EX								7-401
B2FA	NEXT INSTRUCTION ACCESS INTENT	NIAI	IE		EH															7-308
B2FC	TRANSACTION ABORT	TABORT	S		TX	¤[9]		SP	SO		$	EX								7-394
B2FF	TRAP	TRAP4	S			¤[1]	A*		SO	T							B	ST		10-174
B300	LOAD POSITIVE (short BFP)	LPEBR	RRE	C		¤[7,9]			Db											19-35
B301	LOAD NEGATIVE (short BFP)	LNEBR	RRE	C		¤[7,9]			Db											19-34
B302	LOAD AND TEST (short BFP)	LTEBR	RRE	C		¤[7,9]			Db	Xi										19-31
B303	LOAD COMPLEMENT (short BFP)	LCEBR	RRE	C		¤[7,9]			Db											19-31
B304	LOAD LENGTHENED (short to long BFP)	LDEBR	RRE			¤[7,9]			Db	Xi										19-33
B305	LOAD LENGTHENED (long to extended BFP)	LXDBR	RRE			¤[7,9]		SP	Db	Xi										19-33
B306	LOAD LENGTHENED (short to extended BFP)	LXEBR	RRE			¤[7,9]		SP	Db	Xi										19-33
B307	MULTIPLY (long to extended BFP)	MXDBR	RRE			¤[7,9]		SP	Db	Xi										19-37
B308	COMPARE AND SIGNAL (short BFP)	KEBR	RRE	C		¤[7,9]			Db	Xi										19-18
B309	COMPARE (short BFP)	CEBR	RRE	C		¤[7,9]			Db	Xi										19-17
B30A	ADD (short BFP)	AEBR	RRE	C		¤[7,9]			Db	Xi			Xo	Xu	Xx					19-15
B30B	SUBTRACT (short BFP)	SEBR	RRE	C		¤[7,9]			Db	Xi			Xo	Xu	Xx					19-40
B30C	MULTIPLY (short to long BFP)	MDEBR	RRE			¤[7,9]			Db	Xi										19-37
B30D	DIVIDE (short BFP)	DEBR	RRE			¤[7,9]			Db	Xi	Xz		Xo	Xu	Xx					19-27
B30E	MULTIPLY AND ADD (short BFP)	MAEBR	RRD			¤[7,9]			Db	Xi			Xo	Xu	Xx					19-38
B30F	MULTIPLY AND SUBTRACT (short BFP)	MSEBR	RRD			¤[7,9]			Db	Xi			Xo	Xu	Xx					19-38
B310	LOAD POSITIVE (long BFP)	LPDBR	RRE	C		¤[7,9]			Db											19-35
B311	LOAD NEGATIVE (long BFP)	LNDBR	RRE	C		¤[7,9]			Db											19-34
B312	LOAD AND TEST (long BFP)	LTDBR	RRE	C		¤[7,9]			Db	Xi										19-31
B313	LOAD COMPLEMENT (long BFP)	LCDBR	RRE	C		¤[7,9]			Db											19-31
B314	SQUARE ROOT (short BFP)	SQEBR	RRE			¤[7,9]			Db	Xi					Xx					19-40
B315	SQUARE ROOT (long BFP)	SQDBR	RRE			¤[7,9]			Db	Xi					Xx					19-40
B316	SQUARE ROOT (extended BFP)	SQXBR	RRE			¤[7,9]		SP	Db	Xi					Xx					19-40
B317	MULTIPLY (short BFP)	MEEBR	RRE			¤[7,9]			Db	Xi			Xo	Xu	Xx					19-37
B318	COMPARE AND SIGNAL (long BFP)	KDBR	RRE	C		¤[7,9]			Db	Xi										19-18
B319	COMPARE (long BFP)	CDBR	RRE	C		¤[7,9]			Db	Xi										19-17
B31A	ADD (long BFP)	ADBR	RRE	C		¤[7,9]			Db	Xi			Xo	Xu	Xx					19-15
B31B	SUBTRACT (long BFP)	SDBR	RRE	C		¤[7,9]			Db	Xi			Xo	Xu	Xx					19-40
B31C	MULTIPLY (long BFP)	MDBR	RRE			¤[7,9]			Db	Xi			Xo	Xu	Xx					19-37
B31D	DIVIDE (long BFP)	DDBR	RRE			¤[7,9]			Db	Xi	Xz		Xo	Xu	Xx					19-27
B31E	MULTIPLY AND ADD (long BFP)	MADBR	RRD			¤[7,9]			Db	Xi			Xo	Xu	Xx					19-38
B31F	MULTIPLY AND SUBTRACT (long BFP)	MSDBR	RRD			¤[7,9]			Db	Xi			Xo	Xu	Xx					19-38
B324	LOAD LENGTHENED (short to long HFP)	LDER	RRE			¤[7,9]			Da											18-15
B325	LOAD LENGTHENED (long to extended HFP)	LXDR	RRE			¤[7,9]		SP	Da											18-15
B326	LOAD LENGTHENED (short to extended HFP)	LXER	RRE			¤[7,9]		SP	Da											18-15
B32E	MULTIPLY AND ADD (short HFP)	MAER	RRD		HM	¤[7,9]			Da	EU	EO									18-19
B32F	MULTIPLY AND SUBTRACT (short HFP)	MSER	RRD		HM	¤[7,9]			Da	EU	EO									18-19
B336	SQUARE ROOT (extended HFP)	SQXR	RRE			¤[7,9]		SP	Da			SQ								18-23
B337	MULTIPLY (short HFP)	MEER	RRE			¤[7,9]			Da	EU	EO									18-17
B338	MULTIPLY AND ADD UNNRM. (long to ext. low HFP)	MAYLR	RRD		UE	¤[7,9]			Da											18-20
B339	MULTIPLY UNNORM. (long to ext. low HFP)	MYLR	RRD		UE	¤[7,9]			Da											18-22

Figure B-3. Instructions Arranged by Opcode (Part 6 of 23)

Op-code	Name	Mne-monic	Characteristics														Page
B33A	MULTIPLY & ADD UNNORMALIZED (long to ext. HFP)	MAYR	RRD		UE	¤[7,9]		Da									18-20
B33B	MULTIPLY UNNORMALIZED (long to ext. HFP)	MYR	RRD		UE	¤[7,9]	SP	Da									18-22
B33C	MULTIPLY AND ADD UNNRM. (long to ext. high HFP)	MAYHR	RRD		UE	¤[7,9]		Da									18-20
B33D	MULTIPLY UNNORM. (long to ext. high HFP)	MYHR	RRD		UE	¤[7,9]		Da									18-22
B33E	MULTIPLY AND ADD (long HFP)	MADR	RRD		HM	¤[7,9]		Da	EU	EO							18-19
B33F	MULTIPLY AND SUBTRACT (long HFP)	MSDR	RRD		HM	¤[7,9]		Da	EU	EO							18-19
B340	LOAD POSITIVE (extended BFP)	LPXBR	RRE	C		¤[7,9]	SP	Db									19-35
B341	LOAD NEGATIVE (extended BFP)	LNXBR	RRE	C		¤[7,9]	SP	Db									19-34
B342	LOAD AND TEST (extended BFP)	LTXBR	RRE	C		¤[7,9]	SP	Db	Xi								19-31
B343	LOAD COMPLEMENT (extended BFP)	LCXBR	RRE	C		¤[7,9]	SP	Db									19-31
B344	LOAD ROUNDED (long to short BFP)	LEDBR	RRE			¤[7,9]		Db	Xi		Xo	Xu	Xx				19-35
B344	LOAD ROUNDED (long to short BFP)	LEDBRA	RRF-e		F	¤[7,9]	SP	Db	Xi		Xo	Xu	Xx				19-35
B345	LOAD ROUNDED (extended to long BFP)	LDXBR	RRE			¤[7,9]	SP	Db	Xi		Xo	Xu	Xx				19-35
B345	LOAD ROUNDED (extended to long BFP)	LDXBRA	RRF-e		F	¤[7,9]	SP	Db	Xi		Xo	Xu	Xx				19-35
B346	LOAD ROUNDED (extended to short BFP)	LEXBR	RRE			¤[7,9]	SP	Db	Xi		Xo	Xu	Xx				19-35
B346	LOAD ROUNDED (extended to short BFP)	LEXBRA	RRF-e		F	¤[7,9]	SP	Db	Xi		Xo	Xu	Xx				19-35
B347	LOAD FP INTEGER (extended BFP)	FIXBR	RRF-e			¤[7,9]	SP	Db	Xi				Xx				19-32
B347	LOAD FP INTEGER (extended BFP)	FIXBRA	RRF-e		F	¤[7,9]	SP	Db	Xi				Xx				19-32
B348	COMPARE AND SIGNAL (extended BFP)	KXBR	RRE	C		¤[7,9]	SP	Db	Xi								19-18
B349	COMPARE (extended BFP)	CXBR	RRE	C		¤[7,9]	SP	Db	Xi								19-17
B34A	ADD (extended BFP)	AXBR	RRE	C		¤[7,9]	SP	Db	Xi		Xo	Xu	Xx				19-15
B34B	SUBTRACT (extended BFP)	SXBR	RRE	C		¤[7,9]	SP	Db	Xi		Xo	Xu	Xx				19-40
B34C	MULTIPLY (extended BFP)	MXBR	RRE			¤[7,9]	SP	Db	Xi		Xo	Xu	Xx				19-37
B34D	DIVIDE (extended BFP)	DXBR	RRE			¤[7,9]	SP	Db	Xi	Xz	Xo	Xu	Xx				19-27
B350	CONVERT HFP TO BFP (long to short)	TBEDR	RRF-e	C		¤[7,9]	SP	Da									9-28
B351	CONVERT HFP TO BFP (long)	TBDR	RRF-e	C		¤[7,9]	SP	Da									9-28
B353	DIVIDE TO INTEGER (short BFP)	DIEBR	RRF-b	C		¤[7,9]	SP	Db	Xi			Xu	Xx				19-28
B357	LOAD FP INTEGER (short BFP)	FIEBR	RRF-e			¤[7,9]	SP	Db	Xi				Xx				19-32
B357	LOAD FP INTEGER (short BFP)	FIEBRA	RRF-e		F	¤[7,9]	SP	Db	Xi				Xx				19-32
B358	CONVERT BFP TO HFP (short to long)	THDER	RRE	C		¤[7,9]		Da									9-27
B359	CONVERT BFP TO HFP (long)	THDR	RRE	C		¤[7,9]		Da									9-27
B35B	DIVIDE TO INTEGER (long BFP)	DIDBR	RRF-b	C		¤[7,9]	SP	Db	Xi			Xu	Xx				19-28
B35F	LOAD FP INTEGER (long BFP)	FIDBR	RRF-e			¤[7,9]	SP	Db	Xi				Xx				19-32
B35F	LOAD FP INTEGER (long BFP)	FIDBRA	RRF-e		F	¤[7,9]	SP	Db	Xi				Xx				19-32
B360	LOAD POSITIVE (extended HFP)	LPXR	RRE	C		¤[7,9]	SP	Da									18-16
B361	LOAD NEGATIVE (extended HFP)	LNXR	RRE	C		¤[7,9]	SP	Da									18-16
B362	LOAD AND TEST (extended HFP)	LTXR	RRE	C		¤[7,9]	SP	Da									18-14
B363	LOAD COMPLEMENT (extended HFP)	LCXR	RRE	C		¤[7,9]	SP	Da									18-14
B365	LOAD (extended)	LXR	RRE			¤[7,9]	SP	Da									9-31
B366	LOAD ROUNDED (extended to short HFP)	LEXR	RRE			¤[7,9]	SP	Da		EO							18-17
B367	LOAD FP INTEGER (extended HFP)	FIXR	RRE			¤[7,9]	SP	Da									18-15
B369	COMPARE (extended HFP)	CXR	RRE	C		¤[7,9]	SP	Da									18-10
B370	LOAD POSITIVE (long)	LPDFR	RRE		FS	¤[7,9]		Da									9-34
B371	LOAD NEGATIVE (long)	LNDFR	RRE		FS	¤[7,9]		Da									9-34
B372	COPY SIGN (long)	CPSDR	RRF-b		FS	¤[7,9]		Da									9-30
B373	LOAD COMPLEMENT (long)	LCDFR	RRE		FS	¤[7,9]		Da									9-31
B374	LOAD ZERO (short)	LZER	RRE			¤[7,9]		Da									9-35
B375	LOAD ZERO (long)	LZDR	RRE			¤[7,9]		Da									9-35
B376	LOAD ZERO (extended)	LZXR	RRE			¤[7,9]	SP	Da									9-35
B377	LOAD FP INTEGER (short HFP)	FIER	RRE			¤[7,9]		Da									18-15
B37F	LOAD FP INTEGER (long HFP)	FIDR	RRE			¤[7,9]		Da									18-15
B384	SET FPC	SFPC	RRE			¤[7,9]	SP	Db									9-47
B385	SET FPC AND SIGNAL	SFASR	RRE		XF	¤[7,9]	SP	Dt	Xg								9-48

Figure B-3. Instructions Arranged by Opcode (Part 7 of 23)

Op-code	Name	Mne-monic	Characteristics																				Page
B38C	EXTRACT FPC	EFPC	RRE				$\alpha^{7,9}$		Db														9-30
B390	CONVERT FROM LOGICAL (32 to short BFP)	CELFBR	RRF-e		F		$\alpha^{7,9}$	SP	Db							Xx							19-21
B391	CONVERT FROM LOGICAL (32 to long BFP)	CDLFBR	RRF-e		F		$\alpha^{7,9}$	SP	Db														19-21
B392	CONVERT FROM LOGICAL (32 to extended BFP)	CXLFBR	RRF-e		F		$\alpha^{7,9}$	SP	Db														19-21
B394	CONVERT FROM FIXED (32 to short BFP)	CEFBR	RRE				$\alpha^{7,9}$		Db							Xx							19-19
B394	CONVERT FROM FIXED (32 to short BFP)	CEFBRA	RRF-e		F		$\alpha^{7,9}$	SP	Db							Xx							19-19
B395	CONVERT FROM FIXED (32 to long BFP)	CDFBR	RRE				$\alpha^{7,9}$		Db														19-19
B395	CONVERT FROM FIXED (32 to long BFP)	CDFBRA	RRF-e		F		$\alpha^{7,9}$	SP	Db														19-19
B396	CONVERT FROM FIXED (32 to extended BFP)	CXFBR	RRE				$\alpha^{7,9}$	SP	Db														19-19
B396	CONVERT FROM FIXED (32 to extended BFP)	CXFBRA	RRF-e		F		$\alpha^{7,9}$	SP	Db														19-19
B398	CONVERT TO FIXED (short BFP to 32)	CFEBR	RRF-e	C			$\alpha^{7,9}$	SP	Db	Xi						Xx							19-22
B398	CONVERT TO FIXED (short BFP to 32)	CFEBRA	RRF-e	C	F		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-22
B399	CONVERT TO FIXED (long BFP to 32)	CFDBR	RRF-e	C			$\alpha^{7,9}$	SP	Db	Xi						Xx							19-22
B399	CONVERT TO FIXED (long BFP to 32)	CFDBRA	RRF-e	C	F		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-22
B39A	CONVERT TO FIXED (extended BFP to 32)	CFXBR	RRF-e	C			$\alpha^{7,9}$	SP	Db	Xi						Xx							19-22
B39A	CONVERT TO FIXED (extended BFP to 32)	CFXBRA	RRF-e	C	F		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-22
B39C	CONVERT TO LOGICAL (short BFP to 32)	CLFEBR	RRF-e	C	F		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-25
B39D	CONVERT TO LOGICAL (long BFP to 32)	CLFDBR	RRF-e	C	F		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-25
B39E	CONVERT TO LOGICAL (extended BFP to 32)	CLFXBR	RRF-e	C	F		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-25
B3A0	CONVERT FROM LOGICAL (64 to short BFP)	CELGBR	RRF-e		F		$\alpha^{7,9}$	SP	Db							Xx							19-21
B3A1	CONVERT FROM LOGICAL (64 to long BFP)	CDLGBR	RRF-e		F		$\alpha^{7,9}$	SP	Db							Xx							19-21
B3A2	CONVERT FROM LOGICAL (64 to extended BFP)	CXLGBR	RRF-e		F		$\alpha^{7,9}$	SP	Db														19-21
B3A4	CONVERT FROM FIXED (64 to short BFP)	CEGBR	RRE		N		$\alpha^{7,9}$		Db							Xx							19-19
B3A4	CONVERT FROM FIXED (64 to short BFP)	CEGBRA	RRF-e		F		$\alpha^{7,9}$	SP	Db							Xx							19-19
B3A5	CONVERT FROM FIXED (64 to long BFP)	CDGBR	RRE		N		$\alpha^{7,9}$		Db							Xx							19-19
B3A5	CONVERT FROM FIXED (64 to long BFP)	CDGBRA	RRF-e		F		$\alpha^{7,9}$	SP	Db							Xx							19-19
B3A6	CONVERT FROM FIXED (64 to extended BFP)	CXGBR	RRE		N		$\alpha^{7,9}$	SP	Db														19-19
B3A6	CONVERT FROM FIXED (64 to extended BFP)	CXGBRA	RRF-e		F		$\alpha^{7,9}$	SP	Db														19-19
B3A8	CONVERT TO FIXED (short BFP to 64)	CGEBR	RRF-e	C	N		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-22
B3A8	CONVERT TO FIXED (short BFP to 64)	CGEBRA	RRF-e	C	F		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-22
B3A9	CONVERT TO FIXED (long BFP to 64)	CGDBR	RRF-e	C	N		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-22
B3A9	CONVERT TO FIXED (long BFP to 64)	CGDBRA	RRF-e	C	F		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-22
B3AA	CONVERT TO FIXED (extended BFP to 64)	CGXBR	RRF-e	C	N		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-22
B3AA	CONVERT TO FIXED (extended BFP to 64)	CGXBRA	RRF-e	C	F		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-22
B3AC	CONVERT TO LOGICAL (short BFP to 64)	CLGEBR	RRF-e	C	F		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-25
B3AD	CONVERT TO LOGICAL (long BFP to 64)	CLGDBR	RRF-e	C	F		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-25
B3AE	CONVERT TO LOGICAL (extended BFP to 64)	CLGXBR	RRF-e	C	F		$\alpha^{7,9}$	SP	Db	Xi						Xx							19-25
B3B4	CONVERT FROM FIXED (32 to short HFP)	CEFR	RRE				$\alpha^{7,9}$		Da														18-11
B3B5	CONVERT FROM FIXED (32 to long HFP)	CDFR	RRE				$\alpha^{7,9}$		Da														18-11
B3B6	CONVERT FROM FIXED (32 to extended HFP)	CXFR	RRE				$\alpha^{7,9}$	SP	Da														18-11
B3B8	CONVERT TO FIXED (short HFP to 32)	CFER	RRF-e	C			$\alpha^{7,9}$	SP	Da														18-11
B3B9	CONVERT TO FIXED (long HFP to 32)	CFDR	RRF-e	C			$\alpha^{7,9}$	SP	Da														18-11
B3BA	CONVERT TO FIXED (extended HFP to 32)	CFXR	RRF-e	C			$\alpha^{7,9}$	SP	Da														18-11
B3C1	LOAD FPR FROM GR (64 to long)	LDGR	RRE			FG	$\alpha^{7,9}$		Da														9-34
B3C4	CONVERT FROM FIXED (64 to short HFP)	CEGR	RRE		N		$\alpha^{7,9}$		Da														18-11
B3C5	CONVERT FROM FIXED (64 to long HFP)	CDGR	RRE		N		$\alpha^{7,9}$		Da														18-11
B3C6	CONVERT FROM FIXED (64 to extended HFP)	CXGR	RRE		N		$\alpha^{7,9}$	SP	Da														18-11
B3C8	CONVERT TO FIXED (short HFP to 64)	CGER	RRF-e	C	N		$\alpha^{7,9}$	SP	Da														18-11
B3C9	CONVERT TO FIXED (long HFP to 64)	CGDR	RRF-e	C	N		$\alpha^{7,9}$	SP	Da														18-11
B3CA	CONVERT TO FIXED (extended HFP to 64)	CGXR	RRF-e	C	N		$\alpha^{7,9}$	SP	Da														18-11
B3CD	LOAD GR FROM FPR (long to 64)	LGDR	RRE			FG	$\alpha^{7,9}$		Da														9-34
B3D0	MULTIPLY (long DFP)	MDTR	RRF-a			TF	$\alpha^{7,9}$		Dt	Xi			Xo		Xu	Xx							20-47
B3D0	MULTIPLY (long DFP)	MDTRA	RRF-a		F		$\alpha^{7,9}$		Dt	Xi			Xo		Xu	Xx	Xq						20-48
B3D1	DIVIDE (long DFP)	DDTR	RRF-a			TF	$\alpha^{7,9}$		Dt	Xi		Xz	Xo		Xu	Xx							20-37
B3D1	DIVIDE (long DFP)	DDTRA	RRF-a		F		$\alpha^{7,9}$		Dt	Xi		Xz	Xo		Xu	Xx	Xq						20-37

Figure B-3. Instructions Arranged by Opcode (Part 8 of 23)

Op-code	Name	Mne-monic	Characteristics																				Page
B3D2	ADD (long DFP)	ADTR	RRF-a	C	TF	¤[7,9]		Dt	Xi		Xo	Xu	Xx										20-19
B3D2	ADD (long DFP)	ADTRA	RRF-a	C	F	¤[7,9]		Dt	Xi		Xo	Xu	Xx	Xq									20-19
B3D3	SUBTRACT (long DFP)	SDTR	RRF-a	C	TF	¤[7,9]		Dt	Xi		Xo	Xu	Xx										20-55
B3D3	SUBTRACT (long DFP)	SDTRA	RRF-a	C	F	¤[7,9]		Dt	Xi		Xo	Xu	Xx	Xq									20-55
B3D4	LOAD LENGTHENED (short to long DFP)	LDETR	RRF-d		TF	¤[7,9]		Dt	Xi														20-45
B3D5	LOAD ROUNDED (long to short DFP)	LEDTR	RRF-e		TF	¤[7,9]		Dt	Xi		Xo	Xu	Xx	Xq									20-46
B3D6	LOAD AND TEST (long DFP)	LTDTR	RRE	C	TF	¤[7,9]		Dt	Xi														20-41
B3D7	LOAD FP INTEGER (long DFP)	FIDTR	RRF-e		TF	¤[7,9]		Dt	Xi				Xx	Xq									20-42
B3D8	MULTIPLY (extended DFP)	MXTR	RRF-a		TF	¤[7,9]	SP	Dt	Xi		Xo	Xu	Xx										20-47
B3D8	MULTIPLY (extended DFP)	MXTRA	RRF-a		F	¤[7,9]	SP	Dt	Xi		Xo	Xu	Xx	Xq									20-48
B3D9	DIVIDE (extended DFP)	DXTR	RRF-a		TF	¤[7,9]	SP	Dt	Xi	Xz	Xo	Xu	Xx										20-37
B3D9	DIVIDE (extended DFP)	DXTRA	RRF-a		F	¤[7,9]	SP	Dt	Xi	Xz	Xo	Xu	Xx	Xq									20-37
B3DA	ADD (extended DFP)	AXTR	RRF-a	C	TF	¤[7,9]	SP	Dt	Xi		Xo	Xu	Xx										20-19
B3DA	ADD (extended DFP)	AXTRA	RRF-a	C	F	¤[7,9]	SP	Dt	Xi		Xo	Xu	Xx	Xq									20-19
B3DB	SUBTRACT (extended DFP)	SXTR	RRF-a	C	TF	¤[7,9]	SP	Dt	Xi		Xo	Xu	Xx										20-55
B3DB	SUBTRACT (extended DFP)	SXTRA	RRF-a	C	F	¤[7,9]	SP	Dt	Xi		Xo	Xu	Xx	Xq									20-55
B3DC	LOAD LENGTHENED (long to extended DFP)	LXDTR	RRF-d		TF	¤[7,9]	SP	Dt	Xi														20-45
B3DD	LOAD ROUNDED (extended to long DFP)	LDXTR	RRF-e		TF	¤[7,9]	SP	Dt	Xi		Xo	Xu	Xx	Xq									20-46
B3DE	LOAD AND TEST (extended DFP)	LTXTR	RRE	C	TF	¤[7,9]	SP	Dt	Xi														20-41
B3DF	LOAD FP INTEGER (extended DFP)	FIXTR	RRF-e		TF	¤[7,9]	SP	Dt	Xi				Xx	Xq									20-42
B3E0	COMPARE AND SIGNAL (long DFP)	KDTR	RRE	C	TF	¤[7,9]		Dt	Xi														20-23
B3E1	CONVERT TO FIXED (long DFP to 64)	CGDTR	RRF-e	C	TF	¤[7,9]		Dt	Xi				Xx										20-29
B3E1	CONVERT TO FIXED (long DFP to 64)	CGDTRA	RRF-e	C	F	¤[7,9]		Dt	Xi				Xx										20-30
B3E2	CONVERT TO UNSIGNED PACKED (long DFP to 64)	CUDTR	RRE		TF	¤[7,9]		Dt															20-35
B3E3	CONVERT TO SIGNED PACKED (long DFP to 64)	CSDTR	RRF-d		TF	¤[7,9]		Dt															20-35
B3E4	COMPARE (long DFP)	CDTR	RRE	C	TF	¤[7,9]		Dt	Xi														20-22
B3E5	EXTRACT BIASED EXPONENT (long DFP to 64)	EEDTR	RRE		TF	¤[7,9]		Dt															20-39
B3E7	EXTRACT SIGNIFICANCE (long DFP to 64)	ESDTR	RRE		TF	¤[7,9]		Dt															20-39
B3E8	COMPARE AND SIGNAL (extended DFP)	KXTR	RRE	C	TF	¤[7,9]	SP	Dt	Xi														20-23
B3E9	CONVERT TO FIXED (extended DFP to 64)	CGXTR	RRF-e	C	TF	¤[7,9]	SP	Dt	Xi				Xx										20-29
B3E9	CONVERT TO FIXED (extended DFP to 64)	CGXTRA	RRF-e	C	F	¤[7,9]	SP	Dt	Xi				Xx										20-30
B3EA	CONVERT TO UNSIGNED PACKED (extended DFP to 128)	CUXTR	RRE		TF	¤[7,9]	SP	Dt															20-35
B3EB	CONVERT TO SIGNED PACKED (extended DFP to 128)	CSXTR	RRF-d		TF	¤[7,9]	SP	Dt															20-35
B3EC	COMPARE (extended DFP)	CXTR	RRE	C	TF	¤[7,9]	SP	Dt	Xi														20-22
B3ED	EXTRACT BIASED EXPONENT (extended DFP to 64)	EEXTR	RRE		TF	¤[7,9]	SP	Dt															20-39
B3EF	EXTRACT SIGNIFICANCE (extended DFP to 64)	ESXTR	RRE		TF	¤[7,9]	SP	Dt															20-39
B3F1	CONVERT FROM FIXED (64 to long DFP)	CDGTR	RRE		TF	¤[7,9]		Dt					Xx										20-24
B3F1	CONVERT FROM FIXED (64 to long DFP)	CDGTRA	RRF-e		F	¤[7,9]		Dt					Xx	Xq									20-24
B3F2	CONVERT FROM UNSIGNED PACKED (64 to long DFP)	CDUTR	RRE		TF	¤[7,9]		Dt	Dg														20-28
B3F3	CONVERT FROM SIGNED PACKED (64 to long DFP)	CDSTR	RRE		TF	¤[7,9]		Dt	Dg														20-28
B3F4	COMPARE BIASED EXPONENT (long DFP)	CEDTR	RRE	C	TF	¤[7,9]		Dt															20-23
B3F5	QUANTIZE (long DFP)	QADTR	RRF-b		TF	¤[7,9]		Dt	Xi				Xx	Xq									20-49
B3F6	INSERT BIASED EXPONENT (64 to long DFP)	IEDTR	RRF-b		TF	¤[7,9]		Dt															20-40
B3F7	REROUND (long DFP)	RRDTR	RRF-b		TF	¤[7,9]		Dt	Xi				Xx	Xq									20-52
B3F9	CONVERT FROM FIXED (64 to extended DFP)	CXGTR	RRE		TF	¤[7,9]	SP	Dt															20-24
B3F9	CONVERT FROM FIXED (64 to extended DFP)	CXGTRA	RRF-e		F	¤[7,9]	SP	Dt															20-24
B3FA	CONVERT FROM UNSIGNED PACKED (128 to ext. DFP)	CXUTR	RRE		TF	¤[7,9]	SP	Dt	Dg														20-28
B3FB	CONVERT FROM SIGNED PACKED (128 to extended DFP)	CXSTR	RRE		TF	¤[7,9]	SP	Dt	Dg														20-28

Figure B-3. Instructions Arranged by Opcode (Part 9 of 23)

Op-code	Name	Mne-monic	Characteristics																				Page
B3FC	COMPARE BIASED EXPONENT (extended DFP)	CEXTR	RRE	C	TF	¤[7,9]		SP	Dt														20-23
B3FD	QUANTIZE (extended DFP)	QAXTR	RRF-b		TF	¤[7,9]		SP	Dt	Xi					Xx	Xq							20-49
B3FE	INSERT BIASED EXPONENT (64 to extended DFP)	IEXTR	RRF-b		TF	¤[7,9]		SP	Dt														20-40
B3FF	REROUND (extended DFP)	RRXTR	RRF-b		TF	¤[7,9]		SP	Dt	Xi					Xx	Xq							20-52
B6	STORE CONTROL (32)	STCTL	RS-a			P	A	SP											ST			B_2	10-135
B7	LOAD CONTROL (32)	LCTL	RS-a			P	A	SP														B_2	10-50
B900	LOAD POSITIVE (64)	LPGR	RRE	C	N					IF													7-287
B901	LOAD NEGATIVE (64)	LNGR	RRE	C	N					IF													7-283
B902	LOAD AND TEST (64)	LTGR	RRE	C	N					IF													7-270
B903	LOAD COMPLEMENT (64)	LCGR	RRE	C	N					IF													7-273
B904	LOAD (64)	LGR	RRE		N																		7-264
B905	LOAD USING REAL ADDRESS (64)	LURAG	RRE		N	P	A[1]	SP															10-60
B906	LOAD BYTE (64←8)	LGBR	RRE		EI																		7-272
B907	LOAD HALFWORD (64←16)	LGHR	RRE		EI																		7-276
B908	ADD (64)	AGR	RRE	C	N					IF													7-26
B909	SUBTRACT (64)	SGR	RRE	C	N					IF													7-387
B90A	ADD LOGICAL (64)	ALGR	RRE	C	N																		7-30
B90B	SUBTRACT LOGICAL (64)	SLGR	RRE	C	N																		7-389
B90C	MULTIPLY SINGLE (64)	MSGR	RRE		N																		7-307
B90D	DIVIDE SINGLE (64)	DSGR	RRE		N	¤[9]		SP			IK												7-254
B90E	EXTRACT STACKED REGISTERS (64)	EREGG	RRE		N	¤[1]	A[1]*		SE												U_1	U_2	10-25
B90F	LOAD REVERSED (64)	LRVGR	RRE		N																		7-287
B910	LOAD POSITIVE (64←32)	LPGFR	RRE	C	N																		7-287
B911	LOAD NEGATIVE (64←32)	LNGFR	RRE	C	N																		7-284
B912	LOAD AND TEST (64←32)	LTGFR	RRE	C	N																		7-270
B913	LOAD COMPLEMENT (64←32)	LCGFR	RRE	C	N																		7-273
B914	LOAD (64←32)	LGFR	RRE		N																		7-264
B916	LOAD LOGICAL (64←32)	LLGFR	RRE		N																		7-278
B917	LOAD LOGICAL THIRTY ONE BITS (64←31)	LLGTR	RRE		N																		7-281
B918	ADD (64←32)	AGFR	RRE	C	N					IF													7-26
B919	SUBTRACT (64←32)	SGFR	RRE	C	N					IF													7-387
B91A	ADD LOGICAL (64←32)	ALGFR	RRE	C	N																		7-30
B91B	SUBTRACT LOGICAL (64←32)	SLGFR	RRE	C	N																		7-389
B91C	MULTIPLY SINGLE (64←32)	MSGFR	RRE		N																		7-307
B91D	DIVIDE SINGLE (64←32)	DSGFR	RRE		N	¤[9]		SP			IK												7-254
B91E	COMPUTE MESSAGE AUTHENTICATION CODE	KMAC	RRE	C	MS	¤[5,9]	A	SP	IC				GM	I1					ST			R_2	7-219
B91F	LOAD REVERSED (32)	LRVR	RRE		N3																		7-287
B920	COMPARE (64)	CGR	RRE	C	N																		7-134
B921	COMPARE LOGICAL (64)	CLGR	RRE	C	N																		7-152
B925	STORE USING REAL ADDRESS (64)	STURG	RRE		N	P	A[1]	SP											SU				10-165
B926	LOAD BYTE (32←8)	LBR	RRE		EI																		7-272
B927	LOAD HALFWORD (32←16)	LHR	RRE		EI																		7-276
B928	PERFORM CRYPTOGRAPHIC KEY MGMT. OPERATIONS	PCKMO	RRE		M3	P	A	SP					GM						ST				10-74
B929	CIPHER MESSAGE WITH AUTHENTICATION	KMA	RRF-b	C	M8	¤[5,9]	A	SP	IC				GM	I1					ST		R_1	R_2 R_3	7-78
B92A	CIPHER MESSAGE WITH CIPHER FEEDBACK	KMF	RRE	C	M4	¤[5,9]	A	SP	IC				GM	I1					ST		R_1	R_2	7-92
B92B	CIPHER MESSAGE WITH OUTPUT FEEDBACK	KMO	RRE	C	M4	¤[5,9]	A	SP	IC				GM	I1					ST		R_1	R_2	7-120
B92C	PERFORM CRYPTOGRAPHIC COMPUTATION	PCC	RRE	C	M4	¤[5,9]	A	SP	IC				GM	I1					ST				7-315
B92D	CIPHER MESSAGE WITH COUNTER	KMCTR	RRF-b	C	M4	¤[5,9]	A	SP	IC				GM	I1					ST		R_1,R_2,R_3		7-107
B92E	CIPHER MESSAGE	KM	RRE	C	MS	¤[5,9]	A	SP	IC				GM	I1					ST		R_1	R_2	7-53
B92F	CIPHER MESSAGE WITH CHAINING	KMC	RRE	C	MS	¤[5,9]	A	SP	IC				GM	I1					ST		R_1	R_2	7-53
B930	COMPARE (64←32)	CGFR	RRE	C	N																		7-134
B931	COMPARE LOGICAL (64←32)	CLGFR	RRE	C	N																		7-152
B93C	PERFORM RANDOM NUMBER OPERATION	PRNO	RRE	C	M5	¤[5,9]	A	SP	IC	Dg			GM	I1					ST		R_1	R_2	7-346
B93E	COMPUTE INTERMEDIATE MESSAGE DIGEST	KIMD	RRE	C	MS	¤[5,9]	A	SP	IC				GM	I1					ST			R_2	7-188

Figure B-3. Instructions Arranged by Opcode (Part 10 of 23)

Op-code	Name	Mne-monic	Characteristics	Page
B93F	COMPUTE LAST MESSAGE DIGEST	KLMD	RRE C MS ¤[5,9] A SP IC GM I1 ST R_2	7-201
B941	CONVERT TO FIXED (long DFP to 32)	CFDTR	RRF-e C F ¤[7,9] Dt Xi Xx	20-30
B942	CONVERT TO LOGICAL (long DFP to 64)	CLGDTR	RRF-e C F ¤[7,9] Dt Xi Xx	20-32
B943	CONVERT TO LOGICAL (long DFP to 32)	CLFDTR	RRF-e C F ¤[7,9] Dt Xi Xx	20-32
B946	BRANCH ON COUNT (64)	BCTGR	RRE N ¤[9] B	7-41
B949	CONVERT TO FIXED (extended DFP to 32)	CFXTR	RRF-e C F ¤[7,9] SP Dt Xi Xx	20-30
B94A	CONVERT TO LOGICAL (extended DFP to 64)	CLGXTR	RRF-e C F ¤[7,9] SP Dt Xi Xx	20-32
B94B	CONVERT TO LOGICAL (extended DFP to 32)	CLFXTR	RRF-e C F ¤[7,9] SP Dt Xi Xx	20-32
B951	CONVERT FROM FIXED (32 to long DFP)	CDFTR	RRE F ¤[7,9] Dt	20-24
B952	CONVERT FROM LOGICAL (64 to long DFP)	CDLGTR	RRF-e F ¤[7,9] Dt Xx Xq	20-25
B953	CONVERT FROM LOGICAL (32 to long DFP)	CDLFTR	RRF-e F ¤[7,9] Dt	20-25
B959	CONVERT FROM FIXED (32 to extended DFP)	CXFTR	RRE F ¤[7,9] SP Dt	20-24
B95A	CONVERT FROM LOGICAL (64 to extended DFP)	CXLGTR	RRF-e F ¤[7,9] SP Dt	20-25
B95B	CONVERT FROM LOGICAL (32 to extended DFP)	CXLFTR	RRF-e F ¤[7,9] SP Dt	20-25
B960	COMPARE AND TRAP (64)	CGRT	RRF-c GE Dc	7-149
B961	COMPARE LOGICAL AND TRAP (64)	CLGRT	RRF-c GE Dc	7-155
B972	COMPARE AND TRAP (32)	CRT	RRF-c GE Dc	7-149
B973	COMPARE LOGICAL AND TRAP (32)	CLRT	RRF-c GE Dc	7-155
B980	AND (64)	NGR	RRE C N	7-33
B981	OR (64)	OGR	RRE C N	7-310
B982	EXCLUSIVE OR (64)	XGR	RRE C N	7-254
B983	FIND LEFTMOST ONE	FLOGR	RRE C EI SP	7-262
B984	LOAD LOGICAL CHARACTER (64←8)	LLGCR	RRE EI	7-279
B985	LOAD LOGICAL HALFWORD (64←16)	LLGHR	RRE EI	7-280
B986	MULTIPLY LOGICAL (128←64)	MLGR	RRE N SP	7-306
B987	DIVIDE LOGICAL (64←128)	DLGR	RRE N ¤[9] SP IK	7-253
B988	ADD LOGICAL WITH CARRY (64)	ALCGR	RRE C N	7-31
B989	SUBTRACT LOGICAL WITH BORROW (64)	SLBGR	RRE C N	7-391
B98A	COMPARE AND SWAP AND PURGE (64)	CSPG	RRE C DE P A[1] SP $ ST R_2	10-21
B98D	EXTRACT PSW	EPSW	RRE N3 ¤[8,9]	7-261
B98E	INVALIDATE DAT TABLE ENTRY	IDTE	RRF-b U DE P A[1] SP $	10-32
B98F	COMPARE AND REPLACE DAT TABLE ENTRY	CRDTE	RRF-b ED2 P A[1] SP $	10-17
B990	TRANSLATE TWO TO TWO	TRTT	RRF-c C E2 ¤[9] A SP IC GM ST RM R_2	7-411
B991	TRANSLATE TWO TO ONE	TRTO	RRF-c C E2 ¤[9] A SP IC GM ST RM R_2	7-411
B992	TRANSLATE ONE TO TWO	TROT	RRF-c C E2 ¤[9] A SP IC GM ST RM R_2	7-411
B993	TRANSLATE ONE TO ONE	TROO	RRF-c C E2 ¤[9] A SP IC GM ST RM R_2	7-411
B994	LOAD LOGICAL CHARACTER (32←8)	LLCR	RRE EI	7-279
B995	LOAD LOGICAL HALFWORD (32←16)	LLHR	RRE EI	7-280
B996	MULTIPLY LOGICAL (64←32)	MLR	RRE N3 SP	7-305
B997	DIVIDE LOGICAL (32←64)	DLR	RRE N3 ¤[9] SP IK	7-253
B998	ADD LOGICAL WITH CARRY (32)	ALCR	RRE C N3	7-31
B999	SUBTRACT LOGICAL WITH BORROW (32)	SLBR	RRE C N3	7-391
B99A	EXTRACT PRIMARY ASN AND INSTANCE	EPAIR	RRE RA Q SO	10-24
B99B	EXTRACT SECONDARY ASN AND INSTANCE	ESAIR	RRE RA Q SO	10-25
B99D	EXTRACT AND SET EXTENDED AUTHORITY	ESEA	RRE N P	10-24
B99E	PROGRAM TRANSFER WITH INSTANCE	PTI	RRE RA Q A[1]* SP Z[6] T ¢ B	10-107
B99F	SET SECONDARY ASN WITH INSTANCE	SSAIR	RRE RA ¤[1] A[1]* Z[7] T ¢	10-125
B9A1	TEST PENDING EXTERNAL INTERRUPTION	TPEI	RRE C TE P	10-169
B9A2	PERFORM TOPOLOGY FUNCTION	PTF	RRE C CT P SP	10-89
B9AA	LOAD PAGE TABLE ENTRY ADDRESS	LPTEA	RRF-b C D2 P A[1]* SP SO R_2	10-50
B9AC	INSERT REFERENCE BITS MULTIPLE	IRBM	RRE IM P A[1]*	10-30
B9AE	RESET REFERENCE BITS MULTIPLE	RRBM	RRE RB P A[1]*	10-117
B9AF	PERFORM FRAME MANAGEMENT FUNCTION	PFMF	RRE ED1 P A[1] SP II ¢[3]	10-77
B9B0	CONVERT UTF-8 TO UTF-32	CU14	RRF-c C E3 ¤[5,9] A SP IC ST R_1 R_2	7-248
B9B1	CONVERT UTF-16 TO UTF-32	CU24	RRF-c C E3 ¤[5,9] A SP IC ST R_1 R_2	7-231

Figure B-3. Instructions Arranged by Opcode (Part 11 of 23)

Op-code	Name	Mne-monic	Characteristics												Page
B9B2	CONVERT UTF-32 TO UTF-8	CU41	RRE	C	E3	¤[5,9]	A	SP	IC			ST	R_1	R_2	7-241
B9B3	CONVERT UTF-32 TO UTF-16	CU42	RRE	C	E3	¤[5,9]	A	SP	IC			ST	R_1	R_2	7-238
B9BD	TRANSLATE AND TEST REVERSE EXTENDED	TRTRE	RRF-c	C	PE	¤[9]	A	SP	IC			ST	RM		7-404
B9BE	SEARCH STRING UNICODE	SRSTU	RRE	C	E3	¤[9]	A	SP	IC		G0		R_1	R_2	7-368
B9BF	TRANSLATE AND TEST EXTENDED	TRTE	RRF-c	C	PE	¤[9]	A	SP	IC			ST	RM		7-403
B9C8	ADD HIGH (32)	AHHHR	RRF-a	C	HW					IF					7-29
B9C9	SUBTRACT HIGH (32)	SHHHR	RRF-a	C	HW					IF					7-389
B9CA	ADD LOGICAL HIGH (32)	ALHHHR	RRF-a	C	HW										7-31
B9CB	SUBTRACT LOGICAL HIGH (32)	SLHHHR	RRF-a	C	HW										7-391
B9CD	COMPARE HIGH (32)	CHHR	RRE	C	HW										7-151
B9CF	COMPARE LOGICAL HIGH (32)	CLHHR	RRE	C	HW										7-157
B9D8	ADD HIGH (32)	AHHLR	RRF-a	C	HW					IF					7-29
B9D9	SUBTRACT HIGH (32)	SHHLR	RRF-a	C	HW					IF					7-389
B9DA	ADD LOGICAL HIGH (32)	ALHHLR	RRF-a	C	HW										7-31
B9DB	SUBTRACT LOGICAL HIGH (32)	SLHHLR	RRF-a	C	HW										7-391
B9DD	COMPARE HIGH (32)	CHLR	RRE	C	HW										7-151
B9DF	COMPARE LOGICAL HIGH (32)	CLHLR	RRE	C	HW										7-157
B9E0	LOAD HIGH ON CONDITION (32)	LOCFHR	RRF-c		L2										7-284
B9E1	POPULATION COUNT	POPCNT	RRE	C	PK										7-359
B9E2	LOAD ON CONDITION (64)	LOCGR	RRF-c		L1		A								7-284
B9E4	AND (64)	NGRK	RRF-a	C	DO										7-33
B9E6	OR (64)	OGRK	RRF-a	C	DO										7-310
B9E7	EXCLUSIVE OR (64)	XGRK	RRF-a	C	DO										7-254
B9E8	ADD (64)	AGRK	RRF-a	C	DO					IF					7-26
B9E9	SUBTRACT (64)	SGRK	RRF-a	C	DO					IF					7-388
B9EA	ADD LOGICAL (64)	ALGRK	RRF-a	C	DO										7-30
B9EB	SUBTRACT LOGICAL (64)	SLGRK	RRF-a	C	DO										7-389
B9EC	MULTIPLY (128←64)	MGRK	RRF-a		MI2			SP							7-304
B9ED	MULTIPLY SINGLE (64)	MSGRKC	RRF-a	C	MI2					IF					7-307
B9F2	LOAD ON CONDITION (32)	LOCR	RRF-c		L1		A								7-284
B9F4	AND (32)	NRK	RRF-a	C	DO										7-33
B9F6	OR (32)	ORK	RRF-a	C	DO										7-310
B9F7	EXCLUSIVE OR (32)	XRK	RRF-a	C	DO										7-254
B9F8	ADD (32)	ARK	RRF-a	C	DO					IF					7-26
B9F9	SUBTRACT (32)	SRK	RRF-a	C	DO					IF					7-387
B9FA	ADD LOGICAL (32)	ALRK	RRF-a	C	DO										7-30
B9FB	SUBTRACT LOGICAL (32)	SLRK	RRF-a	C	DO										7-389
B9FD	MULTIPLY SINGLE (32)	MSRKC	RRF-a	C	MI2					IF					7-307
BA	COMPARE AND SWAP (32)	CS	RS-a	C		¤[9]	A	SP		$		ST		B_2	7-144
BB	COMPARE DOUBLE AND SWAP (32)	CDS	RS-a	C		¤[9]	A	SP		$		ST		B_2	7-144
BD	COMPARE LOGICAL CHAR. UNDER MASK (low)	CLM	RS-b	C			A							B_2	7-157
BE	STORE CHARACTERS UNDER MASK (low)	STCM	RS-b				A					ST		B_2	7-378
BF	INSERT CHARACTERS UNDER MASK (low)	ICM	RS-b	C			A							B_2	7-262
C00	LOAD ADDRESS RELATIVE LONG	LARL	RIL-b		N3										7-267
C01	LOAD IMMEDIATE (64←32)	LGFI	RIL-a		EI										7-264
C04	BRANCH RELATIVE ON CONDITION LONG	BRCL	RIL-c		N3	¤[10]						B			7-46
C05	BRANCH RELATIVE AND SAVE LONG	BRASL	RIL-b		N3	¤[9]						B			7-46
C06	EXCLUSIVE OR IMMEDIATE (high)	XIHF	RIL-a	C	EI										7-256
C07	EXCLUSIVE OR IMMEDIATE (low)	XILF	RIL-a	C	EI										7-256
C08	INSERT IMMEDIATE (high)	IIHF	RIL-a		EI										7-263
C09	INSERT IMMEDIATE (low)	IILF	RIL-a		EI										7-263
C0A	AND IMMEDIATE (high)	NIHF	RIL-a	C	EI										7-34
C0B	AND IMMEDIATE (low)	NILF	RIL-a	C	EI										7-35
C0C	OR IMMEDIATE (high)	OIHF	RIL-a	C	EI										7-312
C0D	OR IMMEDIATE (low)	OILF	RIL-a	C	EI										7-312

Figure B-3. Instructions Arranged by Opcode (Part 12 of 23)

Op-code	Name	Mne-monic	Characteristics										Page
C0E	LOAD LOGICAL IMMEDIATE (high)	LLIHF	RIL-a		EI								7-281
C0F	LOAD LOGICAL IMMEDIATE (low)	LLILF	RIL-a		EI								7-281
C20	MULTIPLY SINGLE IMMEDIATE (64←32)	MSGFI	RIL-a		GE								7-307
C21	MULTIPLY SINGLE IMMEDIATE (32)	MSFI	RIL-a		GE								7-307
C24	SUBTRACT LOGICAL IMMEDIATE (64←32)	SLGFI	RIL-a	C	EI								7-390
C25	SUBTRACT LOGICAL IMMEDIATE (32)	SLFI	RIL-a	C	EI								7-390
C28	ADD IMMEDIATE (64←32)	AGFI	RIL-a	C	EI			IF					7-27
C29	ADD IMMEDIATE (32)	AFI	RIL-a	C	EI			IF					7-27
C2A	ADD LOGICAL IMMEDIATE (64←32)	ALGFI	RIL-a	C	EI								7-30
C2B	ADD LOGICAL IMMEDIATE (32)	ALFI	RIL-a	C	EI								7-30
C2C	COMPARE IMMEDIATE (64←32)	CGFI	RIL-a	C	EI								7-135
C2D	COMPARE IMMEDIATE (32)	CFI	RIL-a	C	EI								7-134
C2E	COMPARE LOGICAL IMMEDIATE (64←32)	CLGFI	RIL-a	C	EI								7-152
C2F	COMPARE LOGICAL IMMEDIATE (32)	CLFI	RIL-a	C	EI								7-152
C42	LOAD LOGICAL HALFWORD RELATIVE LONG (32←16)	LLHRL	RIL-b		GE		A*						7-280
C44	LOAD HALFWORD RELATIVE LONG (64←16)	LGHRL	RIL-b		GE		A*						7-276
C45	LOAD HALFWORD RELATIVE LONG (32←16)	LHRL	RIL-b		GE		A*						7-276
C46	LOAD LOGICAL HALFWORD RELATIVE LONG (64←16)	LLGHRL	RIL-b		GE		A*						7-280
C47	STORE HALFWORD RELATIVE LONG (16)	STHRL	RIL-b		GE		A*				ST		7-384
C48	LOAD RELATIVE LONG (64)	LGRL	RIL-b		GE		A*	SP					7-264
C4B	STORE RELATIVE LONG (64)	STGRL	RIL-b		GE		A*	SP			ST		7-377
C4C	LOAD RELATIVE LONG (64←32)	LGFRL	RIL-b		GE		A*	SP					7-264
C4D	LOAD RELATIVE LONG (32)	LRL	RIL-b		GE		A	SP					7-264
C4E	LOAD LOGICAL RELATIVE LONG (64←32)	LLGFRL	RIL-b		GE		A*	SP					7-278
C4F	STORE RELATIVE LONG (32)	STRL	RIL-b		GE		A*	SP			ST		7-377
C5	BRANCH PREDICTION RELATIVE PRELOAD	BPRP	MII		EH	¤[9]							7-43
C60	EXECUTE RELATIVE LONG	EXRL	RIL-b		XX	¤[9]	AI*			EX			7-256
C62	PREFETCH DATA RELATIVE LONG	PFDRL	RIL-c		GE	¤[9,11]							7-359
C64	COMPARE HALFWORD RELATIVE LONG (64←16)	CGHRL	RIL-b	C	GE		A*						7-150
C65	COMPARE HALFWORD RELATIVE LONG (32←16)	CHRL	RIL-b	C	GE		A*						7-150
C66	COMPARE LOGICAL RELATIVE LONG (64←16)	CLGHRL	RIL-b	C	GE		A*						7-153
C67	COMPARE LOGICAL RELATIVE LONG (32←16)	CLHRL	RIL-b	C	GE		A*						7-153
C68	COMPARE RELATIVE LONG (64)	CGRL	RIL-b	C	GE		A*	SP					7-135
C6A	COMPARE LOGICAL RELATIVE LONG (64)	CLGRL	RIL-b	C	GE		A*	SP					7-153
C6C	COMPARE RELATIVE LONG (64←32)	CGFRL	RIL-b	C	GE		A*	SP					7-135
C6D	COMPARE RELATIVE LONG (32)	CRL	RIL-b	C	GE		A*	SP					7-135
C6E	COMPARE LOGICAL RELATIVE LONG (64←32)	CLGFRL	RIL-b	C	GE		A*	SP					7-153
C6F	COMPARE LOGICAL RELATIVE LONG (32)	CLRL	RIL-b	C	GE		A*	SP					7-153
C7	BRANCH PREDICTION PRELOAD	BPP	SMI		EH	¤[9]							7-43
C80	MOVE WITH OPTIONAL SPECIFICATIONS	MVCOS	SSF	C	MO	Q	A		SO	G0		ST	B† B‡	10-68
C81	EXTRACT CPU TIME	ECTG	SSF		ET	¤[8,9]	A			GM		R_3	B_1 B_2	7-260
C82	COMPARE AND SWAP AND STORE	CSST	SSF	C	CS	¤[1]	A	SP		$ GM		ST	B_1 B_2	7-146
C84	LOAD PAIR DISJOINT (32)	LPD	SSF	C	IA	¤[9]	A	SP					B_1 B_2	7-285
C85	LOAD PAIR DISJOINT (64)	LPDG	SSF	C	IA	¤[9]	A	SP					B_1 B_2	7-285
CC6	BRANCH RELATIVE ON COUNT HIGH (32)	BRCTH	RIL-b		HW	¤[9]					B			7-47
CC8	ADD IMMEDIATE HIGH (32)	AIH	RIL-a	C	HW				IF					7-29
CCA	ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (32)	ALSIH	RIL-a	C	HW									7-33
CCB	ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (32)	ALSIHN	RIL-a		HW									7-33
CCD	COMPARE IMMEDIATE HIGH (32)	CIH	RIL-a	C	HW									7-151
CCF	COMPARE LOGICAL IMMEDIATE HIGH (32)	CLIH	RIL-a	C	HW									7-158
D0	TRANSLATE AND TEST REVERSE	TRTR	SS-a	C	E3	¤[9]	A			GM			B_1 B_2	7-408
D1	MOVE NUMERICS	MVN	SS-a			¤[9]	A					ST	B_1 B_2	7-301

Figure B-3. Instructions Arranged by Opcode (Part 13 of 23)

Op-code	Name	Mne-monic	Characteristics												Page
D2	MOVE (character)	MVC	SS-a			¤[9]	A					ST	B_1	B_2	7-289
D3	MOVE ZONES	MVZ	SS-a			¤[9]	A					ST	B_1	B_2	7-303
D4	AND (character)	NC	SS-a	C		¤[9]	A					ST	B_1	B_2	7-34
D5	COMPARE LOGICAL (character)	CLC	SS-a	C		¤[9]	A						B_1	B_2	7-152
D6	OR (character)	OC	SS-a	C		¤[9]	A					ST	B_1	B_2	7-311
D7	EXCLUSIVE OR (character)	XC	SS-a	C		¤[9]	A					ST	B_1	B_2	7-255
D9	MOVE WITH KEY	MVCK	SS-d	C		Q	A					ST	B_1	B_2	10-67
DA	MOVE TO PRIMARY	MVCP	SS-d	C		Q	A		SO	¢		ST			10-64
DB	MOVE TO SECONDARY	MVCS	SS-d	C		Q	A		SO	¢		ST			10-64
DC	TRANSLATE	TR	SS-a			¤[9]	A					ST	B_1	B_2	7-402
DD	TRANSLATE AND TEST	TRT	SS-a	C		¤[9]	A				GM		B_1	B_2	7-403
DE	EDIT	ED	SS-a	C		¤[9]	A		Dg			ST	B_1	B_2	8-8
DF	EDIT AND MARK	EDMK	SS-a	C		¤[9]	A		Dg		G1	ST	B_1	B_2	8-11
E1	PACK UNICODE	PKU	SS-f		E2	¤[9]	A	SP				ST	B_1	B_2	7-314
E2	UNPACK UNICODE	UNPKU	SS-a	C	E2	¤[9]	A	SP				ST	B_1	B_2	7-418
E302	LOAD AND TEST (64)	LTG	RXY-a	C	EI		A							B_2	7-271
E303	LOAD REAL ADDRESS (64)	LRAG	RXY-a	C	N	P	A[1]*							BP	10-56
E304	LOAD (64)	LG	RXY-a		N		A							B_2	7-264
E306	CONVERT TO BINARY (32)	CVBY	RXY-a		LD	¤[9]	A			Dg	IK			B_2	7-230
E308	ADD (64)	AG	RXY-a	C	N		A			IF				B_2	7-27
E309	SUBTRACT (64)	SG	RXY-a	C	N		A			IF				B_2	7-388
E30A	ADD LOGICAL (64)	ALG	RXY-a	C	N		A							B_2	7-30
E30B	SUBTRACT LOGICAL (64)	SLG	RXY-a	C	N		A							B_2	7-390
E30C	MULTIPLY SINGLE (64)	MSG	RXY-a		N		A							B_2	7-307
E30D	DIVIDE SINGLE (64)	DSG	RXY-a		N	¤[9]	A	SP			IK			B_2	7-254
E30E	CONVERT TO BINARY (64)	CVBG	RXY-a		N	¤[9]	A			Dg	IK			B_2	7-230
E30F	LOAD REVERSED (64)	LRVG	RXY-a		N		A							B_2	7-287
E312	LOAD AND TEST (32)	LT	RXY-a	C	EI		A							B_2	7-271
E313	LOAD REAL ADDRESS (32)	LRAY	RXY-a	C	LD	P	A[1]*		SO					BP	10-56
E314	LOAD (64←32)	LGF	RXY-a		N		A							B_2	7-264
E315	LOAD HALFWORD (64←16)	LGH	RXY-a		N		A							B_2	7-276
E316	LOAD LOGICAL (64←32)	LLGF	RXY-a		N		A							B_2	7-278
E317	LOAD LOGICAL THIRTY ONE BITS (64←31)	LLGT	RXY-a		N		A							B_2	7-282
E318	ADD (64←32)	AGF	RXY-a	C	N		A			IF				B_2	7-27
E319	SUBTRACT (64←32)	SGF	RXY-a	C	N		A			IF				B_2	7-388
E31A	ADD LOGICAL (64←32)	ALGF	RXY-a	C	N		A							B_2	7-30
E31B	SUBTRACT LOGICAL (64←32)	SLGF	RXY-a	C	N		A							B_2	7-390
E31C	MULTIPLY SINGLE (64←32)	MSGF	RXY-a		N		A							B_2	7-307
E31D	DIVIDE SINGLE (64←32)	DSGF	RXY-a		N	¤[9]	A	SP			IK			B_2	7-254
E31E	LOAD REVERSED (32)	LRV	RXY-a		N3		A							B_2	7-287
E31F	LOAD REVERSED (16)	LRVH	RXY-a		N3		A							B_2	7-287
E320	COMPARE (64)	CG	RXY-a	C	N		A							B_2	7-134
E321	COMPARE LOGICAL (64)	CLG	RXY-a	C	N		A							B_2	7-152
E324	STORE (64)	STG	RXY-a		N		A					ST		B_2	7-377
E325	NONTRANSACTIONAL STORE (64)	NTSTG	RXY-a		TX	¤[9]	A	SP				ST		B_2	7-310
E326	CONVERT TO DECIMAL (32)	CVDY	RXY-a		LD	¤[9]	A					ST		B_2	7-231
E32A	LOAD AND ZERO RIGHTMOST BYTE (64)	LZRG	RXY-a		LZ		A							B_2	7-271
E32E	CONVERT TO DECIMAL (64)	CVDG	RXY-a		N	¤[9]	A					ST		B_2	7-231
E32F	STORE REVERSED (64)	STRVG	RXY-a		N		A					ST		B_2	7-387
E330	COMPARE (64←32)	CGF	RXY-a	C	N		A							B_2	7-134
E331	COMPARE LOGICAL (64←32)	CLGF	RXY-a	C	N		A							B_2	7-152
E332	LOAD AND TEST (64←32)	LTGF	RXY-a	C	GE		A							B_2	7-271
E334	COMPARE HALFWORD (64←16)	CGH	RXY-a	C	GE		A							B_2	7-150
E336	PREFETCH DATA	PFD	RXY-b		GE	¤[9,11]								B_2	7-359
E338	ADD HALFWORD (64←16)	AGH	RXY-a	C	MI2		A			IF				B_2	7-28

Figure B-3. Instructions Arranged by Opcode (Part 14 of 23)

Op-code	Name	Mne-monic	Characteristics											Page
E339	SUBTRACT HALFWORD (64←16)	SGH	RXY-a	C	MI2		A		IF				B_2	7-388
E33A	LOAD LOGICAL AND ZERO RIGHTMOST BYTE (64←32)	LLZRGF	RXY-a		LZ		A						B_2	7-279
E33B	LOAD AND ZERO RIGHTMOST BYTE (32)	LZRF	RXY-a		LZ		A						B_2	7-271
E33C	MULTIPLY HALFWORD (64←16)	MGH	RXY-a		MI2		A						B_2	7-305
E33E	STORE REVERSED (32)	STRV	RXY-a		N3		A				ST		B_2	7-387
E33F	STORE REVERSED (16)	STRVH	RXY-a		N3		A				ST		B_2	7-387
E346	BRANCH ON COUNT (64)	BCTG	RXY-a		N	α[9]					B			7-41
E347	BRANCH INDIRECT ON CONDITION	BIC	RXY-b		MI2	α[9]	A				B		B_2	7-39
E348	LOAD LOGICAL AND SHIFT GUARDED (64←32)	LLGFSG	RXY-a		GF	α[12]	A	SP			B	ST	B_2	7-274
E349	STORE GUARDED STORAGE CONTROLS	STGSC	RXY-a		GF	α[1]	A		SO			ST	B_2	7-383
E34C	LOAD GUARDED (64)	LGG	RXY-a		GF	α[12]	A	SP			B	ST	B_2	7-274
E34D	LOAD GUARDED STORAGE CONTROLS	LGSC	RXY-a		GF	α[1]	A		SO				B_2	7-275
E350	STORE (32)	STY	RXY-a		LD		A					ST	B_2	7-377
E351	MULTIPLY SINGLE (32)	MSY	RXY-a		LD		A						B_2	7-307
E353	MULTIPLY SINGLE (32)	MSC	RXY-a	C	MI2		A		IF				B_2	7-307
E354	AND (32)	NY	RXY-a	C	LD		A						B_2	7-33
E355	COMPARE LOGICAL (32)	CLY	RXY-a	C	LD		A						B_2	7-152
E356	OR (32)	OY	RXY-a	C	LD		A						B_2	7-310
E357	EXCLUSIVE OR (32)	XY	RXY-a	C	LD		A						B_2	7-254
E358	LOAD (32)	LY	RXY-a		LD		A						B_2	7-264
E359	COMPARE (32)	CY	RXY-a	C	LD		A						B_2	7-134
E35A	ADD (32)	AY	RXY-a	C	LD		A		IF				B_2	7-27
E35B	SUBTRACT (32)	SY	RXY-a	C	LD		A		IF				B_2	7-388
E35C	MULTIPLY (64←32)	MFY	RXY-a		GE		A	SP					B_2	7-304
E35E	ADD LOGICAL (32)	ALY	RXY-a	C	LD		A						B_2	7-30
E35F	SUBTRACT LOGICAL (32)	SLY	RXY-a	C	LD		A						B_2	7-390
E370	STORE HALFWORD (16)	STHY	RXY-a		LD		A				ST		B_2	7-384
E371	LOAD ADDRESS	LAY	RXY-a		LD									7-266
E372	STORE CHARACTER	STCY	RXY-a		LD		A				ST		B_2	7-378
E373	INSERT CHARACTER	ICY	RXY-a		LD		A						B_2	7-262
E375	LOAD ADDRESS EXTENDED	LAEY	RXY-a		GE	α[6]						U_1	BP	7-266
E376	LOAD BYTE (32←8)	LB	RXY-a		LD		A							7-272
E377	LOAD BYTE (64←8)	LGB	RXY-a		LD		A							7-272
E378	LOAD HALFWORD (32←16)	LHY	RXY-a		LD		A						B_2	7-276
E379	COMPARE HALFWORD (32←16)	CHY	RXY-a	C	LD		A						B_2	7-150
E37A	ADD HALFWORD (32←16)	AHY	RXY-a	C	LD		A		IF				B_2	7-28
E37B	SUBTRACT HALFWORD (32←16)	SHY	RXY-a	C	LD		A		IF				B_2	7-388
E37C	MULTIPLY HALFWORD (32←16)	MHY	RXY-a		GE		A						B_2	7-305
E380	AND (64)	NG	RXY-a	C	N		A						B_2	7-33
E381	OR (64)	OG	RXY-a	C	N		A						B_2	7-311
E382	EXCLUSIVE OR (64)	XG	RXY-a	C	N		A						B_2	7-254
E383	MULTIPLY SINGLE (64)	MSGC	RXY-a	C	MI2		A		IF				B_2	7-307
E384	MULTIPLY (128←64)	MG	RXY-a		MI2		A	SP					B_2	7-304
E385	LOAD AND TRAP (64)	LGAT	RXY-a		LT		A		Dc				B_2	7-271
E386	MULTIPLY LOGICAL (128←64)	MLG	RXY-a		N		A	SP					B_2	7-306
E387	DIVIDE LOGICAL (64←128)	DLG	RXY-a		N	α[9]	A	SP	IK				B_2	7-253
E388	ADD LOGICAL WITH CARRY (64)	ALCG	RXY-a	C	N		A						B_2	7-31
E389	SUBTRACT LOGICAL WITH BORROW (64)	SLBG	RXY-a	C	N		A						B_2	7-391
E38E	STORE PAIR TO QUADWORD	STPQ	RXY-a		N	α[9]	A	SP			ST		B_2	7-386
E38F	LOAD PAIR FROM QUADWORD (64&64←128)	LPQ	RXY-a		N	α[9]	A	SP					B_2	7-286
E390	LOAD LOGICAL CHARACTER (64←8)	LLGC	RXY-a		N		A						B_2	7-279
E391	LOAD LOGICAL HALFWORD (64←16)	LLGH	RXY-a		N		A						B_2	7-280
E394	LOAD LOGICAL CHARACTER (32←8)	LLC	RXY-a		EI		A						B_2	7-279
E395	LOAD LOGICAL HALFWORD (32←16)	LLH	RXY-a		EI		A						B_2	7-280

Figure B-3. Instructions Arranged by Opcode (Part 15 of 23)

Op-code	Name	Mne-monic	Characteristics																		Page
E396	MULTIPLY LOGICAL (64←32)	ML	RXY-a		N3		A	SP										B_2			7-306
E397	DIVIDE LOGICAL (32←64)	DL	RXY-a		N3	α[9]	A	SP			IK							B_2			7-253
E398	ADD LOGICAL WITH CARRY (32)	ALC	RXY-a	C	N3		A											B_2			7-31
E399	SUBTRACT LOGICAL WITH BORROW (32)	SLB	RXY-a	C	N3		A											B_2			7-391
E39C	LOAD LOGICAL THIRTY ONE BITS AND TRAP (64←31)	LLGTAT	RXY-a		LT		A			Dc								B_2			7-282
E39D	LOAD LOGICAL AND TRAP (64←32)	LLGFAT	RXY-a		LT		A			Dc								B_2			7-279
E39F	LOAD AND TRAP (32L←32)	LAT	RXY-a		LT		A			Dc								B_2			7-271
E3C0	LOAD BYTE HIGH (32←8)	LBH	RXY-a		HW		A											B_2			7-272
E3C2	LOAD LOGICAL CHARACTER HIGH (32←8)	LLCH	RXY-a		HW		A											B_2			7-280
E3C3	STORE CHARACTER HIGH (8)	STCH	RXY-a		HW		A									ST		B_2			7-378
E3C4	LOAD HALFWORD HIGH (32←16)	LHH	RXY-a		HW		A											B_2			7-277
E3C6	LOAD LOGICAL HALFWORD HIGH (32←16)	LLHH	RXY-a		HW		A											B_2			7-281
E3C7	STORE HALFWORD HIGH (16)	STHH	RXY-a		HW		A									ST		B_2			7-384
E3C8	LOAD HIGH AND TRAP (32H←32)	LFHAT	RXY-a		LT		A			Dc								B_2			7-278
E3CA	LOAD HIGH (32)	LFH	RXY-a		HW		A											B_2			7-278
E3CB	STORE HIGH (32)	STFH	RXY-a		HW		A									ST		B_2			7-384
E3CD	COMPARE HIGH (32)	CHF	RXY-a	C	HW		A											B_2			7-151
E3CF	COMPARE LOGICAL HIGH (32)	CLHF	RXY-a	C	HW		A											B_2			7-157
E500	LOAD ADDRESS SPACE PARAMETERS	LASP	SSE	C		P	A[1]	SP	SO									B_1			10-41
E501	TEST PROTECTION	TPROT	SSE	C		P	A[1⋆]											B_1			10-170
E502	STORE REAL ADDRESS	STRAG	SSE		N	P	A[1]	SP								ST		B_1	BP		10-139
E50E	MOVE WITH SOURCE KEY	MVCSK	SSE			Q	A					GM				ST		B_1	B_2		10-71
E50F	MOVE WITH DESTINATION KEY	MVCDK	SSE			Q	A					GM				ST		B_1	B_2		10-66
E544	MOVE (16←16)	MVHHI	SIL		GE		A									ST		B_1			7-289
E548	MOVE (64←16)	MVGHI	SIL		GE		A									ST		B_1			7-289
E54C	MOVE (32←16)	MVHI	SIL		GE		A									ST		B_1			7-289
E554	COMPARE HALFWORD IMMEDIATE (16←16)	CHHSI	SIL	C	GE		A											B_1			7-150
E555	COMPARE LOGICAL IMMEDIATE (16←16)	CLHHSI	SIL	C	GE		A											B_1			7-152
E558	COMPARE HALFWORD IMMEDIATE (64←16)	CGHSI	SIL	C	GE		A											B_1			7-150
E559	COMPARE LOGICAL IMMEDIATE (64←16)	CLGHSI	SIL	C	GE		A											B_1			7-152
E55C	COMPARE HALFWORD IMMEDIATE (32←16)	CHSI	SIL	C	GE		A											B_1			7-150
E55D	COMPARE LOGICAL IMMEDIATE (32←16)	CLFHSI	SIL	C	GE		A											B_1			7-152
E560	TRANSACTION BEGIN (nonconstrained)	TBEGIN	SIL	C	TX	α[9]	A	SP	SO		$	EX				ST					7-395
E561	TRANSACTION BEGIN (constrained)	TBEGINC	SIL	C	CX	α[9]		SP	SO		$	EX									7-399
E634	VECTOR PACK ZONED	VPKZ	VSI		VD	α[7,9]	A	SP	Dv									B_2			25-12
E635	VECTOR LOAD RIGHTMOST WITH LENGTH	VLRL	VSI		VD	α[7,9]	A	SP	Dv									B_2			21-9
E637	VECTOR LOAD RIGHTMOST WITH LENGTH	VLRLR	VRS-d		VD	α[7,9]	A		Dv									B_2			21-9
E63C	VECTOR UNPACK ZONED	VUPKZ	VSI		VD	α[7,9]	A	SP	Dv							ST		B_2			25-20
E63D	VECTOR STORE RIGHTMOST WITH LENGTH	VSTRL	VSI		VD	α[7,9]	A	SP	Dv							ST		B_2			21-19
E63F	VECTOR STORE RIGHTMOST WITH LENGTH	VSTRLR	VRS-d		VD	α[7,9]	A		Dv							ST		B_2			21-19
E649	VECTOR LOAD IMMEDIATE DECIMAL	VLIP	VRI-h		VD	α[7,9]			Dv	Dg											25-9
E650	VECTOR CONVERT TO BINARY	VCVB	VRR-i	C*	VD	α[7,9]			Dv	Dg		IF									25-5
E652	VECTOR CONVERT TO BINARY	VCVBG	VRR-i	C*	VD	α[7,9]			Dv	Dg		IF									25-5
E658	VECTOR CONVERT TO DECIMAL	VCVD	VRI-i	C*	VD	α[7,9]		SP	Dv		DF										25-6
E659	VECTOR SHIFT AND ROUND DECIMAL	VSRP	VRI-g	C*	VD	α[7,9]		SP	Dv	Dg	DF										25-17
E65A	VECTOR CONVERT TO DECIMAL	VCVDG	VRI-i	C*	VD	α[7,9]		SP	Dv		DF										25-6
E65B	VECTOR PERFORM SIGN OPERATION DECIMAL	VPSOP	VRI-g	C*	VD	α[7,9]		SP	Dv	Dg	DF										25-13
E65F	VECTOR TEST DECIMAL	VTP	VRR-g	C	VD	α[7,9]			Dv												25-20
E671	VECTOR ADD DECIMAL	VAP	VRI-f	C*	VD	α[7,9]		SP	Dv	Dg	DF										25-3
E673	VECTOR SUBTRACT DECIMAL	VSP	VRI-f	C*	VD	α[7,9]		SP	Dv	Dg	DF										25-18
E677	VECTOR COMPARE DECIMAL	VCP	VRR-h	C	VD	α[7,9]			Dv	Dg											25-4
E678	VECTOR MULTIPLY DECIMAL	VMP	VRI-f	C*	VD	α[7,9]		SP	Dv	Dg	DF										25-10
E679	VECTOR MULTIPLY AND SHIFT DECIMAL	VMSP	VRI-f	C*	VD	α[7,9]		SP	Dv	Dg	DF										25-11
E67A	VECTOR DIVIDE DECIMAL	VDP	VRI-f	C*	VD	α[7,9]		SP	Dv	Dg	DF	DK									25-8

Figure B-3. Instructions Arranged by Opcode (Part 16 of 23)

Op-code	Name	Mne-monic	Characteristics											Page
E67B	VECTOR REMAINDER DECIMAL	VRP	VRI-f	C*	VD	$\alpha^{7,9}$		SP	Dv	Dg	DF	DK			25-14
E67E	VECTOR SHIFT AND DIVIDE DECIMAL	VSDP	VRI-f	C*	VD	$\alpha^{7,9}$		SP	Dv	Dg	DF	DK			25-16
E700	VECTOR LOAD ELEMENT (8)	VLEB	VRX		VF	$\alpha^{7,9}$	A	SP	Dv					B_2	21-6
E701	VECTOR LOAD ELEMENT (16)	VLEH	VRX		VF	$\alpha^{7,9}$	A	SP	Dv					B_2	21-7
E702	VECTOR LOAD ELEMENT (64)	VLEG	VRX		VF	$\alpha^{7,9}$	A	SP	Dv					B_2	21-7
E703	VECTOR LOAD ELEMENT (32)	VLEF	VRX		VF	$\alpha^{7,9}$	A	SP	Dv					B_2	21-7
E704	VECTOR LOAD LOGICAL ELEMENT AND ZERO	VLLEZ	VRX		VF	$\alpha^{7,9}$	A	SP	Dv					B_2	21-8
E705	VECTOR LOAD AND REPLICATE	VLREP	VRX		VF	$\alpha^{7,9}$	A	SP	Dv					B_2	21-6
E706	VECTOR LOAD	VL	VRX		VF	$\alpha^{7,9}$	A		Dv					B_2	21-6
E707	VECTOR LOAD TO BLOCK BOUNDARY	VLBB	VRX		VF	$\alpha^{7,9}$	A	SP	Dv					B_2	21-10
E708	VECTOR STORE ELEMENT (8)	VSTEB	VRX		VF	$\alpha^{7,9}$	A	SP	Dv				ST	B_2	21-18
E709	VECTOR STORE ELEMENT (16)	VSTEH	VRX		VF	$\alpha^{7,9}$	A	SP	Dv				ST	B_2	21-18
E70A	VECTOR STORE ELEMENT (64)	VSTEG	VRX		VF	$\alpha^{7,9}$	A	SP	Dv				ST	B_2	21-18
E70B	VECTOR STORE ELEMENT (32)	VSTEF	VRX		VF	$\alpha^{7,9}$	A	SP	Dv				ST	B_2	21-18
E70E	VECTOR STORE	VST	VRX		VF	$\alpha^{7,9}$	A		Dv				ST	B_2	21-18
E712	VECTOR GATHER ELEMENT (64)	VGEG	VRV		VF	$\alpha^{7,9}$	A	SP	Dv					B_2	21-4
E713	VECTOR GATHER ELEMENT (32)	VGEF	VRV		VF	$\alpha^{7,9}$	A	SP	Dv					B_2	21-4
E71A	VECTOR SCATTER ELEMENT (64)	VSCEG	VRV		VF	$\alpha^{7,9}$	A	SP	Dv				ST	B_2	21-16
E71B	VECTOR SCATTER ELEMENT (32)	VSCEF	VRV		VF	$\alpha^{7,9}$	A	SP	Dv				ST	B_2	21-16
E721	VECTOR LOAD GR FROM VR ELEMENT	VLGV	VRS-c		VF	$\alpha^{7,9}$		SP	Dv						21-7
E722	VECTOR LOAD VR ELEMENT FROM GR	VLVG	VRS-b		VF	$\alpha^{7,9}$		SP	Dv						21-11
E727	LOAD COUNT TO BLOCK BOUNDARY	LCBB	RXE	C	VF			SP							7-273
E730	VECTOR ELEMENT SHIFT LEFT	VESL	VRS-a		VF	$\alpha^{7,9}$		SP	Dv						22-23
E733	VECTOR ELEMENT ROTATE LEFT LOGICAL	VERLL	VRS-a		VF	$\alpha^{7,9}$		SP	Dv						22-21
E736	VECTOR LOAD MULTIPLE	VLM	VRS-a		VF	$\alpha^{7,9}$	A	SP	Dv					B_2	21-9
E737	VECTOR LOAD WITH LENGTH	VLL	VRS-b		VF	$\alpha^{7,9}$	A		Dv					B_2	21-11
E738	VECTOR ELEMENT SHIFT RIGHT LOGICAL	VESRL	VRS-a		VF	$\alpha^{7,9}$		SP	Dv						22-24
E73A	VECTOR ELEMENT SHIFT RIGHT ARITHMETIC	VESRA	VRS-a		VF	$\alpha^{7,9}$		SP	Dv						22-23
E73E	VECTOR STORE MULTIPLE	VSTM	VRS-a		VF	$\alpha^{7,9}$	A	SP	Dv				ST	B_2	21-19
E73F	VECTOR STORE WITH LENGTH	VSTL	VRS-b		VF	$\alpha^{7,9}$	A		Dv				ST	B_2	21-20
E740	VECTOR LOAD ELEMENT IMMEDIATE (8)	VLEIB	VRI-a		VF	$\alpha^{7,9}$		SP	Dv						21-7
E741	VECTOR LOAD ELEMENT IMMEDIATE (16)	VLEIH	VRI-a		VF	$\alpha^{7,9}$		SP	Dv						21-7
E742	VECTOR LOAD ELEMENT IMMEDIATE (64)	VLEIG	VRI-a		VF	$\alpha^{7,9}$		SP	Dv						21-7
E743	VECTOR LOAD ELEMENT IMMEDIATE (32)	VLEIF	VRI-a		VF	$\alpha^{7,9}$		SP	Dv						21-7
E744	VECTOR GENERATE BYTE MASK	VGBM	VRI-a		VF	$\alpha^{7,9}$			Dv						21-5
E745	VECTOR REPLICATE IMMEDIATE	VREPI	VRI-a		VF	$\alpha^{7,9}$		SP	Dv						21-16
E746	VECTOR GENERATE MASK	VGM	VRI-b		VF	$\alpha^{7,9}$		SP	Dv						21-5
E74A	VECTOR FP TEST DATA CLASS IMMEDIATE	VFTCI	VRI-e	C	VF	$\alpha^{7,9}$		SP	Dv						24-46
E74D	VECTOR REPLICATE	VREP	VRI-c		VF	$\alpha^{7,9}$		SP	Dv						21-15
E750	VECTOR POPULATION COUNT	VPOPCT	VRR-a		VF	$\alpha^{7,9}$		SP	Dv						22-21
E752	VECTOR COUNT TRAILING ZEROS	VCTZ	VRR-a		VF	$\alpha^{7,9}$		SP	Dv						22-10
E753	VECTOR COUNT LEADING ZEROS	VCLZ	VRR-a		VF	$\alpha^{7,9}$		SP	Dv						22-10
E756	VECTOR LOAD	VLR	VRR-a		VF	$\alpha^{7,9}$			Dv						21-6
E75C	VECTOR ISOLATE STRING	VISTR	VRR-a	C*	VF	$\alpha^{7,9}$		SP	Dv						23-5
E75F	VECTOR SIGN EXTEND TO DOUBLEWORD	VSEG	VRR-a		VF	$\alpha^{7,9}$		SP	Dv						21-17
E760	VECTOR MERGE LOW	VMRL	VRR-c		VF	$\alpha^{7,9}$		SP	Dv						21-12
E761	VECTOR MERGE HIGH	VMRH	VRR-c		VF	$\alpha^{7,9}$		SP	Dv						21-12
E762	VECTOR LOAD VR FROM GRS DISJOINT	VLVGP	VRR-f		VF	$\alpha^{7,9}$			Dv						21-11
E764	VECTOR SUM ACROSS WORD	VSUM	VRR-c		VF	$\alpha^{7,9}$		SP	Dv						22-29
E765	VECTOR SUM ACROSS DOUBLEWORD	VSUMG	VRR-c		VF	$\alpha^{7,9}$		SP	Dv						22-28
E766	VECTOR CHECKSUM	VCKSM	VRR-c		VF	$\alpha^{7,9}$			Dv						22-6
E767	VECTOR SUM ACROSS QUADWORD	VSUMQ	VRR-c		VF	$\alpha^{7,9}$		SP	Dv						22-29
E768	VECTOR AND	VN	VRR-c		VF	$\alpha^{7,9}$			Dv						22-5
E769	VECTOR AND WITH COMPLEMENT	VNC	VRR-c		VF	$\alpha^{7,9}$			Dv						22-5
E76A	VECTOR OR	VO	VRR-c		VF	$\alpha^{7,9}$			Dv						22-20

Figure B-3. Instructions Arranged by Opcode (Part 17 of 23)

Op-code	Name	Mne-monic	Characteristics									Page
E76B	VECTOR NOR	VNO	VRR-c	VF	α[7,9]		Dv					22-20
E76C	VECTOR NOT EXCLUSIVE OR	VNX	VRR-c	V1	α[7,9]		Dv					22-20
E76D	VECTOR EXCLUSIVE OR	VX	VRR-c	VF	α[7,9]		Dv					22-11
E76E	VECTOR NAND	VNN	VRR-c	V1	α[7,9]		DV					22-20
E76F	VECTOR OR WITH COMPLEMENT	VOC	VRR-c	V1	α[7,9]		Dv					22-21
E770	VECTOR ELEMENT SHIFT LEFT	VESLV	VRR-c	VF	α[7,9]	SP	Dv					22-23
E772	VECTOR ELEMENT ROTATE AND INSERT UNDER MASK	VERIM	VRI-d	VF	α[7,9]	SP	Dv					22-22
E773	VECTOR ELEMENT ROTATE LEFT LOGICAL	VERLLV	VRR-c	VF	α[7,9]	SP	Dv					22-21
E774	VECTOR SHIFT LEFT	VSL	VRR-c	VF	α[7,9]		Dv					22-25
E775	VECTOR SHIFT LEFT BY BYTE	VSLB	VRR-c	VF	α[7,9]		Dv					22-25
E777	VECTOR SHIFT LEFT DOUBLE BY BYTE	VSLDB	VRI-d	VF	α[7,9]		Dv					22-25
E778	VECTOR ELEMENT SHIFT RIGHT LOGICAL	VESRLV	VRR-c	VF	α[7,9]	SP	Dv					22-24
E77A	VECTOR ELEMENT SHIFT RIGHT ARITHMETIC	VESRAV	VRR-c	VF	α[7,9]	SP	Dv					22-23
E77C	VECTOR SHIFT RIGHT LOGICAL	VSRL	VRR-c	VF	α[7,9]		Dv					22-26
E77D	VECTOR SHIFT RIGHT LOGICAL BY BYTE	VSRLB	VRR-c	VF	α[7,9]		Dv					22-26
E77E	VECTOR SHIFT RIGHT ARITHMETIC	VSRA	VRR-c	VF	α[7,9]		Dv					22-25
E77F	VECTOR SHIFT RIGHT ARITHMETIC BY BYTE	VSRAB	VRR-c	VF	α[7,9]		Dv					22-26
E780	VECTOR FIND ELEMENT EQUAL	VFEE	VRR-b C*	VF	α[7,9]	SP	Dv					23-3
E781	VECTOR FIND ELEMENT NOT EQUAL	VFENE	VRR-b C*	VF	α[7,9]	SP	Dv					23-4
E782	VECTOR FIND ANY ELEMENT EQUAL	VFAE	VRR-b C*	VF	α[7,9]	SP	Dv					23-1
E784	VECTOR PERMUTE DOUBLEWORD IMMEDIATE	VPDI	VRR-c	VF	α[7,9]		Dv					21-15
E785	VECTOR BIT PERMUTE	VBPERM	VRR-c	V1	α[7,9]		Dv					21-4
E78A	VECTOR STRING RANGE COMPARE	VSTRC	VRR-d C*	VF	α[7,9]	SP	Dv					23-6
E78C	VECTOR PERMUTE	VPERM	VRR-e	VF	α[7,9]		Dv					21-15
E78D	VECTOR SELECT	VSEL	VRR-e	VF	α[7,9]		Dv					21-17
E78E	VECTOR FP MULTIPLY AND SUBTRACT	VFMS	VRR-e	VF	α[7,9]	SP	Dv	Xi	Xo	Xu	Xx	24-41
E78F	VECTOR FP MULTIPLY AND ADD	VFMA	VRR-e	VF	α[7,9]	SP	Dv	Xi	Xo	Xu	Xx	24-41
E794	VECTOR PACK	VPK	VRR-c	VF	α[7,9]	SP	Dv					21-13
E795	VECTOR PACK LOGICAL SATURATE	VPKLS	VRR-b C*	VF	α[7,9]	SP	Dv					21-14
E797	VECTOR PACK SATURATE	VPKS	VRR-b C*	VF	α[7,9]	SP	Dv					21-13
E79E	VECTOR FP NEGATIVE MULTIPLY AND SUBTRACT	VFNMS	VRR-e	V1	α[7,9]	SP	Dv	Xi	Xo	Xu	Xx	24-41
E79F	VECTOR FP NEGATIVE MULTIPLY AND ADD	VFNMA	VRR-e	V1	α[7,9]	SP	Dv	Xi	Xo	Xu	Xx	24-41
E7A1	VECTOR MULTIPLY LOGICAL HIGH	VMLH	VRR-c	VF	α[7,9]	SP	Dv					22-17
E7A2	VECTOR MULTIPLY LOW	VML	VRR-c	VF	α[7,9]	SP	Dv					22-17
E7A3	VECTOR MULTIPLY HIGH	VMH	VRR-c	VF	α[7,9]	SP	Dv					22-16
E7A4	VECTOR MULTIPLY LOGICAL EVEN	VMLE	VRR-c	VF	α[7,9]	SP	Dv					22-18
E7A5	VECTOR MULTIPLY LOGICAL ODD	VMLO	VRR-c	VF	α[7,9]	SP	Dv					22-18
E7A6	VECTOR MULTIPLY EVEN	VME	VRR-c	VF	α[7,9]	SP	Dv					22-18
E7A7	VECTOR MULTIPLY ODD	VMO	VRR-c	VF	α[7,9]	SP	Dv					22-18
E7A9	VECTOR MULTIPLY AND ADD LOGICAL HIGH	VMALH	VRR-d	VF	α[7,9]	SP	Dv					22-15
E7AA	VECTOR MULTIPLY AND ADD LOW	VMAL	VRR-d	VF	α[7,9]	SP	Dv					22-14
E7AB	VECTOR MULTIPLY AND ADD HIGH	VMAH	VRR-d	VF	α[7,9]	SP	Dv					22-15
E7AC	VECTOR MULTIPLY AND ADD LOGICAL EVEN	VMALE	VRR-d	VF	α[7,9]	SP	Dv					22-15
E7AD	VECTOR MULTIPLY AND ADD LOGICAL ODD	VMALO	VRR-d	VF	α[7,9]	SP	Dv					22-16
E7AE	VECTOR MULTIPLY AND ADD EVEN	VMAE	VRR-d	VF	α[7,9]	SP	Dv					22-15
E7AF	VECTOR MULTIPLY AND ADD ODD	VMAO	VRR-d	VF	α[7,9]	SP	Dv					22-16
E7B4	VECTOR GALOIS FIELD MULTIPLY SUM	VGFM	VRR-c	VF	α[7,9]	SP	Dv					22-11
E7B8	VECTOR MULTIPLY SUM LOGICAL	VMSL	VRR-d	V1	α[7,9]	SP	Dv					22-19
E7B9	VECTOR ADD WITH CARRY COMPUTE CARRY	VACCC	VRR-d	VF	α[7,9]	SP	Dv					22-5
E7BB	VECTOR ADD WITH CARRY	VAC	VRR-d	VF	α[7,9]	SP	Dv					22-4
E7BC	VECTOR GALOIS FIELD MULTIPLY SUM AND ACCUMULATE	VGFMA	VRR-d	VF	α[7,9]	SP	Dv					22-12
E7BD	VECTOR SUBTRACT WITH BORROW COMPUTE BORROW INDICATION	VSBCBI	VRR-d	VF	α[7,9]	SP	Dv					22-28

Figure B-3. Instructions Arranged by Opcode (Part 18 of 23)

| Op-code | Name | Mne-monic | Characteristics | | |

Op-code	Name	Mne-monic	Characteristics												Page
EB1C	ROTATE LEFT SINGLE LOGICAL (64)	RLLG	RSY-a		N										7-361
EB1D	ROTATE LEFT SINGLE LOGICAL (32)	RLL	RSY-a		N3										7-361
EB20	COMPARE LOGICAL CHAR. UNDER MASK (high)	CLMH	RSY-b	C	N		A							B_2	7-157
EB21	COMPARE LOGICAL CHAR. UNDER MASK (low)	CLMY	RSY-b	C	LD		A							B_2	7-157
EB23	COMPARE LOGICAL AND TRAP (32)	CLT	RSY-b		MI1		A			Dc				B_2	7-155
EB24	STORE MULTIPLE (64)	STMG	RSY-a		N		A					ST		B_2	7-385
EB25	STORE CONTROL (64)	STCTG	RSY-a		N	P	A	SP				ST		B_2	10-135
EB26	STORE MULTIPLE HIGH (32)	STMH	RSY-a		N		A					ST		B_2	7-385
EB2B	COMPARE LOGICAL AND TRAP (64)	CLGT	RSY-b		MI1		A			Dc				B_2	7-155
EB2C	STORE CHARACTERS UNDER MASK (high)	STCMH	RSY-b		N	¤[9,11]	A					ST		B_2	7-378
EB2D	STORE CHARACTERS UNDER MASK (low)	STCMY	RSY-b		LD		A					ST		B_2	7-378
EB2F	LOAD CONTROL (64)	LCTLG	RSY-a		N	P	A	SP						B_2	10-50
EB30	COMPARE AND SWAP (64)	CSG	RSY-a	C	N	¤[9]	A	SP			$	ST		B_2	7-144
EB31	COMPARE DOUBLE AND SWAP (32)	CDSY	RSY-a	C	LD	¤[9]	A	SP			$	ST		B_2	7-144
EB3E	COMPARE DOUBLE AND SWAP (64)	CDSG	RSY-a	C	N	¤[9]	A	SP			$	ST		B_2	7-144
EB44	BRANCH ON INDEX HIGH (64)	BXHG	RSY-a		N	¤[9]						B			7-42
EB45	BRANCH ON INDEX LOW OR EQUAL (64)	BXLEG	RSY-a		N	¤[9]						B			7-42
EB4C	EXTRACT CPU ATTRIBUTE	ECAG	RSY-a		GE	¤[9]									7-257
EB51	TEST UNDER MASK	TMY	SIY	C	LD		A							B_1	7-393
EB52	MOVE (immediate)	MVIY	SIY		LD		A					ST		B_1	7-289
EB54	AND (immediate)	NIY	SIY	C	LD		A				£[2]	ST		B_1	7-34
EB55	COMPARE LOGICAL (immediate)	CLIY	SIY	C	LD		A							B_1	7-152
EB56	OR (immediate)	OIY	SIY	C	LD		A					ST		B_1	7-311
EB57	EXCLUSIVE OR (immediate)	XIY	SIY	C	LD		A					ST		B_1	7-255
EB6A	ADD IMMEDIATE (32←8)	ASI	SIY	C	GE		A			IF	£[1]	ST		B_1	7-27
EB6E	ADD LOGICAL WITH SIGNED IMMEDIATE (32←8)	ALSI	SIY	C	GE		A				£[1]	ST		B_1	7-32
EB7A	ADD IMMEDIATE (64←8)	AGSI	SIY	C	GE		A			IF	£[1]	ST		B_1	7-27
EB7E	ADD LOGICAL WITH SIGNED IMMEDIATE (64←8)	ALGSI	SIY	C	GE		A				£[1]	ST		B_1	7-32
EB80	INSERT CHARACTERS UNDER MASK (high)	ICMH	RSY-b	C	N		A							B_2	7-262
EB81	INSERT CHARACTERS UNDER MASK (low)	ICMY	RSY-b	C	LD		A							B_2	7-262
EB8E	MOVE LONG UNICODE	MVCLU	RSY-a	C	E2	¤[9]	A	SP	IC			ST	R_1	R_3	7-297
EB8F	COMPARE LOGICAL LONG UNICODE	CLCLU	RSY-a	C	E2	¤[9]	A	SP	IC				R_1	R_2	7-163
EB90	STORE MULTIPLE (32)	STMY	RSY-a		LD		A					ST		B_2	7-385
EB96	LOAD MULTIPLE HIGH (32)	LMH	RSY-a		N		A							B_2	7-283
EB98	LOAD MULTIPLE (32)	LMY	RSY-a		LD		A							B_2	7-282
EB9A	LOAD ACCESS MULTIPLE	LAMY	RSY-a		LD	¤[6]	A	SP						UB	7-265
EB9B	STORE ACCESS MULTIPLE	STAMY	RSY-a		LD		A	SP				ST		UB	7-377
EBC0	TEST DECIMAL	TP	RSL-a	C	E2	¤[9]	A						B_1	B_2	8-14
EBDC	SHIFT RIGHT SINGLE (32)	SRAK	RSY-a	C	DO										7-375
EBDD	SHIFT LEFT SINGLE (32)	SLAK	RSY-a	C	DO					IF					7-372
EBDE	SHIFT RIGHT SINGLE LOGICAL (32)	SRLK	RSY-a		DO										7-376
EBDF	SHIFT LEFT SINGLE LOGICAL (32)	SLLK	RSY-a		DO										7-373
EBE0	LOAD HIGH ON CONDITION (32)	LOCFH	RSY-b		L2		A							B_2	7-284
EBE1	STORE HIGH ON CONDITION	STOCFH	RSY-b		L2		A					ST		B_2	7-386
EBE2	LOAD ON CONDITION (64)	LOCG	RSY-b		L1		A							B_2	7-284
EBE3	STORE ON CONDITION (64)	STOCG	RSY-b		L1		A					ST		B_2	7-385
EBE4	LOAD AND AND (64)	LANG	RSY-a	C	IA	¤[9]	A	SP			£	ST		B_2	7-269
EBE6	LOAD AND OR (64)	LAOG	RSY-a	C	IA	¤[9]	A	SP			£	ST		B_2	7-270
EBE7	LOAD AND EXCLUSIVE OR (64)	LAXG	RSY-a	C	IA	¤[9]	A	SP			£	ST		B_2	7-269
EBE8	LOAD AND ADD (64)	LAAG	RSY-a	C	IA	¤[9]	A	SP		IF	£	ST		B_2	7-268
EBEA	LOAD AND ADD LOGICAL (64)	LAALG	RSY-a	C	IA	¤[9]	A	SP			£	ST		B_2	7-268
EBF2	LOAD ON CONDITION (32)	LOC	RSY-b		L1		A							B_2	7-284
EBF3	STORE ON CONDITION (32)	STOC	RSY-b		L1		A					ST		B_2	7-385
EBF4	LOAD AND AND (32)	LAN	RSY-a	C	IA	¤[9]	A	SP			£	ST		B_2	7-269
EBF6	LOAD AND OR (32)	LAO	RSY-a	C	IA	¤[9]	A	SP			£	ST		B_2	7-270

Figure B-3. Instructions Arranged by Opcode (Part 20 of 23)

Op-code	Name	Mne-monic	Characteristics										Page
EBF7	LOAD AND EXCLUSIVE OR (32)	LAX	RSY-a	C	IA	¤[9]	A	SP		£		ST	B_2	7-269
EBF8	LOAD AND ADD (32)	LAA	RSY-a	C	IA	¤[9]	A	SP	IF	£		ST	B_2	7-268
EBFA	LOAD AND ADD LOGICAL (32)	LAAL	RSY-a	C	IA	¤[9]	A	SP		£		ST	B_2	7-268
EC42	LOAD HALFWORD IMMEDIATE ON CONDITION (32←16)	LOCHI	RIE-g		L2									7-277
EC44	BRANCH RELATIVE ON INDEX HIGH (64)	BRXHG	RIE-e		N	¤[9]						B		7-48
EC45	BRANCH RELATIVE ON INDEX LOW OR EQ. (64)	BRXLG	RIE-e		N	¤[9]						B		7-48
EC46	LOAD HALFWORD IMMEDIATE ON CONDITION (64←16)	LOCGHI	RIE-g		L2									7-277
EC4E	LOAD HALFWORD HIGH IMMEDIATE ON CONDITION (32←16)	LOCHHI	RIE-g		L2									7-277
EC51	ROTATE THEN INSERT SELECTED BITS LOW (64)	RISBLG	RIE-f		HW									7-365
EC54	ROTATE THEN AND SELECTED BITS (64)	RNSBG	RIE-f	C	GE									7-361
EC55	ROTATE THEN INSERT SELECTED BITS (64)	RISBG	RIE-f	C	GE									7-363
EC56	ROTATE THEN OR SELECTED BITS (64)	ROSBG	RIE-f	C	GE									7-362
EC57	ROTATE THEN EXCLUSIVE OR SELECT. BITS (64)	RXSBG	RIE-f	C	GE									7-362
EC59	ROTATE THEN INSERT SELECTED BITS (64)	RISBGN	RIE-f		MI1									7-363
EC5D	ROTATE THEN INSERT SELECTED BITS HIGH (64)	RISBHG	RIE-f		HW									7-364
EC64	COMPARE AND BRANCH RELATIVE (64)	CGRJ	RIE-b		GE	¤[10]						B		7-136
EC65	COMPARE LOGICAL AND BRANCH RELATIVE (64)	CLGRJ	RIE-b		GE	¤[10]						B		7-154
EC70	COMPARE IMMEDIATE AND TRAP (64←16)	CGIT	RIE-a		GE					Dc				7-149
EC71	COMPARE LOGICAL IMMEDIATE AND TRAP (64←16)	CLGIT	RIE-a		GE					Dc				7-156
EC72	COMPARE IMMEDIATE AND TRAP (32←16)	CIT	RIE-a		GE					Dc				7-149
EC73	COMPARE LOGICAL IMMEDIATE AND TRAP (32←16)	CLFIT	RIE-a		GE					Dc				7-156
EC76	COMPARE AND BRANCH RELATIVE (32)	CRJ	RIE-b		GE	¤[10]						B		7-135
EC77	COMPARE LOGICAL AND BRANCH RELATIVE (32)	CLRJ	RIE-b		GE	¤[10]						B		7-154
EC7C	COMPARE IMMEDIATE AND BRANCH RELATIVE (64←8)	CGIJ	RIE-c		GE	¤[10]						B		7-136
EC7D	COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (64←8)	CLGIJ	RIE-c		GE	¤[10]						B		7-154
EC7E	COMPARE IMMEDIATE AND BRANCH RELATIVE (32←8)	CIJ	RIE-c		GE	¤[10]						B		7-136
EC7F	COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (32←8)	CLIJ	RIE-c		GE	¤[10]						B		7-154
ECD8	ADD IMMEDIATE (32←16)	AHIK	RIE-d	C	DO					IF				7-27
ECD9	ADD IMMEDIATE (64←16)	AGHIK	RIE-d	C	DO					IF				7-27
ECDA	ADD LOGICAL WITH SIGNED IMMEDIATE (32←16)	ALHSIK	RIE-d	C	DO									7-32
ECDB	ADD LOGICAL WITH SIGNED IMMEDIATE (64←16)	ALGHSIK	RIE-d	C	DO									7-32
ECE4	COMPARE AND BRANCH (64)	CGRB	RRS		GE	¤[9]						B		7-135
ECE5	COMPARE LOGICAL AND BRANCH (64)	CLGRB	RRS		GE	¤[9]						B		7-154
ECF6	COMPARE AND BRANCH (32)	CRB	RRS		GE	¤[9]						B		7-135
ECF7	COMPARE LOGICAL AND BRANCH (32)	CLRB	RRS		GE	¤[9]						B		7-154
ECFC	COMPARE IMMEDIATE AND BRANCH (64←8)	CGIB	RIS		GE	¤[9]						B		7-136
ECFD	COMPARE LOGICAL IMMEDIATE AND BRANCH (64←8)	CLGIB	RIS		GE	¤[9]						B		7-154
ECFE	COMPARE IMMEDIATE AND BRANCH (32←8)	CIB	RIS		GE	¤[9]						B		7-136
ECFF	COMPARE LOGICAL IMMEDIATE AND BRANCH (32←8)	CLIB	RIS		GE	¤[9]						B		7-154
ED04	LOAD LENGTHENED (short to long BFP)	LDEB	RXE			¤[7,9]	A		Db	Xi			B_2	19-34
ED05	LOAD LENGTHENED (long to extended BFP)	LXDB	RXE			¤[7,9]	A	SP	Db	Xi			B_2	19-34

Figure B-3. Instructions Arranged by Opcode (Part 21 of 23)

Op-code	Name	Mne-monic	Characteristics																Page
ED06	LOAD LENGTHENED (short to extended BFP)	LXEB	RXE				$\alpha^{7,9}$	A	SP	Db	Xi							B_2	19-34
ED07	MULTIPLY (long to extended BFP)	MXDB	RXE				$\alpha^{7,9}$	A	SP	Db	Xi							B_2	19-37
ED08	COMPARE AND SIGNAL (short BFP)	KEB	RXE	C			$\alpha^{7,9}$	A		Db	Xi							B_2	19-18
ED09	COMPARE (short BFP)	CEB	RXE	C			$\alpha^{7,9}$	A		Db	Xi							B_2	19-17
ED0A	ADD (short BFP)	AEB	RXE	C			$\alpha^{7,9}$	A		Db	Xi			Xo	Xu	Xx		B_2	19-15
ED0B	SUBTRACT (short BFP)	SEB	RXE	C			$\alpha^{7,9}$	A		Db	Xi			Xo	Xu	Xx		B_2	19-40
ED0C	MULTIPLY (short to long BFP)	MDEB	RXE				$\alpha^{7,9}$	A		Db	Xi							B_2	19-37
ED0D	DIVIDE (short BFP)	DEB	RXE				$\alpha^{7,9}$	A		Db	Xi	Xz		Xo	Xu	Xx		B_2	19-27
ED0E	MULTIPLY AND ADD (short BFP)	MAEB	RXF				$\alpha^{7,9}$	A		Db	Xi			Xo	Xu	Xx		B_2	19-38
ED0F	MULTIPLY AND SUBTRACT (short BFP)	MSEB	RXF				$\alpha^{7,9}$	A		Db	Xi			Xo	Xu	Xx		B_2	19-38
ED10	TEST DATA CLASS (short BFP)	TCEB	RXE	C			$\alpha^{7,9}$			Db									19-41
ED11	TEST DATA CLASS (long BFP)	TCDB	RXE	C			$\alpha^{7,9}$			Db									19-41
ED12	TEST DATA CLASS (extended BFP)	TCXB	RXE	C			$\alpha^{7,9}$		SP	Db									19-41
ED14	SQUARE ROOT (short BFP)	SQEB	RXE				$\alpha^{7,9}$	A		Db	Xi					Xx		B_2	19-40
ED15	SQUARE ROOT (long BFP)	SQDB	RXE				$\alpha^{7,9}$	A		Db	Xi					Xx		B_2	19-40
ED17	MULTIPLY (short BFP)	MEEB	RXE				$\alpha^{7,9}$	A		Db	Xi			Xo	Xu	Xx		B_2	19-37
ED18	COMPARE AND SIGNAL (long BFP)	KDB	RXE	C			$\alpha^{7,9}$	A		Db	Xi							B_2	19-18
ED19	COMPARE (long BFP)	CDB	RXE	C			$\alpha^{7,9}$	A		Db	Xi							B_2	19-17
ED1A	ADD (long BFP)	ADB	RXE	C			$\alpha^{7,9}$	A		Db	Xi			Xo	Xu	Xx		B_2	19-15
ED1B	SUBTRACT (long BFP)	SDB	RXE	C			$\alpha^{7,9}$	A		Db	Xi			Xo	Xu	Xx		B_2	19-40
ED1C	MULTIPLY (long BFP)	MDB	RXE				$\alpha^{7,9}$	A		Db	Xi			Xo	Xu	Xx		B_2	19-37
ED1D	DIVIDE (long BFP)	DDB	RXE				$\alpha^{7,9}$	A		Db	Xi	Xz		Xo	Xu	Xx		B_2	19-27
ED1E	MULTIPLY AND ADD (long BFP)	MADB	RXF				$\alpha^{7,9}$	A		Db	Xi			Xo	Xu	Xx		B_2	19-38
ED1F	MULTIPLY AND SUBTRACT (long BFP)	MSDB	RXF				$\alpha^{7,9}$	A		Db	Xi			Xo	Xu	Xx		B_2	19-38
ED24	LOAD LENGTHENED (short to long HFP)	LDE	RXE				$\alpha^{7,9}$	A		Da								B_2	18-15
ED25	LOAD LENGTHENED (long to extended HFP)	LXD	RXE				$\alpha^{7,9}$	A	SP	Da								B_2	18-15
ED26	LOAD LENGTHENED (short to extended HFP)	LXE	RXE				$\alpha^{7,9}$	A	SP	Da								B_2	18-15
ED2E	MULTIPLY AND ADD (short HFP)	MAE	RXF		HM		$\alpha^{7,9}$	A		Da	EU	EO						B_2	18-19
ED2F	MULTIPLY AND SUBTRACT (short HFP)	MSE	RXF		HM		$\alpha^{7,9}$	A		Da	EU	EO						B_2	18-19
ED34	SQUARE ROOT (short HFP)	SQE	RXE				$\alpha^{7,9}$	A		Da				SQ				B_2	18-23
ED35	SQUARE ROOT (long HFP)	SQD	RXE				$\alpha^{7,9}$	A		Da				SQ				B_2	18-23
ED37	MULTIPLY (short HFP)	MEE	RXE				$\alpha^{7,9}$	A		Da	EU	EO						B_2	18-18
ED38	MULTIPLY AND ADD UNNRM. (long to ext. low HFP)	MAYL	RXF		UE		$\alpha^{7,9}$	A		Da								B_2	18-20
ED39	MULTIPLY UNNORM. (long to ext. low HFP)	MYL	RXF		UE		$\alpha^{7,9}$	A		Da								B_2	18-22
ED3A	MULTIPLY & ADD UNNORMALIZED (long to ext. HFP)	MAY	RXF		UE		$\alpha^{7,9}$	A		Da								B_2	18-20
ED3B	MULTIPLY UNNORMALIZED (long to ext. HFP)	MY	RXF		UE		$\alpha^{7,9}$	A	SP	Da								B_2	18-22
ED3C	MULTIPLY AND ADD UNNRM. (long to ext. high HFP)	MAYH	RXF		UE		$\alpha^{7,9}$	A		Da								B_2	18-20
ED3D	MULTIPLY UNNORM. (long to ext. high HFP)	MYH	RXF		UE		$\alpha^{7,9}$	A		Da								B_2	18-22
ED3E	MULTIPLY AND ADD (long HFP)	MAD	RXF		HM		$\alpha^{7,9}$	A		Da	EU	EO						B_2	18-19
ED3F	MULTIPLY AND SUBTRACT (long HFP)	MSD	RXF		HM		$\alpha^{7,9}$	A		Da	EU	EO						B_2	18-19
ED40	SHIFT SIGNIFICAND LEFT (long DFP)	SLDT	RXF		TF		$\alpha^{7,9}$			Dt									20-54
ED41	SHIFT SIGNIFICAND RIGHT (long DFP)	SRDT	RXF		TF		$\alpha^{7,9}$			Dt									20-54
ED48	SHIFT SIGNIFICAND LEFT (extended DFP)	SLXT	RXF		TF		$\alpha^{7,9}$		SP	Dt									20-54
ED49	SHIFT SIGNIFICAND RIGHT (extended DFP)	SRXT	RXF		TF		$\alpha^{7,9}$		SP	Dt									20-54
ED50	TEST DATA CLASS (short DFP)	TDCET	RXE	C	TF		$\alpha^{7,9}$			Dt									20-56
ED51	TEST DATA GROUP (short DFP)	TDGET	RXE	C	TF		$\alpha^{7,9}$			Dt									20-57
ED54	TEST DATA CLASS (long DFP)	TDCDT	RXE	C	TF		$\alpha^{7,9}$			Dt									20-56
ED55	TEST DATA GROUP (long DFP)	TDGDT	RXE	C	TF		$\alpha^{7,9}$			Dt									20-57
ED58	TEST DATA CLASS (extended DFP)	TDCXT	RXE	C	TF		$\alpha^{7,9}$		SP	Dt									20-56
ED59	TEST DATA GROUP (extended DFP)	TDGXT	RXE	C	TF		$\alpha^{7,9}$		SP	Dt									20-57
ED64	LOAD (short)	LEY	RXY-a		LD		$\alpha^{7,9}$	A		Da								B_2	9-31
ED65	LOAD (long)	LDY	RXY-a		LD		$\alpha^{7,9}$	A		Da								B_2	9-31

Figure B-3. Instructions Arranged by Opcode (Part 22 of 23)

Op-code	Name	Mne-monic	Characteristics														Page
ED66	STORE (short)	STEY	RXY-a		LD	¤[7,9]	A		Da				ST		B_2		9-49
ED67	STORE (long)	STDY	RXY-a		LD	¤[7,9]	A		Da				ST		B_2		9-49
EDA8	CONVERT TO ZONED (from long DFP)	CZDT	RSL-b	C	ZF	¤[7,9]	A	SP					ST		B_2		20-36
EDA9	CONVERT TO ZONED (from extended DFP)	CZXT	RSL-b	C	ZF	¤[7,9]	A	SP					ST		B_2		20-36
EDAA	CONVERT FROM ZONED (to long DFP)	CDZT	RSL-b		ZF	¤[7,9]	A	SP	Dt	Dg					B_2		20-29
EDAB	CONVERT FROM ZONED (to extended DFP)	CXZT	RSL-b		ZF	¤[7,9]	A	SP	Dt	Dg					B_2		20-29
EDAC	CONVERT TO PACKED (from long DFP)	CPDT	RSL-b	C	PC	¤[7,9]	A	SP	Dt	DF			ST		B_2		20-33
EDAD	CONVERT TO PACKED (from extended DFP)	CPXT	RSL-b	C	PC	¤[7,9]	A	SP	Dt	DF			ST		B_2		20-33
EDAE	CONVERT FROM PACKED (to long DFP)	CDPT	RSL-b		PC	¤[7,9]	A	SP	Dt	Dg					B_2		20-26
EDAF	CONVERT FROM PACKED (to extended DFP)	CXPT	RSL-b		PC	¤[7,9]	A	SP	Dt	Dg					B_2		20-26
EE	PERFORM LOCKED OPERATION	PLO	SS-e	C		¤[1]	A	SP			$	GM	ST		FC		7-331
EF	LOAD MULTIPLE DISJOINT (64←32&32)	LMD	SS-e		N	¤[9]	A								B_2	B_4	7-283
F0	SHIFT AND ROUND DECIMAL	SRP	SS-c	C		¤[9]	A		Dg	DF			ST		B_1	B_2	8-12
F1	MOVE WITH OFFSET	MVO	SS-b			¤[9]	A						ST		B_1	B_2	7-302
F2	PACK	PACK	SS-b			¤[9]	A						ST		B_1	B_2	7-312
F3	UNPACK	UNPK	SS-b			¤[9]	A						ST		B_1	B_2	7-416
F8	ZERO AND ADD	ZAP	SS-b	C		¤[9]	A		Dg	DF			ST		B_1	B_2	8-14
F9	COMPARE DECIMAL	CP	SS-b	C		¤[9]	A		Dg						B_1	B_2	8-7
FA	ADD DECIMAL	AP	SS-b	C		¤[9]	A		Dg	DF			ST		B_1	B_2	8-6
FB	SUBTRACT DECIMAL	SP	SS-b	C		¤[9]	A		Dg	DF			ST		B_1	B_2	8-13
FC	MULTIPLY DECIMAL	MP	SS-b			¤[9]	A	SP	Dg				ST		B_1	B_2	8-12
FD	DIVIDE DECIMAL	DP	SS-b			¤[9]	A	SP	Dg		DK		ST		B_1	B_2	8-7

Figure B-3. Instructions Arranged by Opcode (Part 23 of 23)

Appendix C. Condition-Code Settings

This appendix lists the condition-code setting for instructions in z/Architecture which set the condition code. In addition to those instructions listed which set the condition code, the condition code may be changed by DIAGNOSE and the target of EXECUTE. The condition code is loaded by LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, RESUME PROGRAM, and SET PROGRAM MASK and by an interruption. The condition code is set to zero by initial CPU reset and is loaded by the successful conclusion of the initial-program-loading sequence.

Some models may offer instructions which set the condition code and do not appear in this document, such as those provided for assists or as part of special or custom features.

Instruction	Condition Code			
	0	1	2	3
ADD (BFP, DFP)	Zero	< zero	> zero	NaN
ADD (general)	Zero	< zero	> zero	Overflow
ADD DECIMAL	Zero	< zero	> zero	Overflow
ADD HALFWORD	Zero	< zero	> zero	Overflow
ADD HALFWORD IMMEDIATE	Zero	< zero	> zero	Overflow
ADD HIGH	Zero	< zero	> zero	Overflow
ADD IMMEDIATE	Zero	< zero	> zero	Overflow
ADD IMMEDIATE HIGH	Zero	< zero	> zero	Overflow
ADD LOGICAL	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD LOGICAL HIGH	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD LOGICAL IMMEDIATE	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD LOGICAL WITH CARRY	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD LOGICAL WITH SIGNED IMMEDIATE	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD LOGICAL WITH SIGNED IMMEDIATE HIGH	Zero, no carry	Not zero, no carry	Zero, carry	Not zero, carry
ADD NORMALIZED	Zero	< zero	> zero	—
ADD UNNORMALIZED	Zero	< zero	> zero	—
AND	Zero	Not zero	—	—
AND IMMEDIATE	Zero	Not zero	—	—
CANCEL SUBCHANNEL	Function initiated	—	—	Not operational
CHECKSUM	Checksum complete	—	—	CPU-determined completion
CIPHER MESSAGE	Normal completion	Verification mismatch	—	Partial completion
CIPHER MESSAGE WITH AUTHENTICATION	Normal completion	Verification mismatch	Partial completion (LAAD or LPC zero)	Partial completion (time out)
CIPHER MESSAGE WITH CHAINING	Normal completion	Verification mismatch	—	Partial completion
CIPHER MESSAGE WITH CIPHER FEEDBACK	Normal completion	Verification mismatch	—	Partial completion
CIPHER MESSAGE WITH COUNTER	Normal completion	Verification mismatch	—	Partial completion
CIPHER MESSAGE WITH OUTPUT FEEDBACK	Normal completion	Verification mismatch	—	Partial completion
CLEAR SUBCHANNEL	Function initiated	—	—	Not operational
COMPARE (BFP, DFP)	Equal	Low	High	Unordered
COMPARE (general, HFP)	Equal	Low	High	—
COMPARE AND FORM CODEWORD	Equal	OCB=0: low OCB=1: high	OCB=0: high OCB=1: low	—
COMPARE AND REPLACE DAT TABLE ENTRY	Equal	Not equal	—	—
COMPARE AND SIGNAL	Equal	Low	High	Unordered

Figure C-1. Summary of Condition-Code Settings (Part 1 of 7)

© Copyright IBM Corp. 2000, 2017

Instruction	Condition Code			
	0	1	2	3
COMPARE AND SWAP	Equal	Not equal	—	—
COMPARE AND SWAP AND PURGE	Equal	Not equal	—	—
COMPARE AND SWAP AND STORE	Equal	Not equal	—	—
COMPARE BIASED EXPONENT	Equal	Low	High	Unordered
COMPARE DECIMAL	Equal	Low	High	—
COMPARE DOUBLE AND SWAP	Equal	Not equal	—	—
COMPARE HALFWORD	Equal	Low	High	—
COMPARE HALFWORD IMMEDIATE	Equal	Low	High	—
COMPARE HALFWORD RELATIVE LONG	Equal	Low	High	—
COMPARE HIGH	Equal	Low	High	—
COMPARE IMMEDIATE	Equal	Low	High	—
COMPARE IMMEDIATE HIGH	Equal	Low	High	—
COMPARE LOGICAL	Equal	Low	High	—
COMPARE LOGICAL CHARACTERS UNDER MASK	Equal	Low	High	—
COMPARE LOGICAL HIGH	Equal	Low	High	—
COMPARE LOGICAL IMMEDIATE	Equal	Low	High	—
COMPARE LOGICAL IMMEDIATE HIGH	Equal	Low	High	—
COMPARE LOGICAL LONG	Equal	Low	High	—
COMPARE LOGICAL LONG EXTENDED	Equal	Low	High	CPU-determined completion
COMPARE LOGICAL LONG UNICODE	Equal	Low	High	CPU-determined completion
COMPARE LOGICAL RELATIVE LONG	Equal	Low	High	—
COMPARE LOGICAL STRING	Equal	Low	High	CPU-determined completion
COMPARE RELATIVE LONG	Equal	Low	High	—
COMPARE UNTIL SUBSTRING EQUAL	Equal substrings	Last bytes equal	Last bytes unequal	CPU-determined completion
COMPRESSION CALL	Op2 processed	Op1 full and op2 not processed	—	CPU-determined completion
COMPUTE INTERMEDIATE MESSAGE DIGEST	Normal completion	—	—	Partial completion
COMPUTE LAST MESSAGE DIGEST	Normal completion	—	—	Partial completion
COMPUTE MESSAGE AUTHENTICATION CODE	Normal completion	Verification mismatch	—	Partial completion
CONVERT BFP TO HFP	Zero	< zero	> zero	Special case
CONVERT HFP TO BFP	Zero	< zero	> zero	Special case
CONVERT TO FIXED	Zero	< zero	> zero	Special case
CONVERT TO LOGICAL	Zero	< zero	> zero	Special case
CONVERT TO PACKED	Source is zero	Source is less than zero	Source is greater than zero	Infinity, QNaN, SNaN, Partial result
CONVERT TO ZONED	Source is zero	Source is less than zero	Source is greater than zero	Infinity, QNaN, SNaN, Partial result
CONVERT UTF-16 TO UTF-32	Data processed	Op1 full	Invalid low surrogate	CPU-determined completion
CONVERT UTF-16 TO UTF-8	Data processed	Op1 full	Invalid low surrogate	CPU-determined completion
CONVERT UTF-32 TO UTF-16	Data processed	Op1 full	Invalid UTF-32 character	CPU-determined completion

Figure C-1. Summary of Condition-Code Settings (Part 2 of 7)

Instruction	Condition Code			
	0	1	2	3
CONVERT UTF-32 TO UTF-8	Data processed	Op1 full	Invalid UTF-32 character	CPU-determined completion
CONVERT UTF-8 TO UTF-16	Data processed	Op1 full	Invalid UTF-8 character	CPU-determined completion
CONVERT UTF-8 TO UTF-32	Data processed	Op1 full	Invalid UTF-8 character	CPU-determined completion
DIAGNOSE[1]	See note	See note	See note	See note
DIVIDE TO INTEGER	Remainder complete; normal quotient	Remainder complete; quotient overflow or NaN	Remainder incomplete; normal quotient	Remainder incomplete; quotient overflow or NaN
EDIT	Zero	< zero	> zero	—
EDIT AND MARK	Zero	< zero	> zero	—
EXCLUSIVE OR	Zero	Not zero	—	—
EXCLUSIVE OR IMMEDIATE	Zero	Not zero	—	—
EXTRACT STACKED STATE	Branch state entry	Program-call state entry	—	—
FIND LEFTMOST ONE	No one bit found	—	One bit found	—
HALT SUBCHANNEL	Function initiated	Status-pending with other than intermediate status	Busy	Not operational
INSERT ADDRESS SPACE CONTROL	Primary-space mode	Secondary-space mode	Access-register mode	Home-space mode
INSERT CHARACTERS UNDER MASK	All zeros	First bit one	First bit zero	—
LOAD ADDRESS SPACE PARAMETERS	Parameters loaded	Primary ASN not available	Secondary ASN not available or not authorized	Space-switch event
LOAD AND TEST (BFP, DFP)	Zero	< zero	> zero	NaN
LOAD AND TEST (general, HFP)	Zero	< zero	> zero	—
LOAD COMPLEMENT (BFP)	Zero	< zero	> zero	NaN
LOAD COMPLEMENT (gen)	Zero	< zero	> zero	Overflow
LOAD COMPLEMENT (HFP)	Zero	< zero	> zero	—
LOAD COUNT TO BLOCK BOUNDARY	Operand 1 = 16	—	—	Operand 1 < 16
LOAD NEGATIVE (BFP)	Zero	< zero	—	NaN
LOAD NEGATIVE (gen, HFP)	Zero	< zero	—	—
LOAD PAGE TABLE ENTRY ADDRESS	Translation, STE.P = 0	Translation, STE.P = 1	I bit on in RTE or STE, or enhanced-DAT applies and STE.FC=1	Exception condition exists
LOAD POSITIVE (BFP)	Zero	—	> zero	NaN
LOAD POSITIVE (gen)	Zero	—	> zero	Overflow
LOAD POSITIVE (HFP)	Zero	—	> zero	—
LOAD PSW[3] / LOAD PSW EXTENDED[3]	See note	See note	See note	See note
LOAD REAL ADDRESS [2]	Translation available	ST entry invalid	PT entry invalid	ASCE or entry not available or length violation
MODIFY SUBCHANNEL	SCHIB information placed in subchannel	Status-pending	Busy	Not operational
MOVE LONG	Length equal	Length low	Length high	Destructive overlap

Figure C-1. Summary of Condition-Code Settings (Part 3 of 7)

Instruction	Condition Code			
	0	1	2	3
MOVE LONG EXTENDED	Length equal	Length low	Length high	CPU-determined completion
MOVE LONG UNICODE	Length equal	Length low	Length high	CPU-determined completion
MOVE PAGE	Data moved	Operand 1 invalid, both valid in ES, locked, or ES error	Operand 2 invalid	—
MOVE STRING	—	Data moved	—	CPU-determined completion
MOVE TO PRIMARY	Length $\leq$ 256	—	—	Length > 256
MOVE TO SECONDARY	Length $\leq$ 256	—	—	Length > 256
MOVE WITH KEY	Length $\leq$ 256	—	—	Length > 256
MOVE WITH OPTIONAL SPECIFICATIONS	Length $\leq$ 4,096	—	—	Length > 4,096
OR	Zero	Not zero	—	—
OR IMMEDIATE	Zero	Not zero	—	—
PAGE IN	Page-in operation completed	Expanded-storage data error	—	Expanded-storage block not available
PAGE OUT	Page-out operation completed	Expanded-storage data error	—	Expanded-storage block not available
PERFORM CRYPTOGRAPHIC COMPUTATION	Normal completion	Verification mismatch	Invalid index or length	Partial completion
PERFORM FLOATING-POINT OPERATION (test bit one)	Function code valid	—	—	Function code invalid
PERFORM FLOATING-POINT OPERATION (test bit zero)	Normal result	Nontrap exception	Trap exception with alternate action	—
PERFORM LOCKED OPERATION (test bit one)	Function code valid	—	—	Function code invalid
PERFORM LOCKED OPERATION (test bit zero)	Equal	Op1 not equal	Op1 equal, op3 not equal (dcs only)	—
PERFORM RANDOM NUMBER OPERATION	Normal completion	—	—	Partial completion
PERFORM TIMING FACILITY FUNCTION	Function performed	—	—	Function not available
PERFORM TOPOLOGY FUNCTION (FC 0 or 1)	Change initiated	—	Request rejected	—
PERFORM TOPOLOGY FUNCTION (FC 2)	Report not pending	Report pending	—	—
POPULATION COUNT	Zero	Not zero	—	—
PROGRAM RETURN	See note	See note	See note	See note
RESET CHANNEL PATH	Function initiated	—	Busy	Not operational
RESET REFERENCE BIT EXTENDED	R bit zero, C bit zero	R bit zero, C bit one	R bit one, C bit zero	R bit one, C bit one
RESUME PROGRAM[3]	See note	See note	See note	See note
RESUME SUBCHANNEL	Function initiated	Status pending	Function not applicable	Not operational
ROTATE THEN AND SELECTED BITS	Zero	Not zero	—	—
ROTATE THEN EXCLUSIVE OR SELECTED BITS	Zero	Not zero	—	—
ROTATE THEN INSERT SELECTED BITS	Zero	< zero	> zero	—
ROTATE THEN OR SELECTED BITS	Zero	Not zero	—	—
SEARCH STRING	—	Found	Not found	CPU-determined completion
SEARCH STRING UNICODE	—	Found	Not found	CPU-determined completion
SET CLOCK	Set	Secure	—	Not operational

Figure C-1. Summary of Condition-Code Settings (Part 4 of 7)

Instruction	Condition Code			
	0	1	2	3
SET PROGRAM MASK[4]	See note	See note	See note	See note
SET STORAGE KEY EXTENDED	Storage key not set	Entire storage key set	Partial storage key set	Entire storage key set; bits 48-55 of GR R_1 unpredictable
SHIFT AND ROUND DECIMAL	Zero	< zero	> zero	Overflow
SHIFT LEFT (DOUBLE/SINGLE)	Zero	< zero	> zero	Overflow
SHIFT RIGHT (DOUBLE/SINGLE)	Zero	< zero	> zero	—
SIGNAL PROCESSOR	Order accepted	Status stored	Busy	Not operational
START SUBCHANNEL	Function initiated	Status-pending	Busy	Not operational
STORE CHANNEL REPORT WORD	CRW stored	Zeros stored	—	—
STORE CLOCK	Set	Not set	Error	Stopped or not operational
STORE CLOCK EXTENDED	Set	Not set	Error	Stopped or not operational
STORE CLOCK FAST	Set	Not set	Error	Stopped or not operational
STORE FACILITY LIST EXTENDED	Complete facility list stored	—	—	Incomplete facility list stored
STORE SUBCHANNEL	SCHIB stored	—	—	Not operational
STORE SYSTEM INFORMATION	Information provided	—	—	Information not available
SUBTRACT (BFP, DFP)	Zero	< zero	> zero	NaN
SUBTRACT (general)	Zero	< zero	> zero	Overflow
SUBTRACT DECIMAL	Zero	< zero	> zero	Overflow
SUBTRACT HALFWORD	Zero	< zero	> zero	Overflow
SUBTRACT HIGH	Zero	< zero	> zero	Overflow
SUBTRACT LOGICAL	—	Not zero, borrow	Zero, no borrow	Not zero, no borrow
SUBTRACT LOGICAL HIGH	—	Not zero, borrow	Zero, no borrow	Not zero, no borrow
SUBTRACT LOGICAL IMMEDIATE	—	Not zero, borrow	Zero, no borrow	Not zero, no borrow
SUBTRACT LOGICAL WITH BORROW	Zero, borrow	Not zero, borrow	Zero, no borrow	Not zero, no borrow
SUBTRACT NORMALIZED (HFP)	Zero	< zero	> zero	—
SUBTRACT UNNORMALIZED (HFP)	Zero	< zero	> zero	—
TEST ACCESS	ALET 0	DU access list, no exceptions	PS access list, no exceptions	ALET 1 or exceptions
TEST ADDRESSING MODE	Twenty-four bit mode	Thirty-one bit mode	—	Sixty-four bit mode
TEST AND SET	Left bit zero	Left bit one	—	—
TEST BLOCK	Usable	Not usable	—	—
TEST DATA CLASS	Zero (no match)	One (match)	—	—
TEST DATA GROUP	Zero (no match)	One (match)	—	—
TEST DECIMAL	Digits and sign valid	Sign invalid	Digit invalid	Sign and digit invalid
TEST PENDING EXTERNAL INTERRUPTION	None pending	One or more pending	—	—
TEST PENDING INTERRUPTION	Interruption code not stored	Interruption code stored	—	—
TEST PROTECTION	Can fetch, can store	Can fetch, cannot store	Cannot fetch, cannot store	Translation not available
TEST SUBCHANNEL	IRB stored; subchannel status-pending	IRB stored; subchannel not status-pending	—	Not operational

Figure C-1. Summary of Condition-Code Settings (Part 5 of 7)

Instruction	Condition Code			
	0	1	2	3
TEST UNDER MASK	All zeros	Mixed	—	All ones
TEST UNDER MASK (HIGH/LOW)	All zeros	Mixed, left bit zero	Mixed, left bit one	All ones
TRANSACTION BEGIN	Initiated successfully	—	—	—
TRANSACTION END	CPU in transactional-execution mode at start of instruction	—	CPU not in trans.-execution mode at start of instruction	—
TRANSLATE AND TEST	All zeros	Incomplete	Complete	—
TRANSLATE AND TEST EXTENDED	All zeros	Nonzero code selected	—	CPU-determined completion
TRANSLATE AND TEST REVERSE	All zeros	Incomplete	Complete	—
TRANSLATE AND TEST REVERSE EXTENDED	All zeros	Nonzero code selected	—	CPU-determined completion
TRANSLATE EXTENDED	Data processed	Op1 byte equal test byte	—	CPU-determined completion
TRANSLATE ONE TO ONE, ONE TO TWO, TWO TO ONE, TWO TO TWO	Character equal test character not found	Character equal test character found	—	CPU-determined completion
UNPACK ASCII	Sign plus	Sign minus	—	Sign invalid
UNPACK UNICODE	Sign plus	Sign minus	—	Sign invalid
UPDATE TREE	Equal	Not equal or no comparison	—	GR5 nonzero, GR0 negative
VECTOR ADD DECIMAL[5]	Zero	< zero	> zero	Overflow
VECTOR COMPARE DECIMAL	Equal	Low	High	—
VECTOR COMPARE EQUAL[5]	All elements equal	Some elements equal	—	No element equal
VECTOR COMPARE HIGH LOGICAL[5]	All elements high	Some elements high	—	No element high
VECTOR COMPARE HIGH[5]	All elements high	Some elements high	—	No element high
VECTOR CONVERT TO BINARY[5]	No overflow	—	—	Overflow
VECTOR CONVERT TO DECIMAL[5]	No overflow	—	—	Overflow
VECTOR DIVIDE DECIMAL[5]	Zero	< zero	> zero	Overflow
VECTOR ELEMENT COMPARE	Equal	Low	High	—
VECTOR ELEMENT COMPARE LOGICAL	Equal	Low	High	—
VECTOR FIND ANY ELEMENT EQUAL[5]	None equal, zero found	Equal element found, no zeros if ZS=1	Equal element found and zero found	No equal elements, no zeros
VECTOR FIND ELEMENT EQUAL[5]	Zero found	Equal element found, no zeros	Equal element found, and zero	Not equal, no zeros
VECTOR FIND ELEMENT NOT EQUAL[5]	Zero found	Not equal element found, less than	Not equal found, greater than	Equal, no zero
VECTOR FP COMPARE AND SIGNAL SCALAR	Elements equal	First element low	First element high	Elements unordered
VECTOR FP COMPARE EQUAL[5]	All elements equal	Mix of equal and unequal (or unordered) elements	—	All elements not equal (or unordered)
VECTOR FP COMPARE HIGH OR EQUAL[5]	All elements ≥	Mix of ≥ and <	—	All elements < (or unordered)
VECTOR FP COMPARE HIGH[5]	All elements >	Mix of > and ≤	—	All elements ≤ (or unordered)
VECTOR FP COMPARE SCALAR	Elements equal	First element low	First element high	Elements unordered
VECTOR FP TEST DATA CLASS IMMEDIATE	Match	Selected bit 1 for some (but not all) elements	—	No match
VECTOR ISOLATE STRING[5]	Zero element found	—	—	All elements nonzero
VECTOR MULTIPLY AND SHIFT DECIMAL[5]	Zero	< zero	> zero	Overflow

Figure C-1. Summary of Condition-Code Settings (Part 6 of 7)

Instruction	Condition Code			
	0	1	2	3
VECTOR MULTIPLY DECIMAL[5]	Zero	< zero	> zero	Overflow
VECTOR PACK LOGICAL SATURATE[5]	No saturation	Some saturated		All saturated
VECTOR PACK SATURATE[5]	No saturation	Some saturated		All saturated
VECTOR PERFORM SIGN OPERATION DECIMAL[5]	Zero	< zero	> zero	Overflow
VECTOR REMAINDER DECIMAL[5]	Zero	< zero	> zero	Overflow
VECTOR SHIFT AND DIVIDE DECIMAL[5]	Zero	< zero	> zero	Overflow
VECTOR SHIFT AND ROUND DECIMAL[5]	Zero	< zero	> zero	Overflow
VECTOR STRING RANGE COMPARE[5]	Zero found	At least one in ranges, no zero	At least one in ranges, zero found	No ranges match, no zeros
VECTOR SUBTRACT DECIMAL[5]	Zero	< zero	> zero	Overflow
VECTOR TEST DECIMAL	Digits and sign valid	Sign invalid	Digit invalid	Sign and digit invalid
VECTOR TEST UNDER MASK	All zeros or mask zero	Mixed	—	All ones
ZERO AND ADD	Zero	< zero	> zero	Overflow

Explanation and Notes:

— Code is not set by the instruction.

1 For DIAGNOSE, the resulting condition code is model-dependent.

2 For LOAD REAL ADDRESS, condition code 3 is set if address-space-control element not available, region-table entry outside table or invalid, segment-table entry outside table, or, for LRA in 24- or 31-bit mode when bits 0-32 of entry address not all zeros, segment- or page-table entry invalid.

3 For LOAD PSW, LOAD PSW EXTENDED, and RESUME PROGRAM, the condition code is loaded from the condition-code field of the second operand.

4 For SET PROGRAM MASK, the condition code is loaded from bit positions 2 and 3 of the first operand.

5 For various vector-facility instructions, the condition code is optionally set based on the CS control (in a mask field of the instruction).

Figure C-1. Summary of Condition-Code Settings (Part 7 of 7)

Appendix D. Compression Call Facility

Introduction to Compression Call Facility D-1
Compression and Expansion Dictionaries D-1
Compression-Dictionary Entries D-1
Compression Process D-2
Child and Sibling Characters D-2
Child and Extension-Character Combinations . D-3
Restriction on Identical Child and Sibling Characters D-3
Expansion-Dictionary Entries D-6
Expansion Process D-6
Compressed-Data Symbol Size D-7
Symbol Translation D-8
Order Preservation D-8
Entropy Encoding D-9
Results of Dictionary Errors D-10
Dictionary Formats D-10
Notation D-10
Compression Dictionary D-11
Format-0 Sibling Descriptor D-11
Format-1 Sibling Descriptor D-12
Expansion Dictionary D-12
Character Entry D-12
Format-1 Sibling Descriptor D-13

Introduction to Compression Call Facility

The compression call facility consists of the COMPRESSION CALL (CMPSC) instruction, which compresses or expands data as specified by a bit in general register 0. COMPRESSION CALL is primarily intended to operate on randomly accessed DB2* data, typically 80-byte records, so that the data can be kept on DASD in compressed form, thus saving DASD space. However, COMPRESSION CALL can also be used to compress any randomly or sequentially accessed data, provided that there is some degree of repetition of character strings (which may be actual characters or bytes of numeric or graphic data) in the data. COMPRESSION CALL also has an option, named the symbol-translation option and the order-preservation option, that allows the instruction to be used to compress VTAM* network data and DB2 keys, respectively.

In the following material, COMPRESSION CALL is first described for the case in which the symbol translation and order preservation options are not used, and then the differences due to the options are described.

Compression and Expansion Dictionaries

COMPRESSION CALL uses two static dictionaries that must be prepared by the program, before the compression operation, by scanning data that is typical of that to be compressed. One of these dictionaries is named the compression dictionary and is used only during compression. The other dictionary is named the expansion dictionary, is used during expansion, and may, depending on an option, be used slightly during compression. During expansion, the beginning of the expansion dictionary is designated by an address in general register 1. During compression, this same address designates the beginning of the compression dictionary, and the expansion dictionary, if used, immediately follows the compression dictionary.

Each of the compression and expansion dictionaries contains 512, 1K, 2K, 4K, or 8K doubleword entries, as determined by bits in general register 0. An entry in either of the dictionaries is identified or designated by means of the *index* to its position in that dictionary. That is, the first entry in a dictionary is identified or designated by index 0, the next by index 1, and so forth. The two entries at the same index position in each of the two dictionaries correspond to each other in that they both represent the same character string, which string is called a *symbol*.

Compression-Dictionary Entries

The entries in the compression dictionary correspond approximately to a Ziv-Lempel tree. There are 256 *alphabet* entries corresponding to the 256 possible values of a single-byte character, and these are the topmost entries in the tree (which is upside down). The alphabet entries may be *parent* entries that point to *child* entries. The child entries may in turn be par-

© Copyright IBM Corp. 2000, 2017

ent entries that point to their child entries. Each of the alphabet entries and the child entries is called a *character entry*. A character entry represents one or more *extension characters*. An alphabet entry represents one extension character. (The character represented by an alphabet entry is the *first* character of a character symbol, which term is defined in the next paragraph. The character is an extension to a null character represented by a physically non-existing null root entry that conceptually is the parent of the 256 alphabet entries.) A lower-level character entry represents from one to five extension characters. (In a pure Ziv-Lempel tree, each node represents only one extension character.)

In addition to representing one or more extension characters, a compression-dictionary entry represents a *character symbol* that is the extension characters of the entry preceded by the extension characters of all of the entry's ancestor entries, up to an alphabet entry.

The alphabet entries are the first entries in a dictionary and have the indexes 0-255. If an entry is a parent, it contains the index, called a *child pointer*, of its first child. Its other children, if any, follow the first child contiguously in the dictionary (and thus in storage).

Compression Process

Compression occurs as follows. The first character of the string to be compressed is used as an index to locate the alphabet entry that represents the first character. If the alphabet entry has children, the children are processed in the left-to-right order of the children, and the extension characters represented by each child are compared to the next characters of the string until a match is found or all children have been examined. If a match is found, the process is repeated using the children of that child and the next characters of the string. When the last match has been found, which might be just the match on the alphabet entry, the index of the last matching entry is output as the compressed data. This index is called an *index symbol*. The length in bits of an index symbol is the power of 2 that defines the number of entries in the dictionary. Thus, the length is nine, 10, 11, 12, or 13 bits depending on whether the number of entries is 512, 1K, 2K, 4K, or 8K, respectively.

There is an exception to the above that is described below in the section “Restriction on Identical Child and Sibling Characters.”

The unqualified term “symbol” can mean either a character symbol or an index symbol. The first meaning applies when discussing the uncompressed data or the dictionary entries. The second meaning applies when discussing the compressed data.

Child and Sibling Characters

It is a fundamental goal of the compression call facility to minimize the number of storage references required when processing dictionary entries to find matches. To this end, a character entry never contains its own first extension character and always contains the first extension characters of some number of its children, if any. These first extension characters of the children are named *child characters*. The positional number of a child character in a parent along with the pointer to the first child locates the child. If an entry represents more than one extension character, the characters after the first are in the entry and are called *additional extension characters.*

If a parent has more children than the number of bytes available in the parent to contain child characters, the first extension characters of the additional children are in a *sibling-descriptor* dictionary entry that follows the last child entry able to be represented by means of its child character in the parent. These first extension characters in a sibling descriptor are named *sibling characters*. COMPRESSION CALL provides the option of having either format-0 or format-1 sibling descriptors. A format-0 sibling descriptor is one doubleword, contains up to seven sibling characters, and resides entirely within the compression dictionary. It resides at an index position that would otherwise be used by a character entry. A format-1 sibling descriptor is two doublewords, contains up to 14 sibling characters, and has its first doubleword at an index position in the compression dictionary and its second doubleword at the same index position in the expansion dictionary. This position in the expansion dictionary would otherwise be wasted. This partial residence of sibling descriptors in the expansion dictionary is the only way the expansion dictionary is used during compression.

A sibling descriptor contains from one to seven or 14 sibling characters that correspond to the additional children of the parent. The additional child entries follow the sibling descriptor in the dictionary. If there are more than seven or 14 additional children, another sibling descriptor follows the seventh or fourteenth additional child. This following of seven or 14 children

by another sibling descriptor can be repeated again and again. However, a parent must have no more than 260 children; otherwise, a data exception may be recognized when an attempt is made to find a match on a child after the 260th child.

Child and Extension-Character Combinations

The compression dictionary contains only two types of entries: character entries and sibling descriptors. However, the character entries have different formats depending on how many additional extension characters an entry represents and on whether an entry has zero, one, or more than one children. Only the following combinations are allowed:

- If an entry has zero children, it can have from zero to four additional extension characters.
- If an entry has one chid, it can have from zero to four additional extension characters.
- If an entry has more than one child it can have only zero or one additional extension character.

Character entries and sibling descriptors contain count fields whose contents specify the numbers of additional extension characters, child characters, or sibling characters in the entries and also whether a sibling descriptor follows a child. Character entries and sibling descriptors also contain *examine-child* bits that specify whether child entries, whether designated from a parent or a sibling descriptor, need to be examined. A child entry need not be examined if the entry represents only one extension character and has no children. Conversely, if the entry represents (contains) one or more additional extension characters or has children, the entry must be examined in order to continue the matching process.

Restriction on Identical Child and Sibling Characters

The compression process is described above with some lack of detail, which detail is provided here. However, this section, including its programming notes, does not apply if the order-preservation option is used.

After a match has been found on a parent entry, and if the parent has children, the next character of the string is compared against the child characters in the parent and the sibling characters in the sibling descriptors that are among the children of the parent until a match is found or all child and sibling characters have been processed. If a match is found, the next characters of the string are compared against the additional extension characters, if any, in the child designated by means of the matched child or sibling character. If the additional extension characters match, or if there are no additional extension characters, the matching process is repeated using the matched child as the next parent. If there are additional extension characters that do not match the next characters of the string, then, except in one case, it is model-dependent whether the matching process is ended, with the match on the current parent being the last match, or whether the matching process is continued by attempting to match on a remaining child of the current parent. The one exceptional case is when the designated child and also the next child are in a set of children designated by means of consecutive identical child characters (not sibling characters) beginning with the first child character in the parent. In this case, an attempt is made to match on the following children in the set until either a match is found or all such children have been processed.

The effect of the detailed operation just described is that it is assured that it is useful for two or more identical characters to appear as child characters in a parent or sibling characters in a sibling descriptor under a parent only when the characters are all consecutive child characters beginning with the first child character in the parent. In any case where the identical characters are not consecutive child characters beginning with the first child character in the parent, the second character and any subsequent identical child or sibling character may be wasted since, depending on the model, they may never be compared against a string character equal to them.

Programming Notes:

1. When an examine-child bit indicates that the corresponding child need not be examined, it does not necessarily cause the child not to be examined. Therefore, the count fields in the child must contain all zeros as they correctly should; otherwise, there may be a match on one or more false additional extension characters or a false child character in the child. Since the expansion dictionary will not have been built to recreate the falsely matched character or characters, the

expanded data will not be equal to the original uncompressed data.

2. The restriction against identical child or sibling characters may improve performance because it generally allows the matching process to be ended immediately after a match on a child or sibling character followed by a failure to match on additional extension characters in the designated child. On the other hand, the provision that attempted matching continues in the case of identical consecutive leading child characters improves performance in the case of compressing strings, of many different lengths, of the same repeated character. For example, consider the following two different sets of compression-dictionary entries that might be used for compressing strings, of lengths 1-15, of the character A. (Strings of binary zeros or of blanks are a more important case, but those two characters are unprintable and, therefore, cannot be shown in this example.)

   In the first set of entries (Set 1), each parent has only one child, and there are no additional extension characters -- each child represents only one more character than its parent. Set 1 is shown abstractly as follows:

Set 1

Entry Number	Character Symbol Represented by Entry
1	A
2	AA
3	AAA
4	AAAA
5	AAAAA
6	AAAAAA
7	AAAAAAA
8	AAAAAAAA
9	AAAAAAAAA
10	AAAAAAAAAA
11	AAAAAAAAAAA
12	AAAAAAAAAAAA
13	AAAAAAAAAAAAA
14	AAAAAAAAAAAAAA
15	AAAAAAAAAAAAAAA

   The entries in Set 1 are shown with the first and only child of a parent immediately beneath the parent except indented one character position to the right. Part of the abstraction is that the entries are numbered 1-15 even though, in an actual dictionary, entries 0-255 must be alphabet entries. Note that the number of A's represented by an entry is equal to the number of the entry.

   The number of storage references made to the dictionary during compression may be an important determinant of performance, depending on the model. Using Set 1, the number of dictionary storage references needed to compress a string of 1-15 A's is equal to the number of A's in the string.

   In the second set of entries (Set 2), each parent has two children, the first child has one additional extension character, and the second child has no additional extension character -- the first child represents two more characters than its parent, and the second child represents one more character than its parent. Set 2 is shown abstractly as follows. The following also explicitly shows the number of A's represented by each entry and the number of storage references needed to match on each entry.:

Set 2

Entry Number	Character Symbol Represented by Entry	Stg. A's	Stg. Refs
1	A	1	1
2	AAA	3	2
4	AAAAA	5	3
6	AAAAAAA	7	4
8	AAAAAAAAA	9	5
10	AAAAAAAAAAA	11	6
12	AAAAAAAAAAAAA	13	7
14	AAAAAAAAAAAAAAA	15	8
15	AAAAAAAAAAAAAA	14	9
13	AAAAAAAAAAAA	12	8
11	AAAAAAAAAA	10	7
9	AAAAAAAA	8	6
7	AAAAAA	6	5
5	AAAA	4	4
3	AA	2	3

   The entries in Set 2 are shown with the first child of a parent immediately beneath the parent, except indented one character position to the right, and with the other child of the same parent aligned vertically with the first child. For example, entry 1 (A) has the children 2 (AAA) and 3 (AA), and entry 2 (AAA) has the children 4 (AAAAA) and 5 (AAAA). Entries 3 (AA) and 5 (AAAA) do

not have a child. The entries are numbered as they are because the children of a parent must be numbered consecutively.

It can be seen that except for entries 5 and 3 of Set 2, the number of storage references needed to match on a Set-2 entry is always less than the number of storage references needed to match on the Set-1 entry representing the same number of A's.

A dictionary might contain entries in the style of either Set 1 or Set 2 that would allow compression of a string of up to 260 repeated characters as a single index symbol. Using entries in the style of Set 2, instead of the style of Set 1, reduces the number of required storage references by approximately 50%. (Matching a string of 101 A's would require 101 references using the Set-1 style but only 51 references using the Set-2 style.)

3. Long strings of a repeated character can be compressed with even fewer storage references if entries in the style of Set 3, shown below, are used. Using entries in the style of Set 3, instead of the style of Set 1, reduces the number of required storage references by approximately 67%.

Set 3

Entry Number	Character Symbol Represented by Entry	Stg. A's	Stg. Refs
1	A	1	1
2	AAAAAA	6	2
7	AAAAAAA	7	3
8	AAAAAAAAAAAA	12	4
13	AAAAAAAAAAAAA	13	5
14	AAAAAAAAAAAAAAAAA	18	6
15	AAAAAAAAAAAAAAAA	17	7
16	AAAAAAAAAAAAAAA	16	8
17	AAAAAAAAAAAAAA	15	9
18	AAAAAAAAAAAAA	14	10
9	AAAAAAAAAA	11	5
10	AAAAAAAAA	10	6
11	AAAAAAAA	9	7
12	AAAAAAA	8	8
3	AAAAA	5	3
4	AAAA	4	4
5	AAA	3	5
6	AA	2	6

In Set 3, an odd-level parent has no additional extension character and five children, and the children, in order, have four, three, two, one, and zero additional extension characters -- the children represent from five to one more characters than their parent. An even-level parent, which is the first child of an odd-level parent, has four additional extension characters and one child, which is the next odd-level parent. The second through fifth children of an oddlevel parent do not have a child. Specifically in Set 3, entry 1 (A) has the children 2 (AAAAAA), 3 (AAAAA), 4 (AAAA), 5 (AAA), and 6 (AA); entry 2 (AAAAAA) has the child 7 (AAAAAAA); entry 7 has the children 8-12; entry 8 has the child 13, and entry 13 has the children 14-18. Entries 3-6, 9-12, and 15-18 do not have a child. Entry 14 might have a child to continue the pattern.

Matching a string of 18 A's requires six storage references, which is 33% of the number required when using the Set-1 style.

Entries should not be arranged as in the following examples.

Set 4 (Erroneous)

Entry Number	Character Symbol Represented by Entry
1	A
2	AA
3	AAA

Entry 3 in Set 4 is useless because if the string to be matched is AAA, a match of the AA at the beginning of the string will be found on entry 2, and entry 3 will not be examined. (This would be true also if the string to be matched were AAB and entry 3 represented AAB.)

Set 5 (Erroneous)

Entry Number	Character Symbol Represented by Entry
1	A
2	AAA
3	AB
4	AA

Entry 4 in Set 5 may be useless because if the string to be matched is AAX, the matching process may end, depending on the model, after the failure to match the AAX to entry 2. (Entries 2 and 4 are designated by identical child charac-

ters in entry 1, and the first of these child characters is the first child character in entry A, but the two child characters are not consecutive.)

Set 5 (Erroneous)

Entry Number	Character Symbol Represented by Entry
1	A
2	AB
3	AAA
4	AA

Entry 4 in Set 6 may be useless because if the string to be matched is AAX, the matching process may end, depending on the model, after the failure to match the AAX to entry 3. (Entries 3 and 4 are designated by identical consecutive child characters in entry 1, but the first of these child characters is not the first child character in entry A.)

Expansion-Dictionary Entries

The expansion dictionary contains three types of entries: *unpreceded character entries*, *preceded character entries*, and the second halves of the format-1 sibling descriptors that begin in the compression dictionary. An unpreceded character entry represents a character symbol by containing all of the characters of the symbol -- these are the extension characters represented by the corresponding entry in the compression dictionary and the extension characters represented by all the ancestors of that compression-dictionary entry. An unpreceded character entry can contain from one to seven characters. A preceded character entry represents a character symbol by containing from one to five of the rightmost characters of the symbol and also a pointer, called a *predecessor pointer*, to another expansion-dictionary character entry that contains some or all of the next lefthand characters of the character symbol. That is, the predecessor pointer in a preceded character entry can point to either an unpreceded or a preceded character entry.

A predecessor pointer is not necessarily a parent pointer, which term has no role in this definition. That is, a predecessor pointer may designate a higher-level entry that is more remote than simply a parent.

An unpreceded character entry contains a complete-symbol-length field whose contents specify the number of characters in the entry. A preceded character entry contains a partial-symbol-length field whose contents specify the number of characters in the entry, and it also contains an offset that may be used to position the leftmost character of the entry in the expanded-data operand.

Expansion Process

Expansion of an index symbol occurs in a way that depends on whether the value of the symbol is in the range 0-255 or is 256 or larger. If the value of the index symbol is in the range 0-255, a byte containing the value of the symbol is simply stored as the next byte of expanded data, and no reference is made to the expansion dictionary. That is, the index symbol is assumed to designate an expansion-dictionary entry that represents the same character symbol as does the alphabet entry that is designated by the index symbol in the compression dictionary -- the index symbol can be said to designate an alphabet entry in the expansion dictionary. If the value of the index symbol is 256 or larger, expansion occurs by using the value as an index to locate the corresponding expansion-dictionary character entry and then (1) outputting the characters in the entry, and (2) if the entry is a preceded entry, proceeding up the chain of preceded entries and outputting the characters in each of them on the left of the characters already output, until finally an unpreceded entry has been encountered. Following is a more detailed description of the case when the value of the index symbol is 256 or larger.

1. The rightmost byte of the first preceded character entry obtained during the expansion of an index symbol contains the byte offset from the current location in the expanded-data operand to where the first extension character represented by the character entry is to be placed. (All preceded character entries contain an offset since any entry can be the first one obtained during an expansion.) The address of the current location is named the current origin address. At the end of the operation, the current origin address is updated in accordance with one of two methods, and which method is used is unpredictable. When the first method is used, the current origin address is updated by adding to it the sum of the partial symbol length and the offset in the first-obtained preceded entry. When the second method is used, the current origin address is updated by adding to it the number of characters

placed in the first operand location due to the expansion of the index symbol. If the expansion-dictionary entries processed specify a logically correct operation, the results are the same regardless of which method is used. The entries are considered to specify a logically correct operation if they together contain n characters and cause placement of those n characters, without overlaps or spaces, in the n character positions beginning at the original current origin address and ending at the address that is one less than the updated current origin address. The machine does not check in any way that the operation is logically correct.

2. If the first character entry obtained during expansion of an index symbol is an unpreceded entry, the characters in the entry are placed at the current origin address, and then the current origin address is updated by adding to it the complete symbol length in the unpreceded entry.

3. When the first character entry obtained during expansion of an index symbol is a preceded entry, the processing of the preceding entries, which may include additional preceded entries, is in accordance with one of two methods, and which method is used is unpredictable. When the first method is used, the processing is the same as in items 1 and 2; that is, the characters in a preceded entry are placed as determined by the offset in the entry, and the characters in an unpreceded entry are placed at the current origin address. When the second method is used, the characters in each preceding entry are placed immediately to the left of the characters resulting from the previously processed preceded entry. Either the same or different methods may be used for each of preceded and unpreceded entries and for each of multiple preceded entries. If the entries processed specify a logically correct operation, the results are the same regardless of which method is used. The updated current origin address is determined as specified in item 1. If the operation is logically incorrect and leaves spaces in the n character positions beginning at the original current origin address, the contents of those spaces are unpredictable.

   When method 1 is used, storing may be specified to occur at or to the right of the updated current origin address because either the offset in a subsequent preceded entry is larger than the offset in the first one or because the offset in the first one is too small, considering the complete symbol length in the unpreceded entry. (The preceding is incomplete because it does not deal with partial symbol lengths, but the point should be clear.) When method 2 is used, storing may be specified to occur to the left of the original current origin address because the offset in the first preceded entry is too small. In the method-1 or method-2 case, such storing may be within or outside the first-operand location. It may cause a data exception to be recognized, or it may be attempted. If it is attempted, it may cause an access exception to be recognized and the operation to be nullified or suppressed, depending on the exception. If an access exception is recognized, the results may not be true nullification or suppression because storing may have occurred in a location for which no access exception was recognized. Recognition of an access exception may cause condition code 3 to be set instead of being treated as a program-interruption condition.

4. When entropy-encoding is not enabled and the value of a predecessor pointer is in the range 0-255, either the pointer may be used to reference a dictionary entry as already described, or the value may simply be stored as the next byte of expanded data. When entropy-encoding is enabled and the value of a predecessor pointer is in the range 0-255 the pointer is always used to reference a dictionary entry as already described.

Programming Note: If no predecessor pointer designates one of the entries 0-255 in the expansion dictionary, those entries need not actually be provided, and the storage that would be occupied by the entries can be used for other purposes. However, the storage must exist and have the correct storage key since the CPU may pretest the entire expansion dictionary, including entries 0-255, for access exceptions.

Compressed-Data Symbol Size

The bits in general register 0 that specify the number of bits in an index symbol and the size in entries and bytes of a dictionary are named the compressed-data symbol size (CDSS).

Symbol Translation

During compression, symbol translation can be specified by means of a bit in general register 0; this bit is ignored during expansion. When symbol translation is specified, the index symbols generated by the compression process are used to locate two-byte entries in a symbol-translation table (STT), and *interchange symbols* are taken from the STT entries and placed in the compressed-data operand. The beginning of the STT is designated by the address and an offset in general register 1.

When symbol-translation is specified, the CDSS does not specify either the number of bits in an index symbol or the size of the compression dictionary; the CDSS specifies the number of bits in an interchange symbol, the compression dictionary is considered to extend to the beginning of the STT, and the size of the STT in bytes is considered to be one fourth that of the compression dictionary.

Symbol translation is not performed during (before) expansion because the expansion dictionary can be formed to directly expand interchange symbols, that is, during expansion, interchange symbols and index symbols are considered to be identical, and only the term index symbol is used.

Symbol translation and format-1 sibling descriptors must not both be specified; otherwise, the results are unpredictable.

Programming Note: Symbol translation is for use by VTAM. During compression, VTAM will form an adaptive dictionary until compression becomes good, and VTAM will then freeze its dictionary and transform it to the architected form used by COMPRESSION CALL. VTAM will also signal the other end of the session to freeze its dictionary also, but the other end will then continue to use its frozen adaptive dictionary. Since the architected dictionary must contain some entries that are sibling-descriptor entries while the adaptive dictionary has no similar requirement (all of its entries are character entries), most of the entries in the architected dictionary cannot have the same indexes as the logically corresponding entries in the adaptive dictionary, and symbol translation must be used to generate the correct interchange symbols.

Order Preservation

During compression, order preservation can be specified by means of a bit in general register 0. When order preservation is specified, the symbol-translation bit in the register is ignored, but a modified form of symbol translation occurs.

When an appropriate compression dictionary and symbol-translation table are used, order preservation causes, for successive units of uncompressed data that are in collating-sequence order, production of corresponding successive units of compressed data that also are in collating-sequence order.

The compression process used in the basic compression operation uses what is referred to here as the *unordered* comparison algorithm: the children of a parent are not considered to be in any particular order, and the next characters from the string being compressed are compared against the extension characters represented by the children in the left-to-right order of the children until either a match is found or all of the children have been processed. The result of the compression process is the index symbol that designates the last matched entry, which is the parent entry if no match was found on any child or there are no children.

The order-preservation option causes the compression process to use an *ordered* comparison algorithm: the children of a parent are considered to be ordered such that the string of one or more extension characters represented by any child is always earlier in the collating sequence than the string of one or more extension characters represented by the next child of the same parent, that is, the children are considered to be in collating-sequence order. The next characters from the string being compressed are compared against the extension characters represented by the children in the left-to-right order of the children until any of the following is true: (1) a match is found on an entry without children, or a match is found on an entry with children but there is not another character in the string; (2) the next characters from the string have a collating-sequence value less than that of the extension characters represented by a child of the last matched parent; or (3) the next characters from the string have a collating-sequence value greater than that of the extension characters represented by the last child of the last matched parent. The result of the compression process in each of the three cases is as follows:

1. If the process ends because of a match on an entry without children or because of a match on an entry when there is not another character in the string, the result is the index symbol that designates the entry.

2. If the process ends because the next characters of the string have a value less than that of the characters represented by a child, the result is the index symbol that designates the child.

3. If the process ends because the next characters of the string have a value greater than that of the characters represented by the last child, the result is the index symbol that designates the last child.

Note that Case 2 includes the case when the next character of the string match the leftmost extension characters represented by a child but there are not as many next characters remaining in the string as there are additional extension characters represented in the child.

The index symbol produced by the compression process is used to locate a symbol-translation-table entry containing an interchange symbol just as when the symbol-translation option is specified. The resulting interchange symbol that is placed in the compressed-data operand is then as follows:

1. In Case 1, the interchange symbol in the entry.

2. In Case 2, the interchange symbol in the entry, minus one; that is, one is subtracted from the value of the interchange symbol that is in the entry to form the interchange symbol that is stored.

3. In Case 3, the interchange symbol in the entry, plus one; that is, one is added to the value of the interchange symbol that is in the entry to form the interchange symbol that is stored.

Although the order-preservation option does not affect the expansion process, the expansion dictionary must be formed so that the interchange symbols produced by compression are properly expanded when they are treated as index symbols during expansion. During expansion, an index symbol equal to an interchange symbol produced during compression must designate a dictionary entry that represents the same character symbol as did the last entry that was matched during the compression. In Case 2 or 3, the last matched entry is the parent of the entry that caused the Case 2 or Case 3 condition to be satisfied.

Programming Note: The assumption that an index symbol in the range 0-255 designates an alphabet entry in the expansion dictionary requires that interchange symbols be assigned as follows when the order-preservation option is used. If n is the number of the lowest-numbered alphabet entry in the compression dictionary having a child, then the interchange symbols 0-n must be assigned to alphabet entries 0-n, the interchange symbol 256 must be assigned to the first epsilon entry or child under entry n, and the interchange symbols (n+1)-255 must not be assigned.

Entropy Encoding

Entropy encoding can be specified by means of a bit in general register 0. When entropy encoding is specified, a higher degree of compression may be obtained due to using shorter codewords for more frequent symbols.

The compressed operand does not contain index symbols, instead it contains codewords. The codewords may vary from 1-16 bits in length. Each codeword represents an index symbol. The most frequent index symbols will have codewords with fewer bits, and less frequent index symbols will have codewords with more bits.

The mapping of index symbols to codewords is handled differently for compression and expansion. However, the entropy descriptor is common. The entropy descriptor describes the valid codewords from most frequent to least frequent. The entropy descriptor consists of sixteen halfword entries each of which contain a count of the number of entries of the corresponding bit length from one to sixteen.

The mapping of index symbols to codewords is done by generating a prefix tree for all index symbols. The prefix tree is formed using the frequency that each index symbol is seen when compressing data. The codewords are grouped by their information content and enumerated in descending order. This is sometimes called a cannonical Huffman code or an Shannon-Fano code. Since all of the codewords are ordered, it is possible to describe the tree by enumerating the number of entries of each codeword length. The maximum codeword size supported is 16-bits. The entropy descriptor must fully represent a valid

tree. The sum of all of the entries in the entropy descriptor must add up to the number of dictionary entries plus one; otherwise, a general-operand data exception is recognized. Also there cannot be any underflow or overflow when computing the lower and upper bound codewords for each bit length otherwise a general-operand data exception is recognized.

The entropy description is located immediately after the compression or expansion dictionary and is pointed to by the offset field in general register 1. During compression a symbol translation table is located immediately after the entropy descriptor. The symbol translation table is one fourth the size of the compression dictionary.

During compression index symbols which are formed during compression are used to index into the symbol translation table. Each halfword in the symbol translation table contains the codeword which maps to the index symbol left aligned in the halfword. Any unused bits in the symbol translation table entry must contain zeros; otherwise, a general-operand data exception is recognized. The number of bits from the symbol translation table entry to be copied to the first operand location is computed for each symbol translation table entry.

During expansion the bits of the second operand are broken into codewords using the information in the entropy description. The codewords are converted to sequence numbers and used to index directly into the expansion dictionary. Indices 0-255 have no special meaning in the expansion dictionary when the entropy encoding option is specified. Since the entries in the expansion dictionary are indexed by sequence numbers they must be ordered by likelihood. The sequence number is the number of codewords with a higher probability than the current input codeword.

Results of Dictionary Errors

The following may be truncated on the left to eliminate bits not needed because of the dictionary size, or they may always be treated as 13 bits regardless of the dictionary size: child pointers, predecessor pointers, and index symbols that are used to locate entries in a symbol-translation table. If the pointers or symbols are not truncated and a child pointer specifies an entry beyond the end of the compression dictionary, a predecessor pointer specifies an entry beyond the end of the expansion dictionary, or an index symbol specifies an entry beyond the end of the symbol-translation table, any of the following may occur: in the case of a child pointer and a zero examine-child bit, an index symbol containing too many bits may be generated and stored in the first-operand location (possibly overlaying a rightmost bit or bits of the previous index symbol); the contents at the improper location may be used in the normal way as a dictionary or table entry; or a data exception may be recognized. If the contents are to be used in the normal way, an access exception can be recognized for the improper location even after a store has occurred in the first-operand location. Such an access exception does not result in true nullification or suppression, except that the instruction address in the PSW is correct for the type of exception recognized. When order-preservation is specified, the results when dictionary entries are not in collating-sequence order are unpredictable and may include recognition of a data exception.

Dictionary Formats

Notation

(m-n) Value allowed in the field

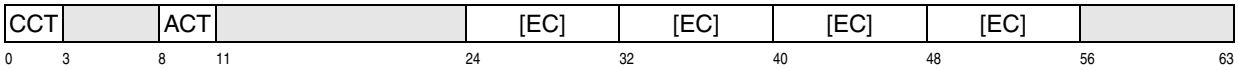
[EC] Extension character may be present

Three periods (...) means the preceding field may be repeated.

Compression Dictionary

Character Entry

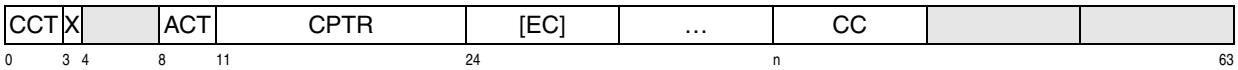
When CCT = 0



ACT — Additional-extension-character count (0-4); number of ECs in entry. if ACT is 5, 6, or 7, it may be treated as 4, may cause data exception, or may cause comparison to mod(ACT,4) additional entries that are obtained from unpredictable locations in the current entry or from storage locations immediately following the current entry; see Note 1

EC — Additional extension character

When CCT = 1



X — examine-child bit; child has one or more additional extension characters or has children; see Note 2

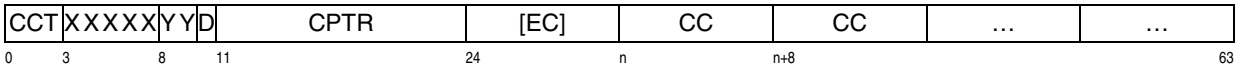
ACT — Additional-extension-character count (0-4). if ACT is 5, 6, or 7, it may be treated as 4, may cause data exception, or may cause comparison to mod(ACT,4) additional entries that are obtained from unpredictable locations in the current entry or from storage locations immediately following the current entry.

CPTR — Index of first child

EC — Additional extension character(s)

CC — Child character

When CCT >1



CCT — Child count (2-6 if D=0, 2-5 if D=1). If D=0 and CCT=6, sibling descriptor follows 5th child (CCT = 7 may be treated as 6 or it may cause a data exception). If D=1 and CCT=5, sibling descriptor follows 4th child ACT (CCt = 6 or 7 may be treated as 5 or it may cause a data exception)

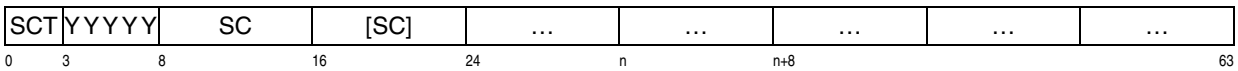
XXXXX — Examine-child bits for children 1-5 (D=0) or 1-4 (D=1). Bit(s) ignored if respective child does not exist.

YY — Examine child bits for 6th & 7th siblings designated by first format-0 sibling descriptor for children of this entry; or examine-child-bits for 13th and 14th siblings designated by first format-1 sibling descriptor for children of this entry. Bit(s) ignored if sibling does not exist.

EC — Additional extension character(s)

CC — Child character

Format-0 Sibling Descriptor

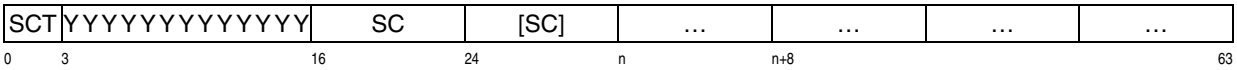


SCT — Sibling Count (0-7); zero is treated as 7 and indicates another SD follows 7th sibling

YYY… — Examine-child bits for siblings 1-5; bit is ignored if sibling does not exist; if this is the first sibling descriptor under parent, bits for siblings 6 and 7 are in parent — otherwise, 6th and 7th siblings must be examined since bits for them are not provided.

SC — Sibling character

Format-1 Sibling Descriptor



SCT Sibling Count (1-15); 15 is treated as 14 and indicates another SD follows 14th sibling. 0 may be treated as 15 or cause a data exception)

YYY… Examine-child bits for siblings 1-12; bit is ignored if sibling does not exist; if this is the first sibling descriptor under parent, bits for siblings 13 and 14 are in parent — otherwise, 1‘3th and 14th siblings must be examined since bits for them are not provided.

SC Sibling character

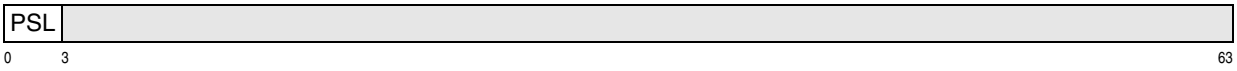
Notes:

1. If ACT or D is nonzero in alphabet entry, any of the following may result:
   - ACT or D is treated as zero.
   - ACT or D does not cause comparison to additional extension characters but does determine the position of the first child character.
   - ACT or D is treated the same as in a nonalphabet entry except that, if this causes a mismatch on the next characters to be compressed, a data exception is recognized.
2. A child may be examined even if its examine-child bit (X or Y) is zero. If an examine-child bit (X or Y) is one and the corresponding child or sibling exists, but it is not true that the child or sibling either has one or more additional extension characters or has children, a data exception may be recognized.

Expansion Dictionary

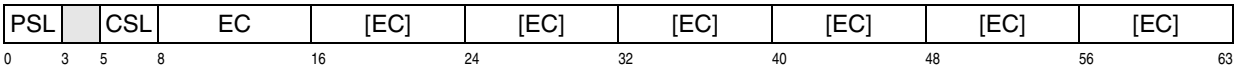
Character Entry

Generic Form



PSL Partial-symbol length (0-5); number of ECs in entry; 0 indicates an unpreceded entry; 6 and 7 may be treated as 5, may cause a data exception, or may cause movement to the first operand location of one or two, respectively, additional ECs that are obtained from unpredictable locations in the current entry or from the storage locations immediately following the current entry.

If PSL=0 (Unpreceded Entry)

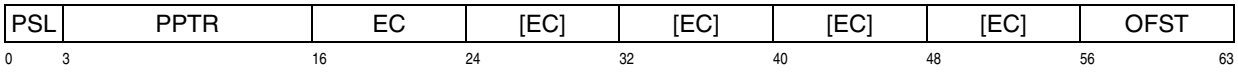


CSL Complete-symbol length (1-7); number of ECs in the entry; 0 may cause a data exception or may cause movement to the first operand location of up to 256 ECs that are obtained from unpredictable locations in the current entry or from storage locations immediately following the current entry

EC Extension character

Bits 3 and 4 of the entry should be zeros; otherwise, any of the following may occur: the bits may be ignored, they may cause a data exception, or they may cause movement to the first operand location of up to 24 additional ECs that are obtained from unpredictable locations in the current entry or from the storage locations immediately following the current entry.

If PSL>0 (Preceded Entry)



PSL Partial-symbol length (1-5); number of ECs in the entry; 0 indicates an unpreceded entry; 6 and 7 may be treated as 5, may cause a data exception, or may cause movement to the first operand location of one or two, respectively, additional ECs that are obtained from unpredictable locations in the current entry or from the storage locations immediately following the current entry.

PPTR Predecessor pointer; index of preceding entry that should contain the next left-hand characters of the character symbol represented by this entry

OFST Offset (0-255) from the current position in the output area to where first EC in this entry is placed; however, for preceded entries after the first one, either method 1 or method 2 may be used -- see the text

When expansion is completed, the output pointer is advanced either by the sum of the PSL and OFST in the first-encountered preceded entry or by the number of characters placed in the first-operand location due to expansion of the index symbol. If the first method is used and PSL is 6 or 7, either 5 or the actual value of PSL may be used.

There is no checking for logical correctness of the operation.

Format-1 Sibling Descriptor

The expansion dictionary contains bits 64-127 of the format-1 sibling descriptors that begin in the compression dictionary.

Appendix G. Table of Powers of 2

Plus		Minus
1	0	1.
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.0625
32	5	0.03125
64	6	0.01562 5
128	7	0.00781 25
256	8	0.00390 625
512	9	0.00195 3125
1,024	10	0.00097 65625
2,048	11	0.00048 82812 5
4,096	12	0.00024 41406 25
8,192	13	0.00012 20703 125
16,384	14	0.00006 10351 5625
32,768	15	0.00003 05175 78125
65,536	16	0.00001 52587 89062 5
131,072	17	0.00000 76293 94531 25
262,144	18	0.00000 38146 97265 625
524,288	19	0.00000 19073 48632 8125
1,048,576	20	0.00000 09536 74316 40625
2,097,152	21	0.00000 04768 37158 20312 5
4,194,304	22	0.00000 02384 18579 10156 25
8,388,608	23	0.00000 01192 09289 55078 125
16,777,216	24	0.00000 00596 04644 77539 0625
33,554,432	25	0.00000 00298 02322 38769 53125
67,108,864	26	0.00000 00149 01161 19384 76562 5
134,217,728	27	0.00000 00074 50580 59692 38281 25
268,435,456	28	0.00000 00037 25290 29846 19140 625
536,870,912	29	0.00000 00018 62645 14923 09570 3125
1,073,741,824	30	0.00000 00009 31322 57461 54785 15625
2,147,483,648	31	0.00000 00004 65661 28730 77392 57812 5
4,294,967,296	32	0.00000 00002 32830 64365 38696 28906 25
8,589,934,592	33	0.00000 00001 16415 32182 69348 14453 125
17,179,869,184	34	0.00000 00000 58207 66091 34674 07226 5625
34,359,738,368	35	0.00000 00000 29103 83045 67337 03613 28125
68,719,476,736	36	0.00000 00000 14551 91522 83668 51806 64062 5
137,438,953,472	37	0.00000 00000 07275 95761 41834 25903 32031 25
274,877,906,944	38	0.00000 00000 03637 97880 70917 12951 66015 625
549,755,813,888	39	0.00000 00000 01818 98940 35458 56475 83007 8125
1,099,511,627,776	40	0.00000 00000 00909 49470 17729 28237 91503 90625
2,199,023,255,552	41	0.00000 00000 00454 74735 08864 64118 95751 95312 5
4,398,046,511,104	42	0.00000 00000 00227 37367 54432 32059 47875 97656 25
8,796,093,022,208	43	0.00000 00000 00113 68683 77216 16029 73937 98828 125
17,592,186,044,416	44	0.00000 00000 00056 84341 88608 08014 86968 99414 0625
35,184,372,088,832	45	0.00000 00000 00028 42170 94304 04007 43484 49707 03125
70,368,744,177,664	46	0.00000 00000 00014 21085 47152 02003 71742 24853 51562 5
140,737,488,355,328	47	0.00000 00000 00007 10542 73576 01001 85871 12426 75781 25
281,474,976,710,656	48	0.00000 00000 00003 55271 36788 00500 92935 56213 37890 625
562,949,953,421,312	49	0.00000 00000 00001 77635 68394 00250 46467 78106 68945 3125
1,125,899,906,842,624	50	0.00000 00000 00000 88817 84197 00125 23233 89053 34472 65625
2,251,799,813,685,248	51	0.00000 00000 00000 44408 92098 50062 61616 94526 67236 32812 5
4,503,599,627,370,496	52	0.00000 00000 00000 22204 46049 25031 30808 47263 33618 16406 25
9,007,199,254,740,992	53	0.00000 00000 00000 11102 23024 62515 65404 23631 66809 08203 125
18,014,398,509,481,984	54	0.00000 00000 00000 05551 11512 31257 82702 11815 83404 54101 5625
36,028,797,018,963,968	55	0.00000 00000 00000 02775 55756 15628 91351 05907 91702 27050 78125
72,057,594,037,927,936	56	0.00000 00000 00000 01387 77878 07814 45675 52953 95851 13525 39062 5
144,115,188,075,855,872	57	0.00000 00000 00000 00693 88939 03907 22837 76476 97925 56762 69531 25
288,230,376,151,711,744	58	0.00000 00000 00000 00346 94469 51953 61418 88238 48962 78381 34765 625
576,460,752,303,423,488	59	0.00000 00000 00000 00173 47234 75976 80709 44119 24481 39190 67382 8125
1,152,921,504,606,846,976	60	0.00000 00000 00000 00086 73617 37988 40354 72059 62240 69595 33691 40625
2,305,843,009,213,693,952	61	0.00000 00000 00000 00043 36808 68994 20177 36029 81120 34797 66845 70312 5
4,611,686,018,427,387,904	62	0.00000 00000 00000 00021 68404 34497 10088 68014 90560 17398 83422 85156 25
9,223,372,036,854,775,808	63	0.00000 00000 00000 00010 84202 17248 55044 34007 45280 08699 41711 42578 125

Figure G-1. Powers of 2

© Copyright IBM Corp. 2000, 2017

Plus		Minus
18,446,744,073,709,551,616	64	0.00000 00000 00000 00005 42101 08624 27522 17003 72640 04349 70855 71289 0625
36,893,488,147,419,103,232	65	
73,786,976,294,838,206,464	66	
147,573,952,589,676,412,928	67	
295,147,905,179,352,825,856	68	
590,295,810,358,705,651,712	69	
1,180,591,620,717,411,303,424	70	
2,361,183,241,434,822,606,848	71	
4,722,366,482,869,645,213,696	72	
9,444,732,965,739,290,427,392	73	
18,889,465,931,478,580,854,784	74	
37,778,931,862,957,161,709,568	75	
75,557,863,725,914,323,419,136	76	
151,115,727,451,828,646,838,272	77	
302,231,454,903,657,293,676,544	78	
604,462,909,807,314,587,353,088	79	
1,208,925,819,614,629,174,706,176	80	
2,417,851,639,229,258,349,412,352	81	
4,835,703,278,458,516,698,824,704	82	
9,671,406,556,917,033,397,649,408	83	
19,342,813,113,834,066,795,298,816	84	
38,685,626,227,668,133,590,597,632	85	
77,371,252,455,336,267,181,195,264	86	
154,742,504,910,672,534,362,390,528	87	
309,485,009,821,345,068,724,781,056	88	
618,970,019,642,690,137,449,562,112	89	
1,237,940,039,285,380,274,899,124,224	90	
2,475,880,078,570,760,549,798,248,448	91	
4,951,760,157,141,521,099,596,496,896	92	
9,903,520,314,283,042,199,192,993,792	93	
19,807,040,628,566,084,398,385,987,584	94	
39,614,081,257,132,168,796,771,975,168	95	
79,228,162,514,264,337,593,543,950,336	96	
158,456,325,028,528,675,187,087,900,672	97	
316,912,650,057,057,350,374,175,801,344	98	
633,825,300,114,114,700,748,351,602,688	99	
1,267,650,600,228,229,401,496,703,205,376	100	
2,535,301,200,456,458,802,993,406,410,752	101	
5,070,602,400,912,917,605,986,812,821,504	102	
10,141,204,801,825,835,211,973,625,643,008	103	
20,282,409,603,651,670,423,947,251,286,016	104	
40,564,819,207,303,340,847,894,502,572,032	105	
81,129,638,414,606,681,695,789,005,144,064	106	
162,259,276,829,213,363,391,578,010,288,128	107	
324,518,553,658,426,726,783,156,020,576,256	108	
649,037,107,316,853,453,566,312,041,152,512	109	
1,298,074,214,633,706,907,132,624,082,305,024	110	
2,596,148,429,267,413,814,265,248,164,610,048	111	
5,192,296,858,534,827,628,530,496,329,220,096	112	
10,384,593,717,069,655,257,060,992,658,440,192	113	
20,769,187,434,139,310,514,121,985,316,880,384	114	
41,538,374,868,278,621,028,243,970,633,760,768	115	
83,076,749,736,557,242,056,487,941,267,521,536	116	
166,153,499,473,114,484,112,975,882,535,043,072	117	
332,306,998,946,228,968,225,951,765,070,086,144	118	
664,613,997,892,457,936,451,903,530,140,172,288	119	
1,329,227,995,784,915,872,903,807,060,280,344,576	120	
2,658,455,991,569,831,745,807,614,120,560,689,152	121	
5,316,911,983,139,663,491,615,228,241,121,378,304	122	
10,633,823,966,279,326,983,230,456,482,242,756,608	123	
21,267,647,932,558,653,966,460,912,964,485,513,216	124	
42,535,295,865,117,307,932,921,825,928,971,026,432	125	
85,070,591,730,234,615,865,843,651,857,942,052,864	126	
170,141,183,460,469,231,731,687,303,715,884,105,728	127	
340,282,366,920,938,463,463,374,607,431,768,211,456	128	

Figure G-1. Powers of 2

Appendix H. Hexadecimal Tables

The following tables aid in converting hexadecimal values to decimal values, or the reverse.

Direct Conversion Table

This table provides direct conversion of decimal and hexadecimal numbers in these ranges:

Hexadecimal	Decimal
000 to FFF	0000 to 4095

To convert numbers outside these ranges, and to convert fractions, use the hexadecimal and decimal conversion tables that follow the direct conversion table in this appendix.

© Copyright IBM Corp. 2000, 2017

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00_	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
01_	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
02_	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
03_	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
04_	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
05_	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
06_	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
07_	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
08_	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
09_	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A_	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B_	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C_	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D_	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E_	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F_	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255
10_	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
11_	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
12_	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
13_	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
14_	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
15_	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
16_	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
17_	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
18_	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
19_	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A_	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B_	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C_	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D_	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E_	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F_	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
20_	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
21_	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
22_	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
23_	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
24_	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
25_	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
26_	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
27_	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
28_	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
29_	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A_	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B_	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C_	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D_	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E_	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F_	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
30_	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
31_	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
32_	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
33_	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
34_	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
35_	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
36_	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
37_	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
38_	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
39_	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A_	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B_	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C_	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D_	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E_	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F_	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
40_	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
41_	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
42_	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
43_	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
44_	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
45_	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
46_	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
47_	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
48_	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
49_	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A_	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B_	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C_	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4D_	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E_	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F_	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
50_	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
51_	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
52_	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
53_	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
54_	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
55_	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
56_	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
57_	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
58_	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
59_	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A_	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5B_	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C_	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D_	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E_	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F_	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
60_	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
61_	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
62_	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
63_	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
64_	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
65_	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
66_	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
67_	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
68_	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
69_	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A_	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B_	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6C_	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D_	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E_	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F_	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791
70_	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
71_	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
72_	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
73_	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
74_	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
75_	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
76_	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
77_	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
78_	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
79_	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A_	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B_	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C_	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D_	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E_	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F_	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80_	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
81_	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
82_	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
83_	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
84_	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
85_	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
86_	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
87_	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
88_	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
89_	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A_	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B_	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C_	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D_	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E_	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F_	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
90_	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
91_	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
92_	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
93_	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
94_	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
95_	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
96_	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
97_	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
98_	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
99_	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A_	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B_	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C_	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D_	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E_	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F_	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559
A0_	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A1_	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A2_	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A3_	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A4_	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A5_	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A6_	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A7_	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A8_	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A9_	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA_	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB_	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC_	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
AD_	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE_	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF_	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B0_	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B1_	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B2_	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B3_	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B4_	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B5_	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B6_	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B7_	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B8_	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B9_	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA_	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB_	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC_	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD_	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE_	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF_	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C0_	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C1_	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C2_	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C3_	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C4_	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C5_	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C6_	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C7_	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C8_	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C9_	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA_	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB_	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC_	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD_	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE_	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF_	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327
D0_	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D1_	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D2_	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D3_	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D4_	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D5_	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D6_	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D7_	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D8_	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D9_	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA_	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB_	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC_	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD_	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE_	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF_	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E0_	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E1_	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E2_	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E3_	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E4_	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E5_	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E6_	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E7_	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E8_	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E9_	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA_	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB_	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC_	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED_	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE_	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF_	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F0_	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F1_	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F2_	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F3_	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F4_	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F5_	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F6_	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F7_	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F8_	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F9_	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA_	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB_	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC_	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD_	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE_	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF_	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

Conversion Table: Hexadecimal and Decimal Integers

Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,879,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,648	8	134,217,728	8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,919,104	9	150,994,944	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,658,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
8		**7**		**6**		**5**		**4**		**3**		**2**		**1**	

TO CONVERT HEXADECIMAL TO DECIMAL

1. Locate the column of the decimal numbers corresponding to the leftmost digit or letter of the hexadecimal; select from this column and record the number that corresponds to the position of the hexadecimal digit or letter.
2. Repeat step 1 for the next (second from the left) position.
3. Repeat step 1 for the units (third from the left) position.
4. Add the numbers selected from the table to form the decimal number.

EXAMPLE

Conversion of Hexadecimal Value		D34
1.	D	3328
2.	3	48
3.	4	+ 4
4.	Decimal	3380

To convert integer numbers greater than the capacity of the table, use the techniques below:

HEXADECIMAL TO DECIMAL

Successive cumulative multiplication from left to right, adding units position.

Example: $D34_{16} = 3380_{10}$

D = 13
x 16
208
3 = + 3
211
x 16
3376
4 = + 4
3380

TO CONVERT DECIMAL TO HEXADECIMAL

1. (a) Select from the table the highest decimal number that is equal to or less than the number to be converted.
   (b) Record the hexadecimal of the column containing the selected number.
   (c) Subtract the selected decimal from the number to be converted.
2. Using the remainder from step 1(c), repeat all of step 1 to develop the second position of the hexadecimal (and a remainder)
3. Using the remainder from step 2, repeat all of step 1 to develop the units position of the hexadecimal.
4. Combine the terms to form the hexadecimal number.

EXAMPLE

Conversion of Decimal Value		3380
1.	D	-3328
		52
2.	3	- 48
		4
3.	4	- 4
4.	Hexadecimal	D34

DECIMAL TO HEXADECIMAL

Divide and collect the remainder in reverse order

Example: $3380_{10} = X_{16}$

		Remainder
16 | 3380 →	4	↑
16 | 211 →	3	|
16 | 13 →	D	|

$3380_{10} = D34_{16}$

POWERS OF 16 TABLE

Example: $268,435,456_{10} = (2.68435456 \times 10^8)_{10} = 1000\ 0000_{16} = (10^7)_{16}$

16^n	n
1	0
16	1
256	2
4 096	3
65 536	4
1 048 576	5
16 777 216	6
268 435 456	7

16^n	n	
4 294 967 296	8	
68 719 476 736	9	
1 099 511 627 776	10	= A
17 592 186 044 416	11	= B
281 474 976 710 656	12	= C
4 503 599 627 370 496	13	= D
72 057 594 037 927 936	14	= E
1 152 921 504 606 846 976	15	= F

← Decimal Values →

Conversion Table: Hexadecimal and Decimal Fractions

HALFWORD													
BYTE					BYTE								
Bits 0123		4567			0123				4567				
Hex	Decimal	Hex	Decimal		Hex	Decimal			Hex	Decimal Equivalent			
.0	.0000	.00	.0000	0000	.000	.0000	0000	0000	.0000	.0000	0000	0000	0000
.1	.0625	.01	.0039	0625	.001	.0002	4414	0625	.0001	.0000	1525	8789	0625
.2	.1250	.02	.0078	1250	.002	.0004	8828	1250	.0002	.0000	3051	7578	1250
.3	.1875	.03	.0117	1875	.003	.0007	3242	1875	.0003	.0000	4577	6367	1875
.4	.2500	.04	.0156	2500	.004	.0009	7656	2500	.0004	.0000	6103	5156	2500
.5	.3125	.05	.0195	3125	.005	.0012	2070	3125	.0005	.0000	7629	3945	3125
.6	.3750	.06	.0234	3750	.006	.0014	6484	3750	.0006	.0000	9155	2734	3750
.7	.4375	.07	.0273	4375	.007	.0017	0898	4375	.0007	.0001	0681	1523	4375
.8	.5000	.08	.0312	5000	.008	.0019	5312	5000	.0008	.0001	2207	0312	5000
.9	.5625	.09	.0351	5625	.009	.0021	9726	5625	.0009	.0001	3732	9101	5625
.A	.6250	.0A	.0390	6250	.00A	.0024	4140	6250	.000A	.0001	5258	7890	6250
.B	.6875	.0B	.0429	6875	.00B	.0026	8554	6875	.000B	.0001	6784	6679	6875
.C	.7500	.0C	.0468	7500	.00C	.0029	2968	7500	.000C	.0001	8310	5468	7500
.D	.8125	.0D	.0507	8125	.00D	.0031	7382	8125	.000D	.0001	9836	4257	8125
.E	.8750	.0E	.0546	8750	.00E	.0034	1796	8750	.000E	.0002	1362	3046	8750
.F	.9375	.0F	.0585	9375	.00F	.0036	6210	9375	.000F	.0002	2888	1835	9375
1		2			3				4				

TO CONVERT .ABC HEXADECIMAL TO DECIMAL

Find .A in position 1 .6250

Find .0B in position 2 .0429 6825

Find .00C in position 3 + .0029 2968 7500

.ABC is equal to .6708 9843 7500

TO CONVERT .13 DECIMAL TO HEXADECIMAL

1. Find .1250 next lowest to .1300
   subtract – .1250 = .2 Hex
2. Find .0039 0625 next lowest to .0050 0000
   – .0039 0625 = .01
3. Find .0009 7656 2500 .0010 9375 0000
   – .0009 7656 2500 = .004
4. Find .0001 0681 1523 4375 .0001 1718 7500 0000
   – .0001 0681 1523 4375 = .0007
   .0000 1037 5976 5625 = .2147 Hex
5. .13 Decimal is approximately equal to ↑

To convert fractions beyond the capacity of the table, use techniques below:

HEXADECIMAL TO FRACTION DECIMAL

Convert the hexadecimal fraction to its decimal equivalent using the same technique as for integer numbers. Divide the results by 16n (n is the number of fraction positions).

Example: $.8A7 = .540771_{10}$

$8A7_{16} = 2215_{10}$ .540771

$16^3 = 4096$ 4096 | 2215.000000

DECIMAL FRACTION TO HEXADECIMAL

Collect integer parts of product in the order of calculation.

Example: $.5408_{10} = .8A7_{16}$

.5408
× 16
8 ← 8. 6528
× 16
A ← 10. 4448
× 16
7 ← 7. 1168

Hexadecimal Addition and Subtraction Table

Example: 6 + 2 = 8, 8 - 2 = 6, and 8 - 6 = 2

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
2	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
3	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12
4	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
5	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14
6	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15
7	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16
8	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17
9	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18
A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19
B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A
C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Hexadecimal Multiplication Table

Example: 2 x 4 = 08, F x 2 = 1E

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
2	02	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
3	03	06	09	0C	0F	12	15	18	1B	1E	21	24	27	2A	2D
4	04	08	0C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	05	0A	0F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	06	0C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	07	0E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	08	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	09	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Appendix I. EBCDIC and ISO-8 Codes

Dec	Hex	EBCDIC-037	ISO-8
0	00	NUL	NUL
1	01	SOH	SOH
2	02	STX	STX
3	03	ETX	ETX
4	04	SEL	EOT
5	05	HT	ENQ
6	06	RNL	ACK
7	07	DEL	BEL
8	08	GE	BS
9	09	SPS	HT
10	0A	RPT	LF
11	0B	VT	VT
12	0C	FF	FF
13	0D	CR	CR
14	0E	SO	SO
15	0F	SI	SI
16	10	DLE	DLE
17	11	DC1	DC1
18	12	DC2	DC2
19	13	DC3	DC3
20	14	RES/ENP	DC4
21	15	NL	NAK
22	16	BS	SYN
23	17	POC	ETB
24	18	CAN	CAN
25	19	EM	EM
26	1A	UBS	SUB
27	1B	CU1	ESC
28	1C	IFS	IFS
29	1D	IGS	IGS
30	1E	IRS	IRS
31	1F	ITB/IUS	IUS
32	20	DS	SP
33	21	SOS	!
34	22	FS	"
35	23	WUS	#
36	24	BYP/INP	$
37	25	LF	%
38	26	ETB	&
39	27	ESC	'
40	28	SA	(
41	29	SFE	)
42	2A	SM/SW	*
43	2B	CSP	+
44	2C	MFA	,
45	2D	ENQ	-
46	2E	ACK	.
47	2F	BEL	/
48	30		0
49	31		1
50	32	SYN	2
51	33	IR	3
52	34	PP	4
53	35	TRN	5
54	36	NBS	6
55	37	EOT	7
56	38	SBS	8
57	39	IT	9
58	3A	RFF	:
59	3B	CU3	;
60	3C	DC4	<
61	3D	NAK	=
62	3E		>
63	3F	SUB	?

Dec	Hex	EBCDIC-037	ISO-8
64	40	SP	@
65	41	RSP	A
66	42	â	B
67	43	ä	C
68	44	à	D
69	45	á	E
70	46	ã	F
71	47	å	G
72	48	ç	H
73	49	ñ	I
74	4A	¢	J
75	4B	.	K
76	4C	<	L
77	4D	(	M
78	4E	+	N
79	4F	|	O
80	50	&	P
81	51	é	Q
82	52	ê	R
83	53	ë	S
84	54	è	T
85	55	í	U
86	56	î	V
87	57	ï	W
88	58	ì	X
89	59	ß	Y
90	5A	!	Z
91	5B	$	[
92	5C	*	\
93	5D	)	]
94	5E	;	^
95	5F	¬	_
96	60	-	‘
97	61	/	a
98	62	Â	b
99	63	Ä	c
100	64	À	d
101	65	Á	e
102	66	Ã	f
103	67	Å	g
104	68	Ç	h
105	69	Ñ	i
106	6A	¦	j
107	6B	,	k
108	6C	%	l
109	6D	_	m
110	6E	>	n
111	6F	?	o
112	70	ø	p
113	71	É	q
114	72	Ê	r
115	73	Ë	s
116	74	È	t
117	75	Í	u
118	76	Î	v
119	77	Ï	w
120	78	Ì	x
121	79	‘	y
122	7A	:	z
123	7B	#	{
124	7C	@	|
125	7D	'	}
126	7E	=	~
127	7F	"	¤

Dec	Hex	EBCDIC-037	ISO-8
128	80	Ø	
129	81	a	
130	82	b	BPH
131	83	c	NBH
132	84	d	IND
133	85	e	NEL
134	86	f	SSA
135	87	g	ESA
136	88	h	HTS
137	89	i	HTJ
138	8A	«	VTS
139	8B	»	PLD
140	8C	ð	PLU
141	8D	ý	RI
142	8E	þ	SS2
143	8F	±	SS3
144	90	°	DCS
145	91	j	PU1
146	92	k	PU2
147	93	l	STS
148	94	m	CCH
149	95	n	MW
150	96	o	SPA
151	97	p	EPA
152	98	q	SOS
153	99	r	
154	9A	ª	SCI
155	9B	º	CSI
156	9C	æ	ST
157	9D		OSC
158	9E	Æ	PM
159	9F	¤	APC
160	A0	µ	RSP
161	A1	~	¡
162	A2	s	¢
163	A3	t	£
164	A4	u	¤
165	A5	v	¥
166	A6	w	¦
167	A7	x	§
168	A8	y	¨
169	A9	z	©
170	AA	¡	ª
171	AB	¿	«
172	AC	Ð	¬
173	AD	Ý	SHY
174	AE	þ	®
175	AF	®	¯
176	B0	^	°
177	B1	£	±
178	B2	¥	²
179	B3	·	³
180	B4	©	´
181	B5	§	µ
182	B6	¶	¶
183	B7	¼	·
184	B8	½	¸
185	B9	¾	¹
186	BA	[	º
187	BB	]	»
188	BC	¯	¼
189	BD	¨	½
190	BE	´	¾
191	BF	×	¿

Dec	Hex	EBCDIC-037	ISO-8
192	C0	{	À
193	C1	A	Á
194	C2	B	Â
195	C3	C	Ã
196	C4	D	Ä
197	C5	E	Å
198	C6	F	Æ
199	C7	G	Ç
200	C8	H	È
201	C9	I	É
202	CA	SHY	Ê
203	CB	ô	Ë
204	CC	ö	Ì
205	CD	ò	Í
206	CE	ó	Î
207	CF	õ	Ï
208	D0	}	Ð
209	D1	J	Ñ
210	D2	K	Ò
211	D3	L	Ó
212	D4	M	Ô
213	D5	N	Õ
214	D6	O	Ö
215	D7	P	×
216	D8	Q	Ø
217	D9	R	Ù
218	DA	¹	Ú
219	DB	û	Û
220	DC	ü	Ü
221	DD	ù	Ý
222	DE	ú	Þ
223	DF	ÿ	ß
224	E0	\	à
225	E1	÷/NSP	á
226	E2	S	â
227	E3	T	ã
228	E4	U	ä
229	E5	V	å
230	E6	W	æ
231	E7	X	ç
232	E8	Y	è
233	E9	Z	é
234	EA	²	ê
235	EB	Ô	ë
236	EC	Ö	ì
237	ED	Ò	í
238	EE	Ó	î
239	EF	Õ	ï
240	F0	0	ð
241	F1	1	ñ
242	F2	2	ò
243	F3	3	ó
244	F4	4	ô
245	F5	5	õ
246	F6	6	ö
247	F7	7	÷
248	F8	8	ø
249	F9	9	ù
250	FA	³	ú
251	FB	Û	û
252	FC	Ü	ü
253	FD	Ù	ý
254	FE	Ú	þ
255	FF	EO	ÿ

© Copyright IBM Corp. 2000, 2017

Control Character Representations

ACK	Acknowledge
BEL	Bell
BS	Backspace
BYP	Bypass
CAN	Cancel
CR	Carriage Return
CSP	Control Sequence Prefix
CU1	Customer Use 1
CU3	Customer Use 3
DC1	Device Control 1
DC2	Device Control 2
DC3	Device Control 3
DC4	Device Control 4
DEL	Delete
DLE	Data Link Escape
DS	Digit Select
EM	End of Medium
ENP	Enable Presentation
ENQ	Enquiry
EO	Eight Ones
EOT	End of Transmission
ESC	Escape
ETB	End of Transmission Block
ETX	End of Text
FF	Form Feed
FS	Field Separator
GE	Graphic Escape
HT	Horizontal Tab
IFS	Interchange File Separator
IGS	Interchange Group Separator
INP	Inhibit Presentation
IR	Index Return
IRS	Interchange Record Separator
IT	Indent Tab
ITB	Intermediate Transmission Block
IUS	International Unit Separator
LF	Line Feed
MFA	Modify Field Attribute
NAK	Negative Acknowledge
NBS	Numeric Backspace
NL	New Line
NUL	Null
POC	Program-Operator Communication
PP	Presentation Position
RES	Restore
RFF	Required Form Feed
RNL	Required New Line
RPT	Repeat
SA	Set Attribute
SBS	Subscript
SEL	Select
SFE	Start Field Extended
SI	Shift In
SM	Set Mode
SO	Shift Out
SOH	Start of Heading
SOS	Start of Significance
SPS	Superscript
STX	Start of Text
SUB	Substitute
SW	Switch
SYN	Synchronous Idle
TRN	Transparent
UBS	Unit Backspace
VT	Vertical Tab
WUS	Word Underscore

Formatting Character Representations

NSP	Numeric Space
RSP	Required Space
SP	Space
SHY	Syllable Hyphen

Additional ISO-8 Control Character Representations

APC	Application Program Command
BPH	Break Permitted Here
CCH	Cancel Character
CSI	Control Sequence Introducer
DCS	Device Control String
EPA	End of Guarded Area
ESA	End of Selected Area
HTJ	Character Tabulation with Justification
HTS	Character Tabulation Set
IFS	Information Separator Four
IGS	Information Separator Three
IND	Index
IRS	Information Separator Two
MW	Message Waiting
NBH	No Break Here
NEL	Next Line
OSC	Operating System Command
PLD	Partial Line Down
PLU	Partial Line Up
PM	Privacy Message
PU1	Private Use One
PU2	Private Use Two
RI	Reverse linefeed (or index)
SCI	Single Character Introducer
SOS	Start of String
SPA	Start of Guarded Area
SSA	Start of Selected Area
SS2	Single Shift Two
SS3	Single Shift Three
ST	String Terminator
STS	Set Transmit State
US	Information Separator One
VTS	Line Tabulation Set

Note: The ISO-8 controls are from ISO 6429, and the graphics are from ISO 8859-1. The ISO-8 graphics are code page 00819, named ISO/ANSI Multilingual.

Index

Numerics

2K-IDAW control 15-28
370-XA architecture 1-32

A

A (ADD) binary instruction 7-26
abort
  transactional execution 5-89
aborted-transaction-instruction-address
  in the transaction diagnostic block 5-95
absolute address 3-4
absolute storage 3-4
access-control bits
  in segment-table entry 3-46, 3-49
access-control bits in storage key 3-9
access-exception-fetch/store-indication facility
  facility indication 4-100
access exceptions 6-47, 6-53
  priority of 6-53
  recognition of 6-47
access key 3-11
  for channel-program execution 3-11, 15-25, 15-30
  for channel-subsystem monitoring 3-11
  for CPU 3-11
access list 5-56
  See also access-list entry
  accessing capability, revocation of 5-51
  allocation and invalidation of entries in 5-49
  authorizing the use of entries in 5-49
  concepts 5-47
  designation (ALD) 5-56
  length (ALL) 5-56
  origin (ALO) 5-56
access-list-controlled protection 3-13
  exception for 6-34
access-list designation 5-56
access-list entry (ALE) 5-56
  authorization index (ALEAX) 5-57
  number
    See ALEN
  sequence exception 6-22
    as an access exception 6-47
  sequence number (ALESN)
    in ALE 5-56
    in ALET 5-54
  token
    See ALET
access-register mode 3-40
access-register translation (ART) 5-53
  as part of LOAD REAL ADDRESS, STORE REAL ADDRESS, TEST ACCESS, and TEST PROTECTION 5-59
  introduction to 5-47
  lookaside buffer
    See ALB
  sequence of table fetches 5-119
access-register-translation (ART) tables 5-55
access registers 2-6
  designation of 5-46
  functions of 5-46
  instructions for use of 5-52
  machine-check save area for 3-87
  save areas for 3-83
  store-status save area for 3-87
  validity bit for 11-21
access to storage 5-113
  See also reference
active
  device 16-16, 16-36
  subchannel 16-16, 16-36
active allegiance 15-13
active communication 15-13
activity-control field (SCSW) 16-14, 16-36
  following TEST SUBCHANNEL 14-21
AD (ADD NORMALIZED) HFP instruction 18-8
  example A-41
adaptive dictionary 7-180
ADB (ADD) BFP instruction 19-15
ADBR (ADD) BFP instruction 19-15
ADD BFP instructions 19-15
ADD binary instructions 7-26
ADD DECIMAL instruction 8-6
  example A-36
ADD DFP instructions 20-19
ADD HALFWORD IMMEDIATE instruction 7-28
ADD HALFWORD instruction 7-28
  example A-8
ADD HIGH binary instructions 7-29
ADD IMMEDIATE binary instructions 7-27
ADD IMMEDIATE HIGH binary instructions 7-29
ADD LOGICAL HIGH binary instructions 7-31
ADD LOGICAL IMMEDIATE instructions 7-30
ADD LOGICAL instructions 7-30
ADD LOGICAL WITH CARRY instructions 7-31
ADD LOGICAL WITH SIGNED IMMEDIATE HIGH binary instructions 7-33
ADD LOGICAL WITH SIGNED IMMEDIATE instructions 7-32

© Copyright IBM Corp. 2000, 2017

ADD NORMALIZED HFP instructions 18-8
example A-41
ADD UNNORMALIZED HFP instructions 18-9
example A-41
additional floating-point (AFP) registers 9-9
address 3-2
24-bit, 31-bit, and 64-bit 3-6
in branch-address generation 5-13
in operand address generation 5-12
absolute 3-4
AR-specified virtual
See AR-specified
arithmetic 3-5, 5-11
unsigned binary 7-6
backward stack-entry 5-80
base
See base address
branch
See branch address
channel-program
See channel-program address
comparison 12-1
controls for 12-1
effect on CPU state 4-2
CPU
See CPU address
data (I/O)
See data address
effective
See effective address
failing-storage
See failing-storage address
format 3-3
forward-section header 5-81
generation 5-10
for storage addressing 3-7
home virtual
See home virtual address
I/O 13-5
instruction
See instruction address
invalid 6-20
logical
See logical address
numbering of for byte locations 3-2
PER
See PER address
prefixing
See prefix
primary virtual
See primary virtual address
real 3-4
secondary virtual
See secondary virtual address
size of
controlled by addressing mode 5-10
storage 3-2
summary information 3-70
translation
See dynamic address translation, prefix
types 3-4
virtual 3-5
wraparound
See wraparound
address-limit checking (I/O)
effect of I/O-system reset on 17-15
limit mode (bits in PMCW) 15-3
address-limit-checking control (I/O) 15-27, 16-12
used for IPL 17-17
address of] 15-35
address space 3-23
AR-specified 5-46
changing of 3-23
control bits
control bit 5-76
in the PSW 4-6
use in address translation 3-40
created by DAT 3-38
number
See ASN
address-space-control element (ASCE) 3-42
effective 3-52
home 3-45
in AST entry 3-32
used for access-register translation 5-58
used for subspace groups 5-69
primary 3-42
secondary 3-44
type exception 6-22
addressing
trimodal 7-8
address-and-translation-mode identification (ATMID) 4-30
addressing exception 6-20
as an access exception 6-47, 6-53
addressing mode 5-10
bit in linkage-stack state entry 5-83
bit in the PSW 4-7
effect on address size 3-6
effect on operand-address generation 5-12
effect on sequential instruction-address generation 5-10
effect on wraparound 3-7
in branch-address generation 5-13
in examples A-7
in operand address generation 5-12
set by BRANCH AND SAVE AND SET MODE instruction 7-37

set by BRANCH AND SET MODE instruction 7-38
set by SET ADDRESSING MODE instruction 7-370
use of 5-18
ADR (ADD NORMALIZED) HFP instruction 18-8
ADTR (ADD) DFP instruction 20-19
AE (ADD NORMALIZED) HFP instruction 18-8
example A-41
AEB (ADD) BFP instruction 19-15
AEBR (ADD) BFP instruction 19-15
AER (ADD NORMALIZED) HFP instruction 18-8
AFI (ADD IMMEDIATE) instruction 7-27, 7-29
AFP (additional floating-point) registers 9-9
AFP-register data exception 6-25
AFT (ASN first table) 3-31
AFTE (ASN-first-table entry) 3-31
AFTO (ASN-first-table origin) 3-31
AFX (ASN-first-table index) 3-30
invalid bit 3-31
translation exception 6-22
AG (ADD) binary instruction 7-27
AGF (ADD) binary instruction 7-27
AGFI (ADD IMMEDIATE) instruction 7-27
AGFR (ADD) binary instruction 7-26
AGH (ADD HALFWORD) instruction 7-28
AGHI (ADD HALFWORD IMMEDIATE) instruction 7-29
AGHIK (ADD IMMEDIATE) instruction 7-27
AGR (ADD) binary instruction 7-26
AGRK (ADD) binary instruction 7-26
AGSI (ADD IMMEDIATE) instruction 7-27
AH (ADD HALFWORD) instruction 7-28
example A-8
AHHHR (ADD HIGH) binary instruction 7-29
AHI (ADD HALFWORD IMMEDIATE) instruction 7-28
AHIK (ADD IMMEDIATE) instruction 7-27
AHY (ADD HALFWORD) instruction 7-28
AIH (ADD IMMEDIATE HIGH) binary instruction 7-29
AKM (authorization key mask) 5-38
AL (ADD LOGICAL) instruction 7-30
ALB (ART-lookaside buffer) 5-64
entry
clearing of 5-66
effect of translation changes on 5-66
usable state 5-65
ALC (ADD LOGICAL WITH CARRY) instruction 7-31
ALCG (ADD LOGICAL WITH CARRY) instruction 7-31
ALCGR (ADD LOGICAL WITH CARRY) instruction 7-31
ALCR (ADD LOGICAL WITH CARRY) instruction 7-31
ALD (access-list designation) 5-56
ALE
See access-list entry
ALEAX (access-list-entry authorization index) 5-57
ALEN (access-list-entry number) 5-54
invalid bit 5-56
translation exception 6-22
as an access exception 6-47
alert (class of machine-check condition) 11-10
alert interruption condition (I/O) 16-5
alert-status bit (I/O) 16-17
ALESN (access-list-entry sequence number)
in ALE 5-56
in ALET 5-54
ALET (access-list-entry token) 5-48, 5-54
specification exception 6-22
as an access exception 6-47
ALFI (ADD LOGICAL IMMEDIATE) instruction 7-30
ALG (ADD LOGICAL) instruction 7-30
ALGF (ADD LOGICAL) instruction 7-30
ALGFI (ADD LOGICAL IMMEDIATE) instruction 7-30
ALGFR (ADD LOGICAL) instruction 7-30
ALGHSIK (ADD LOGICAL WITH SIGNED IMMEDIATE) instruction 7-32
ALGR (ADD LOGICAL) instruction 7-30
ALGRK (ADD) binary instruction 7-30
ALGSI (ADD LOGICAL WITH SIGNED IMMEDIATE) instruction 7-32
ALHHHR (ADD LOGICAL HIGH) binary instruction 7-31
ALHSIK (ADD LOGICAL WITH SIGNED IMMEDIATE) instruction 7-32
alignment 3-3
ALL (access-list length) 5-56
allegiance
active 15-13
channel-path 15-12
dedicated 15-13
effect on CLEAR SUBCHANNEL of 15-12
working 15-12
allowed interruptions 6-6
ALO (access-list origin) 5-56
ALR (ADD LOGICAL) instruction 7-30
ALRK (ADD) binary instruction 7-30
ALSI (ADD LOGICAL WITH SIGNED IMMEDIATE) instruction 7-32
ALSIH (ADD LOGICAL WITH SIGNED IMMEDIATE HIGH) instruction 7-33
ALSIHN (ADD LOGICAL WITH SIGNED IMMEDIATE HIGH) binary instruction 7-33
alter-and-display controls 12-2
alteration
storage (PER event) 4-36

ALY (ADD LOGICAL) instruction 7-30
ancillary-report bit
in channel-report word 17-27
in machine-check-interruption code 11-18
in subchannel logout 16-49
AND IMMEDIATE instructions 7-34
AND instructions 7-33
examples A-8
AP (ADD DECIMAL) instruction 8-6
example A-36
AR (ADD) binary instruction 7-26
AR-specified (access-register-specified) 3-5, 3-23
address space 3-23, 5-46
virtual address 3-5
AR-specified (access-register-specified) virtual address
effective address-space-control element for 3-52
architected register context
See CPU
architectural mode
identification 3-76, 3-85
indication of 12-2, 12-3
selection of by IML controls 12-3
selection of by manual controls 12-3
selection of by signal-processor order 4-89
architecture
compatibility 1-34
arithmetic
address
See address arithmetic
binary 7-5
examples A-2
decimal 8-3
examples A-4, A-36
floating-point 9-1
examples A-5, A-41
logical (unsigned binary) 7-6
examples A-3
ARK (ADD) binary instruction 7-26
ART
See access-register translation
ART-lookaside buffer
See ALB
ASCE
See address-space-control element
ASCE (address-space-control element)
type exception 6-22
ASCII character code
handled by architecture xxix
ASI (ADD IMMEDIATE) instruction 7-27
ASN (address-space number) 3-23
AST (ASN second table) 3-31
authorization 3-35
first table (AFT) 3-31
first-table index
See AFX
first-table origin (AFTO) 3-31
in entry-table entry 5-38
second-table (AST) 3-31
second-table address in ETE 5-38
second-table entry (ASTE)
for subspace groups 5-68
instance exception 6-23
instance number (ASTEIN), used in ASN reuse 3-26
origin, in ALE 5-57
primary (PASTE) 5-35
pseudo 3-24
sequence exception 6-23
sequence exception as an access exception 6-47
sequence number (ASTESN), in ALE 5-57
sequence number (ASTESN), in ASTE 5-58
validity exception 6-24
validity exception as an access exception 6-47
second-table index
See ASX
second-table origin (ASTO) 3-31
trace-control bit 4-15
translation 3-30
exceptions 6-56
translation-control bit 3-30, 5-29
ASN-and-LX-reuse-control bit 3-25
ASN-and-LX-reuse facility 1-7, 3-25
facility indication 4-99
assembler language A-7
instruction formats in
See instruction lists and page numbers in Appendix B
assigned storage locations 3-73
AST (ASN second table) 3-31
AST entry 5-57
See also ASN-second-table entry
ASTE
See ASN-second-table entry
ASTESN (AST-entry sequence number)
in ALE 5-57
in ASTE 5-58
ASTO (ASN-second-table origin) 3-31
ASX (ASN-second-table index) 3-30
invalid bit 3-31
use in ART 5-58
translation exception 6-24
AT
See authority table
ATL (authority-table length) 3-32
use in ART 5-58

ATMID (addressing-and-translation-mode identification) 4-30
ATO (authority-table origin) 3-32
use in ART 5-58
attached ART-table entry 5-65
attached region-table, segment-table, or page-table entry 3-63
attachment of I/O devices 13-3
attribute-set indication
CPU attribute
CPU speed 7-258
AU (ADD UNNORMALIZED) HFP instruction 18-9
example A-41
AUR (ADD UNNORMALIZED) HFP instruction 18-9
authority table (AT) 5-29
designation 3-32, 5-58
length 3-32, 5-58
origin 3-32, 5-58
authorization
ASN 3-35
index (AX) 3-35, 5-29
key mask (AKM) 5-38
mechanisms 5-27
summary of 5-32
testing of 5-74
authorization check 16-52
indicator for (in ERW) 16-52
automatic reconfiguration 1-31
auxiliary storage 3-2, 3-38
availability (characteristic of a system) 1-35
AW (ADD UNNORMALIZED) HFP instruction 18-9
AWR (ADD UNNORMALIZED) HFP instruction 18-9
AX (authorization index) 3-35, 5-29
AXBR (ADD) BFP instruction 19-15
AXR (ADD NORMALIZED) HFP instruction 18-8
AY (ADD) binary instruction 7-27

B

B field of instruction 5-11
backed-up bit (machine-check condition) 11-17
backup
processing (synchronous machine-check condition) 11-18
backward stack-entry address 5-80
backward stack-entry validity bit 5-80
BAKR (BRANCH AND STACK) instruction 10-11
examples A-10
BAL (BRANCH AND LINK) instruction 7-35
examples A-8
BALR (BRANCH AND LINK) instruction 7-35
examples A-8
BAS (BRANCH AND SAVE) instruction 7-36
example A-8
base address 5-11
register for 2-4
base-AST-entry origin (BASTEO) 5-67
base-authority state 10-8
base space 5-18
base-space bit 5-68
basic addressing mode
bit in entry-table entry 5-37
basic I/O functions 15-1
basic PROGRAM CALL 5-72, 10-93
BASR (BRANCH AND SAVE) instruction 7-36
example A-8
BASSM (BRANCH AND SAVE AND SET MODE) instruction 7-37
example A-8
BASTEO (base-AST-entry origin) 5-67
BC (BRANCH ON CONDITION) instruction 7-40
example A-12
BCR (BRANCH ON CONDITION) instruction 7-40
BCT (BRANCH ON COUNT) instruction 7-41
example A-12
BCTG (BRANCH ON COUNT) instruction 7-41
BCTGR (BRANCH ON COUNT) instruction 7-41
BCTR (BRANCH ON COUNT) instruction 7-41
example A-12
BFP
data class
testing of 19-41
BFP (binary floating point) 9-4
BFP data 19-2
conversion of 9-27
BFP facility 19-1
BFP-instruction data exception 6-25
bias for exponent 9-2, 19-2, 20-3
function of view 9-2
summary for BFP 19-2, 19-3
summary for DFP 20-3, 20-6
biased exponent 9-2, 19-2, 20-3
BIC (BRANCH INDIRECT ON CONDITION) instruction 7-39
big endian 7-288, 7-387
bimodal addressing
See addressing mode
binary
See also fixed point
arithmetic 7-5
examples A-2
negative zero 7-4
number representation 7-4
examples A-2
overflow 7-5
example A-2
See also fixed point
sign bit 7-4

binary floating point (BFP) 9-4
binary integer
conversion from floating point 18-11, 19-22, 19-25, 20-29, 20-32
conversion to floating point 18-11, 19-19, 20-24, 20-25
binary-to-decimal conversion 7-231, 25-7
example A-19
bit 3-2
numbering of within a group of bytes 3-3
bits to left of compressed data 7-175
bits to right of compressed data 7-175
block-concurrent storage references 5-126
block number
expanded storage 2-2
block of I/O data 15-24
block of storage 3-4
See also page
testing for usability of 10-167
borrow 7-390
boundary alignment 3-3
for instructions 5-3
BPP (BRANCH PREDICTION PRELOAD) instruction 7-43
BPRP (BRANCH PREDICTION RELATIVE PRELOAD) instruction 7-43
branch address 5-12
control bit 4-27
in linkage-stack state entry 5-83
in trace entry 4-22
BRANCH AND LINK instructions 7-35
examples A-8
BRANCH AND SAVE AND SET MODE instruction 7-37
examples A-8
BRANCH AND SAVE instructions 7-36
examples A-8
BRANCH AND SET AUTHORITY instruction 10-7
BRANCH AND SET MODE instruction 7-38
examples A-8
BRANCH AND STACK instruction 10-11
examples A-10
BRANCH IN SUBSPACE GROUP instruction 10-13
BRANCH INDIRECT ON CONDITION instruction 7-39
BRANCH ON CONDITION instructions 7-40
example A-12
BRANCH ON COUNT instructions 7-41
example A-12
BRANCH ON INDEX HIGH instruction 7-42
examples A-12
BRANCH ON INDEX LOW OR EQUAL instruction 7-42
examples A-13
BRANCH RELATIVE AND SAVE instruction 7-46
BRANCH RELATIVE AND SAVE LONG instruction 7-46
BRANCH RELATIVE ON CONDITION instruction 7-46
BRANCH RELATIVE ON CONDITION LONG instruction 7-46
BRANCH RELATIVE ON COUNT HIGH instruction 7-47
BRANCH RELATIVE ON COUNT instruction 7-47
BRANCH RELATIVE ON INDEX HIGH instruction 7-48
BRANCH RELATIVE ON INDEX LOW OR EQUAL instruction 7-48
branch state entry 5-81, 10-11
branch-trace-control bit 4-15
branching
branch-address generation 5-12
in a channel program 15-76
relative 5-13
to perform decision making, loop control, and subroutine linkage 5-14
using the linkage stack 5-73
BRAS (BRANCH RELATIVE AND SAVE) instruction 7-46
BRASL (BRANCH RELATIVE AND SAVE LONG) instruction 7-46
BRC (BRANCH RELATIVE ON CONDITION) instruction 7-46
BRCL (BRANCH RELATIVE ON CONDITION LONG) instruction 7-46
BRCT (BRANCH RELATIVE ON COUNT) instruction 7-47
BRCTG (BRANCH RELATIVE ON COUNT) instruction 7-47
BRCTH (BRANCH RELATIVE ON COUNT HIGH) instruction 7-47
breaking-event address
in the transaction diagnostic block 5-96
assigned storage locations for 3-80
breaking-event-address register 4-44
broadcasted-purging facility 1-27
BRXH (BRANCH RELATIVE ON INDEX HIGH) instruction 7-48
BRXHG (BRANCH RELATIVE ON INDEX HIGH) instruction 7-48
BRXLE (BRANCH RELATIVE ON INDEX LOW OR EQUAL) instruction 7-48
BRXLG (BRANCH RELATIVE ON INDEX LOW OR EQUAL) instruction 7-48
BSA (BRANCH AND SET AUTHORITY) instruction 10-7
BSG (BRANCH IN SUBSPACE GROUP) instruction 10-13

BSM (BRANCH AND SET MODE) instruction 7-38
example A-8
buffer storage (cache) 3-2
burst mode (channel-path operation) 13-3
busy
in I/O operations 13-10
in SIGNAL PROCESSOR 4-94
BX (byte index) 3-39
BXH (BRANCH ON INDEX HIGH) instruction 7-42
examples A-12
BXHG (BRANCH ON INDEX HIGH) instruction 7-42
BXLE (BRANCH ON INDEX LOW OR EQUAL) instruction 7-42
examples A-13
BXLEG (BRANCH ON INDEX LOW OR EQUAL) instruction 7-42
byte 3-2
numbering of in storage 3-2
byte index (BX) 3-39
byte-multiplex mode (channel-path operation) 13-3

C

C (COMPARE) binary instruction 7-134
cache 3-2
called-space identification 5-83
cancel-I/O facility 1-28
CANCEL SUBCHANNEL instruction 14-3
candidates
selection of 9-13
canonical
declets 20-6
canonicalization
of DFP data 20-7
capability list 5-51
carry 7-5
CBC (checking-block code) 11-2
invalid 11-2
in registers 11-9
in storage 11-7
in storage keys 11-7
near-valid 11-2
valid 11-2
CBN (compressed-data bit number) 7-173
example of 7-179
CCC (channel-control check) 16-29
CCW (channel-command word) 15-31
2K-IDAW control
used for IPL 17-17
address of 16-19
byte count in 15-33
channel-program-type control
used for IPL 17-17
check (in subchannel logout) 16-48
command codes
See commands
contents of 15-31
current 15-31
designation of storage area in 15-32
format-0 and format-1 15-31
format control 16-11, 16-12, 16-35, 16-36
used for IPL 17-17
IDA flag in 15-32
IDAW control
used for IPL 17-17
incorrect-length-suppression-mode control
used for IPL 17-17
indirect data addressing used in 13-9, 15-66
invalid format of 16-27
invalid specification of 16-25, 16-40
MIDA flag in 15-33
modification control
used for IPL 17-17
modified indirect data addressing used in 15-68
modified-CCW-IDA control
used for IPL 17-17
ORB-extension control
used for IPL 17-17
PCI flag in 15-32
prefetch control in 16-11
used for IPL 17-17
prefetching 15-62
retry of
See command retry
skip flag in 15-32
streaming-mode control
used for IPL 17-17
suspend flag in 15-32
CCW channel program 13-7
CCW-format control 15-27
CCW-type IPL (CCW-type initial program loading) 17-17
CD (COMPARE) HFP instruction 18-10
CDB (COMPARE) BFP instruction 19-17
CDBR (COMPARE) BFP instruction 19-17
CDFBR (CONVERT FROM FIXED) BFP instruction 19-19, 19-21
CDFBRA (CONVERT FROM FIXED) BFP instruction 19-19
CDFR (CONVERT FROM FIXED) HFP instruction 18-11
CDFTR (CONVERT FROM FIXED) DFP instruction 20-24
CDGBR (CONVERT FROM FIXED) BFP instruction 19-19, 19-21
CDGBRA (CONVERT FROM FIXED) BFP instruction 19-19

CDGR (CONVERT FROM FIXED) HFP instruction 18-11
CDGTR (CONVERT FROM FIXED) DFP instruction 20-24
CDGTRA (CONVERT FROM FIXED) DFP instruction 20-24
CDLFBR (CONVERT FROM LOGICAL) BFP instruction 19-21
CDLFTR (CONVERT FROM LOGICAL) DFP instruction 20-25
CDLGBR (CONVERT FROM LOGICAL) BFP instruction 19-21
CDLGTR (CONVERT FROM LOGICAL) DFP instruction 20-25
CDPT (CONVERT FROM PACKED) instruction 20-26
CDR (COMPARE) HFP instruction 18-10
examples A-42
CDS (COMPARE DOUBLE AND SWAP) instruction 7-144
examples A-46
CDSG (COMPARE DOUBLE AND SWAP) instruction 7-144
CDSS (compressed-data symbol size) 7-174
CDSTR (CONVERT FROM SIGNED PACKED) DFP instruction 20-28
CDSY (COMPARE DOUBLE AND SWAP) instruction 7-144
CDTR (COMPARE) DFP instruction 20-22
CDUTR (CONVERT FROM UNSIGNED PACKED) DFP instruction 20-28
CDZT '????' Zoned-decimal source, long DFP result 20-29, 20-36
CE (COMPARE) HFP instruction 18-10
CEB (COMPARE) BFP instruction 19-17
CEBR (COMPARE) BFP instruction 19-17
CEDTR (COMPARE BIASED EXPONENT) DFP instruction 20-23
CEFBR (CONVERT FROM FIXED) BFP instruction 19-19, 19-21
CEFBRA (CONVERT FROM FIXED) BFP instruction 19-19
CEFR (CONVERT FROM FIXED) HFP instruction 18-11
CEGBR (CONVERT FROM FIXED) BFP instruction 19-19, 19-21
CEGBRA (CONVERT FROM FIXED) BFP instruction 19-19
CEGR (CONVERT FROM FIXED) HFP instruction 18-11
CELFBR (CONVERT FROM LOGICAL) BFP instruction 19-21
CELGBR (CONVERT FROM LOGICAL) BFP instruction 19-21
central processing unit
See CPU
CER (COMPARE) HFP instruction 18-10
CEXTR (COMPARE BIASED EXPONENT) DFP instruction 20-23
CFC (COMPARE AND FORM CODEWORD) instruction 7-137
example A-53
CFDBR (CONVERT TO FIXED) BFP instruction 19-22
CFDBRA (CONVERT TO FIXED) BFP instruction 19-22
CFDR (CONVERT TO FIXED) HFP instruction 18-11
CFEBR (CONVERT TO FIXED) BFP instruction 19-22
CFEBRA (CONVERT TO FIXED) BFP instruction 19-22
CFER (CONVERT TO FIXED) HFP instruction 18-11
CFI (COMPARE IMMEDIATE) instruction 7-134
CFXBR (CONVERT TO FIXED) BFP instruction 19-22
CFXBRA (CONVERT TO FIXED) BFP instruction 19-22
CFXR (CONVERT TO FIXED) HFP instruction 18-11
CG (COMPARE) binary instruction 7-134
CGDBR (CONVERT TO FIXED) BFP instruction 19-22
CGDBRA (CONVERT TO FIXED) BFP instruction 19-22
CGDR (CONVERT TO FIXED) HFP instruction 18-11
CGDTR (CONVERT TO FIXED) DFP instruction 20-29, 20-30
CGEBR (CONVERT TO FIXED) BFP instruction 19-22
CGEBRA (CONVERT TO FIXED) BFP instruction 19-22
CGER (CONVERT TO FIXED) HFP instruction 18-11
CGF (COMPARE) binary instruction 7-134
CGFI (COMPARE IMMEDIATE) instruction 7-135
CGFR (COMPARE) binary instruction 7-134
CGFRL (COMPARE RELATIVE LONG) instruction 7-135
CGH (COMPARE HALFWORD) binary instruction 7-150
CGHI (COMPARE HALFWORD IMMEDIATE) instruction 7-150
CGHRL (COMPARE HALFWORD RELATIVE LONG) binary instruction 7-150
CGHSI (COMPARE HALFWORD IMMEDIATE) instruction 7-150
CGIB (COMPARE IMMEDIATE AND BRANCH) instruction 7-136

CGIJ (COMPARE IMMEDIATE AND BRANCH RELATIVE) instruction 7-136
CGIT (COMPARE IMMEDIATE AND TRAP) instruction 7-149
CGR (COMPARE) binary instruction 7-134
CGRB (COMPARE AND BRANCH) instruction 7-135
CGRJ (COMPARE AND BRANCH RELATIVE) instruction 7-136
CGRL (COMPARE RELATIVE LONG) binary instruction 7-135
CGRT (COMPARE AND TRAP) instruction 7-149
CGXBR (CONVERT TO FIXED) BFP instruction 19-22
CGXBRA (CONVERT TO FIXED) BFP instruction 19-22
CGXR (CONVERT TO FIXED) HFP instruction 18-11
CGXTR (CONVERT TO FIXED) DFP instruction 20-29, 20-30, 20-32
CH (COMPARE HALFWORD) instruction 7-150
  example A-14
chaining check (subchannel status) 16-30
chaining of CCWs 15-63
  command
    See command chaining of CCWs
  data
    See data chaining of CCWs
chaining of CRWs 17-26, 17-27
change bit in storage key 3-9
change recording 3-20
channel-command word
  See CCW
channel commands
  See commands (I/O)
channel-control check (subchannel status) 16-29
channel-data check (subchannel status) 16-28, 16-35, 16-44
channel path 13-3
  active allegiance for 15-13
  available for selection 15-13
  dedicated allegiance for 15-13
  effect of I/O-system reset on 17-14
  masks in SCHIB
    See LPM, LPUM, PAM, PIM, PNOM, POM
  multipath mode of 15-4, 15-23
  not operational 16-13
  parallel-I/O-interface type 13-3
  serial-I/O-interface type 13-3
  storing of status for 14-16
  type of 13-3, 13-6
  working allegiance for 15-12
channel-path identifier
  See CHPID
channel-path reset 17-13
  effect of I/O-system reset on 17-15
channel-path-reset function 15-80
  completion of 15-81
  initiation by RESET CHANNEL PATH 14-9
  reset signal issued as part of 17-12
  signaling for 15-81
channel-path-status word 14-16
channel-path timeout
  indicator for (in ERW) 16-52
channel program 15-31
  2K-IDAW control for 15-28
  branching in
    See TIC
  CCW channel program 13-7
  execution of 13-11, 15-23
    resumption of 14-11
    suspension of 13-11, 15-73
  format-2-IDAW control for 15-27
  modification control for 15-26
  serialization 5-133
  streaming-mode control for 15-25
  suspend control for 15-25
  synchronization control for 15-26
  TCW channel program 13-7
channel-program address 16-19
  field-validity flag for in IRB 16-49
  used for IPL 17-17
channel-program type
  CCW channel program 13-7
  TCW channel program 13-7
channel-program-type control 15-27
channel report 17-25
  generated as a result of RCHP 14-9
channel report pending 11-17, 17-25
  effect of I/O-system reset on 17-15
  subclass-mask bit for 11-26
channel-report word
  See CRW
channel subsystem 2-7, 13-1
  addressing used in 13-5
  damage 11-17
  effect of I/O-system reset on 17-13
  effect of power-on reset on 4-80
  isolated state of 16-52
channel-subsystem-call facility 1-31
channel-subsystem monitoring 17-1
  effect of I/O-system reset on 17-15
channel-subsystem recovery 11-5, 17-24
channel-subsystem timer 17-2
  effect of I/O-system reset on 17-15
channel-subsystem timing 17-2
channel-subsystem timing-facility bit (in PMCW) 15-4
characteristic (of HFP number) 18-1
  See also exponent

characters
represented by eight-bit code xxix
check bits 3-3, 11-2
check stop 4-3, 11-9
as signal-processor status 4-97
during manual operation 12-1
effect on CPU timer 4-63
entering of 11-13
indicator 12-3
malfunction alert for 6-13
system 11-10
checking block 11-2
checking-block code
See CBC
checkpoint 11-2
checkpoint synchronization 11-3
action 11-4
operations 11-4
CHECKSUM instruction 7-49
CHH (COMPARE HIGH) instruction 7-151
CHHR (COMPARE HIGH) instruction 7-151
CHHSI (COMPARE HALFWORD IMMEDIATE) instruction 7-150
CHI (COMPARE HALFWORD IMMEDIATE) instruction 7-150
CHLR (COMPARE HIGH) instruction 7-151
CHPID (channel-path identifier) 13-5
in PMCW 15-8
used in RESET CHANNEL PATH 14-9
CHRL (COMPARE HALFWORD RELATIVE LONG) binary instruction 7-150
CHSI (COMPARE HALFWORD IMMEDIATE) instruction 7-150
CHY (COMPARE HALFWORD) instruction 7-150
CIB (COMPARE IMMEDIATE AND BRANCH) instruction 7-136
CIH (COMPARE IMMEDIATE HIGH) instruction 7-151
CIJ (COMPARE IMMEDIATE AND BRANCH RELATIVE) instruction 7-136
CIPHER MESSAGE instruction 7-53
CIPHER MESSAGE WITH AUTHENTICATION instruction 7-78
CIPHER MESSAGE WITH CHAINING instruction 7-53
CIPHER MESSAGE WITH CIPHER FEEDBACK instruction 7-92
CIPHER MESSAGE WITH COUNTER instruction 7-107
CIPHER MESSAGE WITH OUTPUT FEEDBACK instruction 7-120
CIT (COMPARE IMMEDIATE AND TRAP) instruction 7-149
CKSM (CHECKSUM) instruction 7-49
CL (COMPARE LOGICAL) instruction 7-152
class
of BFP data 19-4
testing of 19-41
of DFP data
testing of 20-56
CLC (COMPARE LOGICAL) instruction 7-152
example A-15
CLCL (COMPARE LOGICAL LONG) instruction 7-158
example A-16
CLCLE (COMPARE LOGICAL LONG EXTENDED) instruction 7-160
CLCLU (COMPARE LOGICAL LONG UNICODE) instruction 7-163
clear function 15-14, 15-22
bit in SCSW for 16-14
completion of 15-15
initiated by CLEAR SUBCHANNEL 14-5
path management for 15-14, 15-22
pending 16-16
signaling for 15-15
subchannel modification by 15-15
clear reset 4-79
clear signal 17-12
issued as part of clear function 15-15
CLEAR SUBCHANNEL instruction 14-5
See also clear function
effect on device status of 15-16
function initiated by 15-14, 15-22
use of after RESET CHANNEL PATH 14-10
clearing operation
by clear-reset function 4-79
by load-clear key 12-4
by load-clear-list-directed key 12-4
by system-reset-clear key 12-6
by TEST BLOCK instruction 10-167
CLFDBR (CONVERT TO FIXED) BFP instruction 19-25
CLFDTR (CONVERT TO LOGICAL) DFP instruction 20-32
CLFEBR (CONVERT TO FIXED) BFP instruction 19-25
CLFHSI (COMPARE LOGICAL IMMEDIATE) instruction 7-152
CLFI (COMPARE LOGICAL IMMEDIATE) instruction 7-152
CLFIT (COMPARE LOGICAL IMMEDIATE AND TRAP) instruction 7-156
CLFT (COMPARE LOGICAL AND TRAP) instruction 7-155
CLFXBR (CONVERT TO FIXED) BFP instruction 19-25
CLG (COMPARE LOGICAL) binary instruction 7-152

CLGDBR (CONVERT TO FIXED) BFP instruction 19-25
CLGDTR (CONVERT TO LOGICAL) DFP instruction 20-32
CLGEBR (CONVERT TO FIXED) BFP instruction 19-25
CLGF (COMPARE LOGICAL) binary instruction 7-152
CLGFI (COMPARE LOGICAL IMMEDIATE) instruction 7-152
CLGFR (COMPARE LOGICAL) binary instruction 7-152
CLGFRL (COMPARE LOGICAL RELATIVE LONG) binary instruction 7-153
CLGHRL (COMPARE LOGICAL RELATIVE LONG) binary instruction 7-153
CLGHSI (COMPARE LOGICAL IMMEDIATE) instruction 7-152
CLGIB (COMPARE LOGICAL IMMEDIATE AND BRANCH) instruction 7-154
CLGIJ (COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE) instruction 7-154
CLGIT (COMPARE LOGICAL IMMEDIATE AND TRAP) instruction 7-156
CLGR (COMPARE LOGICAL) binary instruction 7-152
CLGRB (COMPARE LOGICAL AND BRANCH) instruction 7-154
CLGRJ (COMPARE LOGICAL AND BRANCH RELATIVE) instruction 7-154
CLGRL (COMPARE LOGICAL RELATIVE LONG) binary instruction 7-153
CLGRT (COMPARE LOGICAL AND TRAP) instruction 7-155
CLGT (COMPARE LOGICAL AND TRAP) instruction 7-155
CLGXBR (CONVERT TO FIXED) BFP instruction 19-25
CLGXTR (CONVERT TO LOGICAL) DFP instruction 20-32
CLHH (COMPARE LOGICAL HIGH) instruction 7-157
CLHHR (COMPARE LOGICAL HIGH) instruction 7-157
CLHHSI (COMPARE LOGICAL IMMEDIATE) instruction 7-152
CLHLR (COMPARE LOGICAL HIGH) instruction 7-157
CLHRL (COMPARE LOGICAL RELATIVE LONG) binary instruction 7-153
CLI (COMPARE LOGICAL) instruction 7-152
  example A-15
CLIB (COMPARE LOGICAL IMMEDIATE AND BRANCH) instruction 7-154
CLIH (COMPARE LOGICAL IMMEDIATE HIGH) instruction 7-158
CLIJ (COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE) instruction 7-154
CLIY (COMPARE LOGICAL) instruction 7-152
CLM (COMPARE LOGICAL CHARACTERS UNDER MASK) instruction 7-157
  example A-16
CLMH (COMPARE LOGICAL CHARACTERS UNDER MASK) instruction 7-157
CLMY (COMPARE LOGICAL CHARACTERS UNDER MASK) instruction 7-157
clock
  See TOD clock
clock comparator 4-59
  external interruption 6-12
  machine-check save area for 3-86
  save areas for 3-82
  store-status save area for 3-86
  validity bit for 11-22
clock unit 4-46, 7-382
CLR (COMPARE LOGICAL) instruction 7-152
  example A-16
CLRB (COMPARE LOGICAL AND BRANCH) instruction 7-154
CLRJ (COMPARE LOGICAL AND BRANCH RELATIVE) instruction 7-154
CLRL (COMPARE LOGICAL RELATIVE LONG) binary instruction 7-153
CLRT (COMPARE LOGICAL AND TRAP) instruction 7-155
CLST (COMPARE LOGICAL STRING) instruction 7-166
  examples A-18
CLY (COMPARE LOGICAL) instruction 7-152
CMPSC (COMPRESSION CALL) instruction 7-170
CMPSC-enhancement facility 1-7
  facility indication CMPSC enhancement facility facility indication] 4-100
code
  ASCII
    handled by architecture xxix
  checking-block
    See CBC
  command (in CCW)
    See command code in CCW
  condition
    See condition code
  data-exception (DXC) 6-17, 20-15
  decimal digit and sign 8-2
  deferred condition (I/O) 16-9
  EBCDIC
    handled by architecture xxix
    table for I-2

eight-bit
handled by architecture xxix
error-recovery (I/O) 17-27
external-damage 11-23
validity bit for 11-21
I/O-interruption subclass 15-2
instruction-length
See ILC
interruption
See interruption code
linkage-stack-entry type 5-79
monitor
See monitor code
operation 5-3
PER
See PER code
reporting-source (I/O) 17-27
storage-access (in subchannel logout) 16-50
vector-exception (VXC) 6-20
codeword (for sorting operations) 7-137
example A-54
cohort 20-2
combination field
in DFP data format 20-4
command chaining of CCWs 15-63
effect of status modifier on 15-63
flag in CCW for 15-32
overview of 13-11
command chaining of TCWs
overview of 13-11
command code in CCW 15-57
See also commands
See also common I/O-device commands
invalid 16-25
command codes
See command code in CCW
command mode 13-9
command retry 15-77
effect on PCI of 15-65
commands (I/O) 15-57
See also common I/O-device commands
transfer in channel 15-77
commit
transactional execution 5-90
common I/O-device commands
publication referenced xxix
common-region bit 3-47
common-segment bit 3-50
common-rounding-point view 9-13
COMPARE AND BRANCH instructions 7-135
COMPARE AND BRANCH RELATIVE instructions 7-135
COMPARE AND FORM CODEWORD instruction 7-137
example A-53
COMPARE AND REPLACE DAT TABLE ENTRY instruction 10-17
COMPARE AND SIGNAL BFP instructions 19-18
COMPARE AND SIGNAL DFP instructions 20-23
COMPARE AND SWAP AND PURGE instruction 10-21
compare-and-swap-and-store facility 1-7
facility indication 4-99
compare-and-swap-and-store facility 2 1-8
facility indication 4-99
COMPARE AND SWAP AND STORE instruction 7-146
COMPARE AND SWAP instruction 7-144
examples A-46
compare-and-trap-instruction data exception 6-25
COMPARE AND TRAP instructions 7-149
COMPARE BFP instructions 19-17
COMPARE BIASED EXPONENT DFP instructions 20-23
COMPARE binary instructions 7-134
COMPARE DECIMAL instruction 8-7
example A-36
COMPARE DFP instructions 20-22
COMPARE DOUBLE AND SWAP instruction 7-144
examples A-46
COMPARE HALFWORD IMMEDIATE instructions 7-150
COMPARE HALFWORD instruction 7-150
example A-14
COMPARE HFP instructions 18-10
examples A-42
COMPARE HIGH instructions 7-151
COMPARE IMMEDIATE AND BRANCH instructions 7-136
COMPARE IMMEDIATE AND BRANCH RELATIVE instructions 7-136
COMPARE IMMEDIATE AND TRAP instructions 7-149
COMPARE IMMEDIATE binary instructions 7-134
COMPARE IMMEDIATE HIGH instruction 7-151
COMPARE LOGICAL AND BRANCH instructions 7-154
COMPARE LOGICAL AND BRANCH RELATIVE instructions 7-154
COMPARE LOGICAL AND TRAP instructions 7-155
COMPARE LOGICAL CHARACTERS UNDER MASK instruction 7-157
example A-16
COMPARE LOGICAL HIGH instruction 7-157
COMPARE LOGICAL IMMEDIATE AND BRANCH instructions 7-154
COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE instructions 7-154

COMPARE LOGICAL IMMEDIATE AND TRAP instructions 7-156
COMPARE LOGICAL IMMEDIATE HIGH instruction 7-158
COMPARE LOGICAL IMMEDIATE instructions 7-152
COMPARE LOGICAL instructions 7-152
examples A-15
COMPARE LOGICAL LONG EXTENDED instruction 7-160
COMPARE LOGICAL LONG instruction 7-158
example A-16
COMPARE LOGICAL LONG UNICODE instruction 7-163
COMPARE LOGICAL STRING instruction 7-166
examples A-18
COMPARE RELATIVE LONG instructions 7-135
COMPARE UNTIL SUBSTRING EQUAL instruction 7-167
comparison
address
See address comparison
decimal 8-7
example A-36
hexadecimal-floating-point
examples A-42
logical 7-7
examples A-15
of IEEE data 9-24
signed-binary 7-7
TOD-clock 4-59
compatibility 1-34
among systems implementing different architectures 1-34
among systems implementing same architecture 1-34
control-program 1-34
problem-state 1-35
completion of I/O functions
by channel-path-reset function 15-81
by clear function 15-15
by halt function 15-17
during initiation 15-78
completion of instruction execution 5-22
completion of unit of operation 5-24
compressed-data bit number (CBN) 7-173
example of 7-179
compressed-data symbol size (CDSS) 7-174
COMPRESSION CALL instruction 7-170
restriction related to
on leftmost bits in symbol-translation-table entry 7-176
on length of chain of compression-dictionary entries 7-178
on length of chain of expansion-dictionary entries 7-178
on length of character symbol during compression 7-178
on length of character symbol during expansion 7-178
on number of children 7-178, 7-180
on offset to symbol-translation table 7-174
on symbol translation with format-1 sibling descriptors 7-174
compression dictionary
size of with symbol translation 7-174
compression facility
publication referenced xxix, xxx
COMPUTE INTERMEDIATE MESSAGE DIGEST instruction 7-188
COMPUTE LAST MESSAGE DIGEST instruction 7-201
COMPUTE MESSAGE AUTHENTICATION CODE instruction 7-219
conceptual sequence 5-113
as related to storage-operand accesses 5-129
conclusion of I/O operations 13-10, 16-1
during initiation 15-78
conclusion of instruction execution 5-22
concurrency of access for storage references 5-126
concurrent sense
in ECW 16-56
indicator for (in ERW) 16-52
concurrent-sense count (in ERW) 16-52
concurrent-sense facility 17-24
condition code 4-6
deferred 16-9
for IEEE instructions 9-24
in the PSW 4-6
inserted by INSERT PROGRAM MASK 7-264
preserved by BRANCH AND LINK 5-15, 7-36
set by SET PROGRAM MASK 7-371
summary C-1
tested by BRANCH ON CONDITION instruction 7-40
used for decision making 5-14
validity bit for 11-21
conditional emergency signal (external interruption)
signal-processor order 4-90
conditional-SSKE facility 1-8
and invalid CBC in storage key 11-8
conditional-swapping instructions
See COMPARE AND SWAP instruction, COMPARE DOUBLE AND SWAP instruction
conditional-SSKE facility
facility indication 4-99
conditions for interruption
See interruption conditions

configuration 2-1
of storage 3-4
configuration-alert facility (I/O) 17-24
configuration identification (in STORE CPU ID result) 10-136, 10-137
configuration-topology facility 1-8
facility indication 4-99
configuration-z/Architecture-architectural-mode facility 1-8
conflict
transactional execution 5-90
conflict token
in the transaction diagnostic block 5-95
connective
See logical connective
consistency (storage operand) 5-125
examples A-49, A-51
console device 12-1
console integration 1-31
constrained transaction
transactional execution 5-90
constrained transactional-execution facility
facility indication 4-100
constrained-transactional-execution facility 1-8
constrained transaction-execution mode
transactional execution 5-90
contraction 4-84
control 4-1
instructions 10-1
manual
See manual operation
control-program compatibility 1-34
control register 2-6, 4-8
save areas for 3-83
validity bit 11-21
control-register assignment 4-10
(CRx.y indicates control register x, bit position y)
CR0.15:
See (SA23-2260)
CR0.43:
instruction-fetch-suppression-enablement-control bit 3-42
CR0.33:
SSM-suppression-control bit 6-38, 10-133
CR0.34:
TOD-clock-sync-control bit 4-48, 4-53
CR0.35:
low-address-protection-control bit 3-14
CR0.36:
extraction-authority-control bit 5-28
CR0.37:
secondary-space-control bit 3-41, 5-29
CR0.38:
fetch-protection-override-control bit 3-13
CR0.39:
storage-protection-override-control bit 3-12
CR0.40:
enhanced-DAT-enablement-control bit 3-41
CR0.44:
ASN-and-LX-reuse-control bit 3-25
CR0.45:
AFP-register-control bit 9-11
CR0.48:
malfunction-alert subclass-mask bit 6-13
CR0.49:
emergency-signal subclass-mask bit 6-13
CR0.50:
external-call subclass-mask bit 6-13
CR0.52:
clock-comparator subclass-mask bit 6-12
CR0.53:
CPU-timer subclass-mask bit 6-12
CR0.54:
service-signal subclass-mask bit 6-14
CR0.57:
interrupt-key subclass-mask bit 6-13
CR0.58:
See (SA23-2260)
CR0.59:
timing-alert subclass-mask bit 6-14
CR0.61:
crypto-control bit 6-24
CR0.8:
transactional-execution control 5-92
CR0.9:
transactional-execution program-interruption-filtering override 5-92
CR1:
primary address-space-control element (PASCE) 3-42
CR1.0-51:
primary real-space token origin (PRSTKO) 3-44
primary region-table origin (PRTO) 3-42
primary segment-table origin (PSTO) 3-42
CR1.54:
primary subspace-group-control bit 3-43
CR1.55:
primary private-space-control bit 3-43
CR1.56:
primary storage-alteration-event-control bit 3-43
CR1.57:
primary space-switch-event-control bit 3-43, 6-37
CR1.58:
primary real-space-control bit 3-43
CR1.60-61:

primary designation-type-control bits 3-44
CR1.62-63:
primary region-table length (PRTL) 3-44
primary segment-table length (PSTL) 3-44
CR2.33-57:
dispatchable-unit-control-table origin (DUCTO) 5-53
CR2.61:
transaction-diagnostic-scope control 5-92
CR2.62-63:
transaction-diagnostic control 5-92
CR3.0-31:
secondary ASTEIN (SASTEIN) 3-26
CR3.32-47:
PSW-key mask (PKM) 5-28
CR3.48-63:
secondary ASN (SASN) 3-24
CR4.0-31:
primary ASTEIN (PASTEIN) 3-26
CR4.32-47:
authorization index (AX) 3-35, 5-29
CR4.48-63:
primary ASN (PASN) 3-24
CR5.33-57:
primary-AST-entry origin (PASTEO) 5-35, 5-54
CR6.32-39:
I/O-interruption subclass mask 6-15
CR7:
secondary address-space-control element (SASCE) 3-44
CR7.0-51:
secondary real-space token origin (SRSTKO) 3-44
secondary region-table origin (SRTO) 3-44
secondary segment-table origin (SSTO) 3-44
CR7.54:
secondary subspace-group-control bit 3-44
CR7.55:
secondary private-space-control bit 3-44
CR7.56:
secondary storage-alteration-event-control bit 3-44
CR7.58:
secondary real-space-control bit 3-44
CR7.60-61:
secondary designation-type-control bits 3-44
CR7.62-63:
secondary region-table length (SRTL) 3-44
secondary segment-table length (SSTL) 3-44
CR8.16-31:
enhanced-monitor-mask bits 6-30, 7-288
CR8.32-47:
extended authorization index (EAX) 5-54
CR8.48-63:
monitor-mask bits 6-30, 7-288
CR9.32:
PER successful-branching-event-mask bit 4-27
CR9.33:
PER instruction-fetching-event-mask bit 4-27
CR9.34:
PER storage-alteration-event-mask bit 4-27
CR9.36:
PER store-using-real-address-event-mask bit 4-27
CR9.37:
PER zero-address-detection-event-mask bit 4-27
CR9.38:
PER transaction-end-event-mask bit 4-27
CR9.39:
PER instruction-fetching-nullification-event-mask bit 4-27
CR9.40:
PER branch-address-control bit 4-27
CR9.41:
PER event-suppression-control bit 4-27
CR9.42:
PER storage-alteration-space-control bit 4-28
CR10.0-63:
PER starting address 4-27
CR11.0-63:
PER ending address 4-27
CR12.0:
branch-trace-control bit 4-15
CR12.1:
mode-trace-control bit 4-15
CR12.2-61:
trace-entry address 4-15
CR12.62:
ASN-trace-control bit 4-15
CR12.63:
explicit-trace-control bit 4-15
CR13:
home address-space-control element (HASCE) 3-45
CR13.0-51:
home real-space token origin (HRSTKO) 3-45
home region-table origin (HRTO) 3-45
home segment-table origin (HSTO) 3-45
CR13.55:
home private-space-control bit 3-45
CR13.56:
home storage-alteration-event-control bit 3-45
CR13.57:
home space-switch-event-control bit 3-45, 6-37

CR13.58:
home real-space-control bit 3-45
CR13.60-61:
home designation-type-control bits 3-45
CR13.62-63:
home region-table length (HRTL) 3-45
home segment-table length (HSTL) 3-45
CR14.35:
channel-report-pending subclass-mask bit 11-26
CR14.36:
recovery subclass-mask bit 11-26
CR14.37:
degradation subclass-mask bit 11-27
CR14.38:
external-damage subclass-mask bit 11-27
CR14.39:
warning subclass-mask bit 11-27
CR14.42:
TOD-clock-control-override control 4-48
CR14.44:
ASN-translation-control bit 3-30, 5-29
CR14.45-63:
ASN-first-table origin (AFTO) 3-31
CR15.0-60:
linkage-stack-entry address 5-78
controlregisters
machine-check save area for 3-87
store-status save area for 3-87
control unit 2-8, 13-4
effect of I/O-system reset on 17-14
sharing of 13-4
type of 15-14
control-unit-defer time (I/O) 17-9
control unit defer time interval (in measurement block) 17-6
control-unit-queuing measurement (I/O) 17-9
control-unit-queuing-time interval (in measurement block) 17-5
controlled-ASN bit 3-25
conversion
between HFP and BFP data 9-27
binary-to-decimal 7-231, 25-7
example A-19
decimal-to-binary 7-230, 25-5
example A-18
hexadecimal-floating-point-number
basic example A-6
examples with instructions A-44
of floating-point format 9-23, 19-5
CONVERT BFP TO HFP floating-point-support instructions 9-27
CONVERT FROM FIXED BFP instructions 19-19, 19-22, 19-33
CONVERT FROM FIXED DFP instructions 20-24
CONVERT FROM FIXED HFP instructions 18-11
CONVERT FROM LOGICAL DFP instructions 20-25
CONVERT FROM SIGNED PACKED DFP instructions 20-28
CONVERT FROM UNSIGNED PACKED DFP instructions 20-28
CONVERT FROM ZONED DFP instruction 20-26, 20-29, 20-33, 20-36
CONVERT HFP TO BFP floating-point-support instructions 9-28
CONVERT TO BINARY instruction
example A-18
CONVERT TO BINARY instructions 7-230
CONVERT TO DECIMAL instruction
example A-19
CONVERT TO DECIMAL instructions 7-231
CONVERT TO FIXED BFP instructions 19-22, 19-25
CONVERT TO FIXED DFP instructions 20-29
CONVERT TO FIXED HFP instructions 18-11
CONVERT TO LOGICAL DFP instructions 20-32
CONVERT TO SIGNED PACKED DFP instructions 20-35
CONVERT TO UNSIGNED PACKED DFP instructions 20-35
CONVERT UNICODE TO UTF-8 instruction 7-234
CONVERT UTF-16 TO UTF-32 instruction 7-231
CONVERT UTF-16 TO UTF-8 instruction 7-234
CONVERT UTF-32 TO UTF-16 instruction 7-238
CONVERT UTF-32 TO UTF-8 instruction 7-241
CONVERT UTF-8 TO UNICODE instruction 7-244
CONVERT UTF-8 TO UTF-16 instruction 7-244
CONVERT UTF-8 TO UTF-32 instruction 7-248
Coordinated Universal Time (UTC) used in TOD epoch 4-51
COPY ACCESS instruction 7-252
COPY SIGN floating-point-support instruction 9-30
core
identification (ID) 2-3, 4-83, 4-84, 12-3, 12-7
multithreading 2-3
count field
in CCW 15-33
invalid 16-25
in SCSW 16-31, 16-44
counter updating (example) A-47
counting operations 7-41
coupling facility 1-32
CP (COMPARE DECIMAL) instruction 8-7
example A-36
CPA
See channel-program address
CPDT (CONVERT TO PACKED) instruction 20-33

CPSDR (COPY SIGN) floating-point-support instruction 9-30
CPU (central processing unit) 2-2
address 4-83, 4-84
assigned storage locations for 3-74, 3-84
when stored during external interruptions 6-11
architected register context 2-3, 4-92
checkpoint 11-2
configuration identification 10-136, 10-137
effect of power-on reset on 4-80
environment 10-136
format indication 10-136, 10-137
hangup due to string of interruptions 4-3
machine-type number 10-136, 10-137
registers 2-3
save areas for 3-82
reset 4-77
signal-processor order 4-85
retry 11-2
serialization 5-130
effects of transactional execution 5-130
signaling 4-84
state 4-2
check-stop 4-3
load 4-3
no effect on TOD clock 4-47
operating 4-3
stopped 4-2
types 2-3
primary 2-3, 4-81, 12-7
secondary 2-3
CPU-measurement-counter facility
facility indication 4-100
CPU-measurement-sampling facility
facility indication 4-100
CPU state 4-2
CPU timer 4-62
external interruption 6-12
machine-check save area for 3-86
save areas for 3-82
store-status save area for 3-86
validity bit for 11-22
CPUs-per-core
operator indicator 12-3
CPXT (CONVERT TO PACKED) instruction 20-33
CPYA (COPY ACCESS) instruction 7-252
CR
See control register
CR (COMPARE) binary instruction 7-134
CRB (COMPARE AND BRANCH) instruction 7-135
CRDTE (COMPARE AND REPLACE DAT TABLE ENTRY) instruction 10-17
CRJ (COMPARE AND BRANCH RELATIVE) instruction 7-135
CRL (COMPARE RELATIVE LONG) binary instruction 7-135
CRT (COMPARE AND TRAP) instruction 7-149
CRW (channel-report word)
chaining of 17-26, 17-27
error-recovery code (ERC) in 17-27
overflow in 17-27
reporting-source code (RSC) in 17-27
reporting-source ID (RSID) in 17-28
solicited 17-26
storing of 14-17
crypto-operation exception 6-24
cryptographic facility 1-31, 2-7
CS (COMPARE AND SWAP) instruction 7-144
examples A-46
CSCH (CLEAR SUBCHANNEL) instruction 14-5
CSDTR (CONVERT TO SIGNED PACKED) DFP instruction 20-35
CSG (COMPARE AND SWAP) instruction 7-144
CSP (COMPARE AND SWAP AND PURGE) instruction 10-21
CSPG (COMPARE AND SWAP AND PURGE) instruction 10-21
CSS priority 15-29
CSST (COMPARE AND SWAP AND STORE) instruction 7-146
CSXTR (CONVERT TO SIGNED PACKED) DFP instruction 20-35
CSY (COMPARE AND SWAP) instruction 7-144
CTN configuration change (machine-check condition) 11-24
CU12 (CONVERT UTF-8 TO UTF-16) instruction 7-244
CU14 (CONVERT UTF-8 TO UTF-32) instruction 7-248
CU21 (CONVERT UTF-16 TO UTF-8) instruction 7-234
CU24 (CONVERT UTF-16 TO UTF-32) instruction 7-231
CU41 (CONVERT UTF-32 TO UTF-8) instruction 7-241
CU42 (CONVERT UTF-32 TO UTF-16) instruction 7-238
CUDTR (CONVERT TO UNSIGNED PACKED) DFP instruction 20-35
current CCW 15-31
See also CCW
current PSW 4-4, 5-14
See also PSW
stored during interruption 6-2
CUSE (COMPARE UNTIL SUBSTRING EQUAL) instruction 7-167
CUTFU (CONVERT UTF-8 TO UNICODE) instruction 7-244

CUUTF (CONVERT UNICODE TO UTF-8) instruction 7-234
CUXTR (CONVERT TO UNSIGNED PACKED) DFP instruction 20-35
CVB (CONVERT TO BINARY) instruction 7-230
  example A-18
CVBG (CONVERT TO BINARY) instruction 7-230
CVBY (CONVERT TO BINARY) instruction 7-230
CVD (CONVERT TO DECIMAL) instruction 7-231
  example A-19
CVDG (CONVERT TO DECIMAL) instruction 7-231
CVDY (CONVERT TO DECIMAL) instruction 7-231
CXBR (COMPARE) BFP instruction 19-17
CXFBR (CONVERT FROM FIXED) BFP instruction 19-19, 19-21
CXFBRA (CONVERT FROM FIXED) BFP instruction 19-19
CXFR (CONVERT FROM FIXED) HFP instruction 18-11
CXFTR (CONVERT FROM FIXED) DFP instruction 20-24
CXGBR (CONVERT FROM FIXED) BFP instruction 19-19, 19-21
CXGBRA (CONVERT FROM FIXED) BFP instruction 19-19
CXGR (CONVERT FROM FIXED) HFP instruction 18-11
CXGTR (CONVERT FROM FIXED) DFP instruction 20-24
CXGTRA (CONVERT FROM FIXED) DFP instruction 20-24
CXLFBR (CONVERT FROM LOGICAL) BFP instruction 19-21
CXLFTR (CONVERT FROM LOGICAL) DFP instruction 20-25
CXLGBR (CONVERT FROM LOGICAL) BFP instruction 19-21
CXLGTR (CONVERT FROM LOGICAL) DFP instruction 20-25
CXPT (CONVERT FROM PACKED) instruction 20-26
CXR (COMPARE) HFP instruction 18-10
CXSTR (CONVERT FROM SIGNED PACKED) DFP instruction 20-28
CXTR (COMPARE) DFP instruction 20-22
CXUTR (CONVERT FROM UNSIGNED PACKED) DFP instruction 20-28
CY (COMPARE) binary instruction 7-134

D

D (DIVIDE) binary instruction 7-252
  example A-19
D field of instruction 5-11
damage
  channel-subsystem 11-17
  code (external) 11-23
    validity bit for 11-21
  external 11-16
    subclass-mask bit for 11-27
  instruction-processing 11-15
  processing 11-19
  service-processor 11-17
  system 11-15
  timing-facility 11-16
DAT
  leaf-table entry 3-42, 3-53, 3-54, 3-65
  See dynamic address translation
  leaf-table entry 3-46
DAT-enhancement facility
  facility indication 4-98
DAT-enhancement facility 1 1-8
DAT-enhancement facility 2 1-8
DAT mode (bit in PSW)
  use in address translation 3-40
DAT mode (bit in the PSW) 4-6
DAT protection 3-13
  bit for in page-table entry 3-51
  bit for in region-table entry 3-46
  bit for in segment-table entry 3-50
DAT-table format error 6-46
data
  blocking of (I/O) 15-24
  format for
    binary-floating-point instructions 19-2
    decimal instructions 8-1
    decimal-floating-point instructions 20-3
    general instructions 7-3
    hexadecimal-floating-point instructions 18-2
  indirect addressing of (I/O) 13-9, 15-66
  measurement (I/O)
    See measurement data
  modified indirect addressing of (I/O) 15-68
  prefetching of for I/O operation 15-59
  transport indirect addressing of (I/O) 15-70
data address (I/O)
  invalid 16-26
  invalid specification of 16-26
data chaining of CCWs 15-61
  flag in CCW for 15-32
  overview of 13-11
data check
  measurement-block 16-48
data exception 6-25
  AFP-register 6-25
  BFP-instruction 6-25
  compare-and-trap-instruction 6-25
  DFP-instruction 6-25

general-operand 6-25, 8-5
IEEE-exception 6-25, 9-18
priority of program interruptions for 6-17
simulated IEEE exception 6-25
data exception code
in the transaction diagnostic block 5-96
data-exception code (DXC) 6-17, 20-15
data signaling and quiet NaNs
classes of BFP data 19-4
classes of DFP data 20-7
data streaming (I/O) 13-4
effect of CCW count on 15-63
DCTI (device-connect-time interval)
in ESW 16-54
in extended-measurement word 16-57
in measurement block 17-4
DCW (device-command word)
contents of 15-38
current 15-38
DD (DIVIDE) HFP instruction 18-12
DDB (DIVIDE) BFP instruction 19-27
DDBR (DIVIDE) BFP instruction 19-27
DDR (DIVIDE) HFP instruction 18-12
DDTR (DIVIDE) DFP instruction 20-37
DDTRA (DIVIDE) DFP instruction 20-37
DE (DIVIDE) HFP instruction 18-12
DEB (DIVIDE) BFP instruction 19-27
DEBR (DIVIDE) BFP instruction 19-27
decimal
arithmetic 8-3
comparison 8-7
digit codes 8-2
divide exception 6-26
instructions 8-1
examples A-36
number representation 8-1
examples A-4
operand overlap 8-3
overflow
exception 6-26
mask in PSW 4-7
sign codes 8-2
decimal (DFP)
data formats 20-3
decimal-floating-point
packed-conversion facility 1-9
zoned-conversion facility 1-9
decimal floating point (DFP) 9-4
See also DFP
decimal-floating-point (DFP)
facility 20-1
decimal-floating-point (DFP) facility
facility indication 4-99
decimal-floating-point facility 1-9
decimal floating point packed conversion facility
facility indication 4-100
decimal-floating-point packed-conversion facility 1-9
decimal-floating-point-rounding facility 1-9
decimal-floating-point zoned-conversion facility 1-9
facility indication 4-100
decimal-to-binary conversion 7-230, 25-5
example A-18
decimal-floating-point (DFP)
facility 1-9
rounding facility 1-9
dedicated allegiance 15-13
default QNaN 9-3, 19-4, 20-7
deferred condition code 16-9
degradation (machine-check condition) 11-16
subclass-mask bit for 11-27
degradation, storage (machine-check condition) 11-19
delay in storing 5-123
delayed access exception (machine-check condition) 11-17
deletion of malfunctioning unit 11-5
denormalization rounding 9-12
denormalized value 9-12
DER (DIVIDE) HFP instruction 18-12
examples A-42
derived quantum 20-3
designation
authority-table 3-32
entry-table 5-36
home real-space (HRSD) 3-45
home region-table 3-45
home segment-table 3-45
linkage-first-table 5-35
linkage-table 5-35
primary real-space 3-42
primary region-table 3-42
primary segment-table 3-42
real-space 3-42
in AST entry 3-32
used for access-register translation 5-58
used for subspace groups 5-69
region-table 3-42
in AST entry 3-32
used for access-register translation 5-58
used for subspace groups 5-69
secondary real-space 3-44
secondary region-table 3-44
secondary segment-table 3-44
segment-table 3-42
in AST entry 3-32
used for access-register translation 5-58
used for subspace groups 5-69
designation (origin and length)

access-list 5-55
designation-type-control bits
home 3-45
primary 3-44
secondary 3-44
destructive overlap 5-128, 7-291, 7-296, 7-299
in the access-register mode 5-116
device 2-8, 13-5
console 12-1
effect of I/O-system reset on 17-14
device-active bit 16-16, 16-36
device-active-only measurement (I/O) 17-9
device-active-only-time interval (in measurement block) 17-6
device address 13-6
device-busy time (in extended measurement word) 16-58
device-busy time (in measurement block) 17-6
device-command word
See DCW
device-connect-time interval
See DCTI
device-connect-time measurement 17-10
effect of suspension on 15-75
enable 15-4
device-disconnect-time interval (in measurement block) 17-5
device identifier 13-6
device number 13-6
assignment of 13-6
in PMCW 15-5
device-number valid (bit in PMCW) 15-4
device status 16-24
field-validity flag for (in subchannel logout) 16-29, 16-49
with inappropriate bit combination 16-50
device status check 16-50
DFP
canonical data 20-6
canonical declets 20-6
classes of DFP data 20-6
data signaling and quiet NaNs 20-7
infinities 20-7
normal numbers 20-7
subnormal numbers 20-6
zero 20-6
cohort 20-2
data class
testing of 20-56
data formats 20-3
combination field 20-4
encoded trailing significand field 20-5
values of finite numbers 20-5
derived quantum 20-3
format conversion 20-7
formatting instructions 20-8
signed-packed-decimal format 20-28, 20-35
unsigned-packed-decimal format 20-28, 20-35
of DFP data
canonicalization 20-7
preferred quantum 20-2, 20-10
quantum 20-2
scaled preferred quantum 20-3
significand 20-5
significant digits 20-5
special quantum-handling operations 20-3
DFP (decimal floating point) 9-4
DFP-instruction data exception 6-25
DIAGNOSE instruction 10-23
dictionary
adaptive 7-180
DIDBR (DIVIDE TO INTEGER) BFP instruction 19-28
DIEBR (DIVIDE TO INTEGER) BFP instruction 19-28
digit codes (decimal) 8-2
digit selector (in EDIT) 8-8
direct-access storage 3-2
disabling for interruptions 6-6
disallowed interruptions 6-6
dispatchable unit (DU) 5-48
access-list designation (DUALD) 5-55
control table (DUCT) 5-55
origin (DUCTO) 5-53
when subspace-group facility installed 5-66
when trap facility installed 10-175
displacement (in relative addressing) 5-11
display (manual controls) 12-2
distinct-operands facility 1-9
facility indication 4-100
DIVIDE BFP instructions 19-27
DIVIDE binary instructions 7-252
example A-19
DIVIDE DECIMAL instruction 8-7
example A-36
DIVIDE DFP instructions 20-37
divide exception
decimal 6-26
fixed-point 6-27
HFP 6-28
DIVIDE HFP instructions 18-12
examples A-42
DIVIDE LOGICAL instructions 7-253
DIVIDE SINGLE binary instructions 7-254
DIVIDE TO INTEGER BFP instructions 19-28
remainder result of 19-5
divisible instruction execution 5-115

DL (DIVIDE LOGICAL) instruction 7-253
DLG (DIVIDE LOGICAL) binary instruction 7-253
DLGR (DIVIDE LOGICAL) binary instruction 7-253
DLR (DIVIDE LOGICAL) instruction 7-253
doubleword 3-3
doubleword-concurrent storage references 5-126
DP (DIVIDE DECIMAL) instruction 8-7
  example A-36
DR (DIVIDE) binary instruction 7-252
DSG (DIVIDE SINGLE) binary instruction 7-254
DSGF (DIVIDE SINGLE) binary instruction 7-254
DSGFR (DIVIDE SINGLE) binary instruction 7-254
DSGR (DIVIDE SINGLE) binary instruction 7-254
DU (dispatchable unit) 5-48
DUALD (dispatchable-unit access-list designation) 5-55
DUCT (dispatchable-unit control table) 5-55, 5-66
DUCTO (dispatchable-unit-control-table origin) 5-53
dump (standalone) 12-5
DXBR (DIVIDE) BFP instruction 19-27
DXC (data-exception code) 6-17, 19-10, 20-15
  summary figures 6-17, 6-18
DXR (DIVIDE) HFP instruction 18-12
DXTRA (DIVIDE) DFP instruction 20-37
dynamic address translation (DAT) 3-38
  by LOAD REAL ADDRESS instruction 10-56
  by STORE REAL ADDRESS instruction 10-139
  control of 3-40
  explicit and implicit 3-52
  mode bit in PSW
    use in address translation 3-40
  mode bit in the PSW 4-6
  sequence of table fetches 5-119
dynamic-reconnection feature 13-3

E

E (exa) xxviii
E instruction format 5-6
EAR (EXTRACT ACCESS) instruction 7-257
early exception recognition 6-9
EAX
  See extended authorization index
EBCDIC (Extended Binary-Coded-Decimal Interchange Code)
  architecture designed for xxix
  character code
    table for I-2
ECAG (EXTRACT CPU ATTRIBUTE) instruction 7-257
ECC (error checking and correction) 11-2
ECTG (EXTRACT CPU TIME) instruction 7-260
ECW (extended-control word) 16-56
  indication in SCSW 16-12
ED (EDIT) instruction 8-8
  examples A-37
EDIT AND MARK instruction 8-11
  example A-38
EDIT instruction 8-8
  examples A-37
editing instructions 8-4
  See also ED instruction, EDMK instruction
EDMK (EDIT AND MARK) instruction 8-11
  example A-38
EEDTR (EXTRACT BIASED EXPONENT) DFP instruction 20-39
EEXTR (EXTRACT BIASED EXPONENT) DFP instruction 20-39
effective access-list designation 5-55
effective address 3-5
  controlled by addressing mode 5-10
  generation 5-10
  used for storage interlocks 5-115
effective address-space-control element 3-52
EFPC (EXTRACT FPC) floating-point-support instruction 9-30
EKM (entry key mask) 5-38
emergency signal (external interruption) 6-12
  signal-processor order 4-85
EMIF (ESCON-multiple-image facility) 1-32
EMW (extended-measurement word) 16-56
  in IRB 16-56
enabled (bit for TRAP) 10-175
enabled (bit in PMCW) 15-3
enabling for interruptions 6-6
  subchannel 16-6
enabling of subchannel 15-3, 16-6
encoded trailing significand field
  in DFP data format 20-5
endian 7-288, 7-387
ending of instruction execution 5-22
enhanced-DAT facility 1-10
  and change-bit override 3-10
  and DAT protection 3-14
  and format-control in R3TE 3-19
  and format-control in STE 3-10, 3-19
enhanced-DAT facility 1 1-9
  facility indication 4-99
enhanced-DAT facility1
  facility indication 4-101, 7-176, 7-177
enhanced-DAT facility 2
  facility indication 4-100
enhanced-monitor facility 1-11
  counter-array origin 5-110
  counter-array size 5-109
  exception count 5-109
  facility indication 4-99
  monitor masks 5-109

enhanced-monitor masks 4-11
enhanced-DAT-enablement control
and enhanced-DAT applicability 3-42
and the P bit in RTEs 3-47
definition 3-41
in control register 0 4-10
Enterprise Systems Connection Architecture (ESCON) I/O interface
publication referenced xxix
entropy-encoding compression facility 1-11
facility indication 4-100
entry
extended authorization index 5-76
key 5-76
entry (for tracing) 4-15
entry descriptor 5-79
entry index (EX) 5-33
entry key mask (EKM) 5-38
entry table (ET)
designation 5-36
length (ETL) 5-36, 5-37
origin (ETO) 5-36, 5-37
entry-table entry (ETE) 5-37, 5-75
entry-type code 5-79
environment (in STORE CPU ID result) 10-136
EPAIR (EXTRACT PRIMARY ASN AND INSTANCE) instruction 10-24
EPAR (EXTRACT PRIMARY ASN) instruction 10-24
epoch (for TOD clock) 4-51
definition 4-46
epoch index 4-47
epoch index (for TOD clock)
definition 4-46
EPSW (EXTRACT PSW) instruction 7-261
equipment check
in signal-processor status 4-95
ERC (error-recovery code) 17-27
See also CRW
EREG (EXTRACT STACKED REGISTERS) instruction 10-25
EREGG (EXTRACT STACKED REGISTERS) instruction 10-25
error
checking and correction 11-2
from DIAGNOSE instruction 10-23
I/O-error alert 16-50
indirect storage 11-20
intermittent 11-6
PSW-format 6-9
secondary (I/O) 16-50
solid 11-6
state of TOD clock 4-48
storage 11-19
storage-key 11-19
error-recovery code (ERC) 17-27
See also CRW
ERW (extended-report word) 16-47, 16-51
as result of channel-control check 16-29, 16-43
as result of channel-data check 16-28, 16-42, 16-43
ESA/370 architecture 1-32
ESA/390
compatibility with z/Architecture 1-34
ESA/390 architecture
architectural-mode controls 12-3
ESA/390-compatibility-mode facility 1-11, 5-111
facility indication 4-101
ESAIR (EXTRACT SECONDARY ASN AND INSTANCE) instruction 10-25
ESAR (EXTRACT SECONDARY ASN) instruction 10-24
ESCON (Enterprise Systems Connection Architecture) 13-3
ESCON (Enterprise Systems Connection Architecture) I/O interface
publication referenced xxix
ESCON channel-to-channel adapter
publication referenced xxix
ESCON-multiple-image facility (EMIF) 1-32
ESDTR (EXTRACT SIGNIFICANCE) DFP instruction 20-39
ESEA (EXTRACT AND SET EXTENDED AUTHORITY) instruction 10-24
ESTA (EXTRACT STACKED STATE) instruction 10-26
ESW (extended-status word) 16-47
See also extended status
ESW format bit (in SCSW) 16-9
ESXTR (EXTRACT SIGNIFICANCE) DFP instruction 20-39
ET
See entry table
ETE
See entry-table entry
ETF2-enhancement facility 1-11
facility indication 4-99
ETF3-enhancement facility 1-11
facility indication 4-99
ETL (entry-table length) 5-36, 5-37
ETND (EXTRACT TRANSACTION NESTING DEPTH) instruction 7-261
ETO (entry-table origin) 5-36, 5-37
ETR (external time reference) 2-7
ETR (external time reference) facility 1-31
ETR sync check (machine-check condition) 11-23
ETR-timing-alert condition
See timing alert
event 6-16

monitor 7-288
PER 4-26
space-switch 6-37
transactional-execution aborted 6-46
EX (entry index) 5-33
translation exception 6-26
EX (EXECUTE)
See EXECUTE instruction
EX-translation exception 6-26
exception access identification 3-74
in the transaction diagnostic block 5-96
exceptions 6-16
access (collective program-interruption name) 6-47, 6-53
addressing 6-20
AFX-translation 6-22
ALE-sequence 6-22
ALEN-translation 6-22
ALET-specification 6-22
ASCE-type 6-22
ASN-translation (collective program-interruption name) 6-56
associated with
ART 5-64
stacking process 5-86
unstacking process 5-89
ASTE-instance 6-23
ASTE-sequence 6-23
ASTE-validity 6-24
ASX-translation 6-24
crypto-operation 6-24
data 6-25
decimal-divide 6-26
decimal-overflow 6-26
delayed access (machine-check condition) 11-17
during translation 3-62
EX-translation 6-26
extended-authority 6-27
fixed-point-divide 6-27
fixed-point-overflow 6-27
HFP-divide 6-28
HFP-exponent-overflow 6-28
HFP-exponent-underflow 6-28
HFP-significance 6-28
HFP-square-root 6-29
IEEE 9-18
LFX-translation 6-29
LSTE-sequence 6-29
LSX-translation 6-29
LX-translation 6-30
operand (of I/O instruction) 6-31
operation 6-31
page-translation 6-32
PC-translation-specification 6-32
primary-authority 6-33
privileged-operation 6-33
protection 6-34
PSW-related 6-9
recognition of
early and late 6-9
region-first-translation 6-35
region-second-translation 6-36
region-third-translation 6-36
region translation 6-35
secondary-authority 6-37
segment-translation 6-37
special-operation 6-38
specification 6-40
stack-empty 6-44
stack-full 6-44
stack-operation 6-44
stack-specification 6-45
stack-type 6-45
subspace-replacement (collective program-interruption name) 6-56
trace (collective program-interruption name) 6-56
trace-table 6-45
transaction constraint 6-45
translation-specification 6-46
EXCLUSIVE OR IMMEDIATE instructions 7-256
EXCLUSIVE OR instructions 7-254
examples A-20
execute-extensions facility 1-11
facility indication 4-99
EXECUTE instruction 7-256
effect of address comparison on 12-1
example A-21
exceptions while fetching target of 6-8
PER event for target of 4-36
EXECUTE RELATIVE LONG instruction 7-256
execution-hint facility 1-11
facility indication 4-100
exigent machine-check conditions 11-10
expanded storage 2-2
block number 2-2
expansion 4-83
expansion dictionary
location of during compression 7-174
expansion-operation bit 7-173
expansion process
flowchart of 7-186
explicit address translation 3-52
explicit-trace-control bit 4-15
exponent 18-1
See also characteristic, floating point
overflow
HFP 18-1
signed value 9-2

underflow
HFP 18-1
mask in PSW 4-7
exponent bias 9-2, 19-2, 20-3
summary for BFP 19-3
summary for DFP 20-6
extended addressing mode
bit in entry-table entry 5-38
bit in trace entry 4-23
extended-authority exception 6-27
as an access exception 6-47
extended authorization 5-63
extended authorization index (EAX) 5-54
control bit 5-76
in entry-table entry 5-76
in linkage-stack state entry 5-82
extended binary-floating-point number 19-2
extended control (bit in SCSW) 16-12
extended-control word 16-56
See also ECW
extended decimal-floating-point number 20-4
extended hexadecimal-floating-point number 18-2
extended-I/O-measurement-block facility 1-12
extended-I/O-measurement-word facility 1-12
extended-immediate facility 1-12
facility indication 4-99
extended-measurement-word 17-11
extended-measurement-word enable (I/O)
update enable 15-8
extended measurement word mode enable (I/O) 15-8
extended-report word
See ERW
extended-save-area address
machine-check save area for 3-86
store-status save area for 3-86
extended-sorting facility 1-31
extended status
See also ESW
flags in subchannel logout for 16-47
format-0 16-47
format-1 16-53
format-2 16-54
format-3 16-55
secondary-CCW address 16-53
extended-status word 16-47
See also extended status
extended-status-word-format bit 16-9
extended-subchannel-logout descriptor (in ERW) 16-53
in ESW 16-53
extended-subchannel-logout pending 16-52
indicator for (in ERW) 16-52
extended-translation facility 2 1-12
facility indication 4-99
extended-translation facility 3 1-13
facility indication 4-99
external call
external interruption 6-13
pending (signal-processor status) 4-96
signal-processor order 4-85
external damage 11-16
subclass-mask bit for 11-27
external-damage code 11-23
assigned storage locations for 3-80, 3-86
validity bit for 11-21
external interruption 6-10
clock-comparator 6-12, 4-59
CPU-timer 4-62, 6-12
direct conditions 6-11
emergency-signal 6-12
external-call 6-13
interrupt-key 6-13
malfunction-alert 6-13
mask in the PSW 4-6
measurement-alert 6-13
parameter 6-11
assigned storage locations for 3-74, 3-84
pending conditions 6-11
priority of conditions 6-11
service-signal 6-14
timing-alert 6-14
external time reference (ETR) 2-7
external-time-reference (ETR) facility 1-31
externally initiated functions 4-73
I/O 17-16
EXTRACT ACCESS instruction 7-257
EXTRACT AND SET EXTENDED AUTHORITY instruction 10-24
EXTRACT BIASED EXPONENT DFP instructions 20-39
extract-CPU-time facility 1-13, 7-261
facility indication 4-99
EXTRACT CPU TIME instruction 7-260
EXTRACT FPC floating-point-support instruction 9-30
EXTRACT PRIMARY ASN AND INSTANCE instruction 10-24
EXTRACT PRIMARY ASN instruction 10-24
EXTRACT PSW instruction 7-261
EXTRACT SECONDARY ASN AND INSTANCE instruction 10-25
EXTRACT SECONDARY ASN instruction 10-24
EXTRACT SIGNIFICANCE DFP instructions 20-39
EXTRACT STACKED REGISTERS instruction 10-25
EXTRACT STACKED STATE instruction 10-26
EXTRACT TRANSACTION NESTING DEPTH instruction 7-261

extraction-authority-control bit 5-28

F

facility indication
access-exception-fetch/store-indication facility 4-100
ASN-and-LX-reuse facility 4-99
CMPSC-enhancement facility 4-100
compare-and-swap-and-store facility 4-99
compare-and-swap-and-store facility 2 4-99
conditional-SSKE facility 4-99
configuration-topology facility 4-99
configuration-z/architecture-architectural-mode facility 4-101
constrained transactional-execution facility 4-100
CPU-measurement-counter facility 4-100
CPU-measurement-sampling facility 4-100
DAT-enhancement facility 4-98
decimal-floating-point facility 4-99
decimal-floating-point high-performance 4-99
decimal-floating-point packed-conversion facility 4-100
decimal-floating-point zoned-conversion facility 4-100
distinct-operands facility 4-100
enhanced-DAT facility 1 4-99
enhanced-DAT facility 2 4-100
enhanced-DAT facility 1 4-101, 7-176, 7-177
enhanced-monitor facility 4-99
entropy-encoding compression facility 4-100
ESA/390-compatibility-mode facility 4-101
ETF2-enhancement facility 4-99
ETF3-enhancement facility 4-99
execute-extensions facility 4-99
execution-hint facility 4-100
extended-immediate facility 4-99
extended-translation facility 2 4-99
extended-translation facility 3 4-99
extract-CPU-time facility 4-99
fast-BCR-serialization facility 4-100
floating-point-extension facility 4-99
floating-point-support-enhancement facilities 4-99
general-instructions-extension facility 4-99
guarded-storage facility 4-100
HFP-multiply-and-add/subtract facility 4-99
HFP-unnormalized-extension facility 4-99
high-word facility 4-100
insert-reference-bits-multiple facility 4-101
instruction-execution-protection facility 4-100
interlocked-access facility 1 4-100
interlocked-access facility 2 4-100
IPTE-range facility 4-99
load-and-trap facility 4-100
load-and-zero-rightmost-byte facilty 4-100
load/store-on-condition facility 1 4-100
local-TLB-clearing facility 4-100
long-displacement facility 4-99
long-displacement-facilty high-performance 4-99
message-security-assist extension 3 4-100
message-security-assist extension 4 4-100
message-security-assist extension 5 4-100
message-security-assist extension 8 4-101
message-security assist 4-99
miscellaneous-instruction-extensions facility 1 4-100
miscellaneous-instruction-extensions facility 2 4-100
move-with-optional-specifications facility 4-99
nonquiescing key-setting facility 4-99
order-preserving-compression facility 4-99
parsing-enhancement facility 4-99
PFPO facility 4-99
population-count facility 4-100
processor-assist facility 4-100
reset-reference-bits-multiple facility 4-100
sense-running-status facility 4-99
set-program-parameters facility 4-99
side-effect-access facility 4-100
store-clock-fast facility 4-99
store-CPU-counter-multiple facility 4-101
store-facility-list-extended facility 4-99
store-hypervisor-information facility 4-100
test-pending-external-interruption 4-101
TOD-clock epoch-extension facility 4-101
TOD-clock-steering facility 4-99
transactional-execution facility 4-100
vector-enhancements facility 1 4-101
vector facility for z/Architecture 4-100
vector-packed-decimal facility 4-101
z/architecture active 4-98
z/architecture installed 4-98
z/architecture instructions in ESA/390 mode 4-98
facility indications 4-98
failing-storage address 11-24
assigned storage locations for 3-80, 3-86
in ESW 16-47, 16-53
as result of channel-control check 16-29, 16-43
as result of channel-data check 16-28, 16-42, 16-43
validity bit for 11-21
validity flag for (in ERW) 16-52
failing-storage-address format
indicator for (in ERW) 16-53
fast-BCR-serialization facility 1-13
facility indication 4-100

FCX-bidirectional-data-transfer facility 1-14
FCX facility 1-13
fetch-only bit 5-56
fetch protection 3-11
bit in segment-table entry 3-46, 3-49
bit in storage key 3-9
override-control bit 3-13
fetch reference 5-123
access exceptions for 6-50
fetching
handling of invalid CBC in storage during 11-8
of ART-table and DAT-table entries 5-119
of instructions 5-118
of PSWs during interruptions 5-129
of storage operands 5-123
Fibre-Channel Extensions 13-6
fibre-channel-extensions facility 1-13
Fibre-Channel Extensions 13-6
FICON I/O interface
publication referenced xxix
FIDBR (LOAD FP INTEGER) BFP instruction 19-32
FIDBRA (LOAD FP INTEGER) BFP instruction 19-32
FIDR (LOAD FP INTEGER) HFP instruction 18-15
FIDTR (LOAD FP INTEGER) DFP instruction 20-42
FIEBR (LOAD FP INTEGER) BFP instruction 19-32
FIEBRA (LOAD FP INTEGER) BFP instruction 19-32
field 3-3
field separator (in EDIT) 8-8
field-validity flags (in subchannel logout) 16-49
relation to channel-control check of 16-29
FIER (LOAD FP INTEGER) HFP instruction 18-15
FIFO (first in first out) queuing
example for lock and unlock A-49
fill byte (in EDIT) 8-8
filtering
of program interruptions 5-103
FIND LEFTMOST ONE instruction 7-262
example A-22
FIXBR (LOAD FP INTEGER) BFP instruction 19-32
FIXBRA (LOAD FP INTEGER) BFP instruction 19-32
fixed-length field 3-3
fixed logout
assigned storage locations for 3-82
machine-check 11-27
fixed point
See also binary
divide exception 6-27
overflow exception 6-27
mask in PSW 4-7
See also binary
FIXR (LOAD FP INTEGER) HFP instruction 18-15
FIXTR (LOAD FP INTEGER) DFP instruction 20-42
flags
for IEEE exceptions 9-10
in the transaction diagnostic block 5-95
floating interruption conditions 6-6, 11-25
clearing of 4-79
floating point
See also exponent
binary (BFP) 9-4
binary data format 19-2
conversion
between formats 9-23, 19-5
conversion from binary integer 18-11, 19-19, 20-24
conversion from unsigned binary integer 19-21, 20-25
conversion to binary integer 18-11, 19-22, 20-29
conversion to unsigned binary integer 19-25, 20-32
data
lengthening format of 18-15, 19-33, 20-45
shortening format of 18-17, 19-35, 20-46
data class 19-4
decimal (DFP) 9-4
hexadecimal (HFP) 9-3
hexadecimal data formats 18-2
instructions 9-1
numbers 19-2
registers 2-4, 9-8
clearing of 9-35
save areas for 3-82
validity bit for 11-21
shifting
See normalization
floating-point
registers
machine-check save area for 3-87
store-status save area for 3-87
floating-point-control (FPC) register 9-9
save area for 3-82
validity bit for 11-22
floating-point extension facility 1-14
floating-point-extension facility
facility indication 4-99
floating-point-support-enhancement facilities
facility indication 4-99
floating-point-support-sign-handling facility 1-15
FLOGR (FIND LEFTMOST ONE) instruction 7-262
example A-22
format
address 3-3
binary-floating-point data 19-2
CCW
See CCW format control
decimal data 8-1
error
in DAT-table entry 6-46

in PSW 6-9
general data 7-3
in the transaction diagnostic block 5-94
information 3-3
instruction 5-3
PSW 4-5
format-0 and format-1 CCWs 15-31
format-2-IDAW control 15-27
format indication (in STORE CPU ID result) 10-136, 10-137
format-1-sibling-descriptor bit 7-174
formats
hexadecimal-floating-point data 18-2
forward section header address 5-81
forward-section validity bit 5-81
FPC
See floating-point-control (FPC) register
FPC (floating-point-control) register 9-9
FPR-GR-transfer facility 1-15
fraction 18-1
fraction view of floating-point numbers 9-2
free-pool manipulation
programming example A-50
freeze 7-180
fullword
See word
function control (I/O) 16-13
function-pending time 17-2
in extended-measurement word 16-57
functionally-constrained rounding 9-12
functionally-rounded value 9-12
function-pending time
in measurement block 17-5

G

G (giga) xxviii
general instructions 7-3
examples A-7
general-instructions-extension facility 1-15
facility indication 4-99
general-operand data exception 6-25, 8-5
general register
save areas 3-82
general registers 2-4
machine-check save area for 3-87
store-status save area for 3-87
validity bit for 11-21
glue module 5-21
GMT (Greenwich Mean Time) obsolete term for UTC 4-51
Greenwich Mean Time (GMT) obsolete term for Coordinated Universal Time 4-51
guard digit 18-4
guarded-storage facility 1-15, 4-64
facility indication 4-100
guarded-storage-event parameter list 4-66
guarded-storage-facility registers 4-64
guarded-storage-designation register 4-65
guarded-storage-event-parameter-list register 4-66
guarded-storage-facility-enablement control 4-64
guarded-storage-section-mask register 4-65
operation 4-69
event detection 4-69
event processing 4-70

H

halfword 3-3
halfword-concurrent storage references 5-126
halt function 15-16
bit in SCSW for 16-13
completion of 15-17
initiated by HALT SUBCHANNEL 14-6
path management for 15-16
pending 16-16
signaling for 15-17
halt signal 17-12
issued as part of halt function 15-17
HALT SUBCHANNEL instruction 14-6
See also halt function
effect on SCSW count field 15-19
function initiated by 15-16
use of after RESET CHANNEL PATH 14-10
HALVE HFP instructions 18-13
example A-43
HASCE (home address-space-control element) 3-45
HDR (HALVE) HFP instruction 18-13
example A-43
HER (HALVE) HFP instruction 18-13
hex
See hexadecimal
hexadecimal (hex) representation 5-7
hexadecimal floating point
conversion
examples with instructions A-44
instructions
examples A-41
hexadecimal floating point (HFP) 9-3
conversion
basic example A-6
hexadecimal-floating-point number
examples A-5
HFP (hexadecimal floating point) 9-3
HFP data 18-2
conversion of 9-27

HFP exponent
overflow
exception 6-28
underflow
exception 6-28
HFP-multiply-and-add/subtract facility 1-16
facility indication 4-99
HFP significance
exception 6-28
HFP square root
exception 6-29
HFP-unnormalized-extension facility 1-16
facility indication 4-99
HFP-unnormalized-extensions facility 1-16
high-speed data transfer (I/O) 13-4
high-word facility 1-16
facility indication 4-100
home address space 3-23, 5-44
facilities 5-44
home designation-type-control bits 3-45
home private-space-control bit 3-45
home real-space-control bit 3-45
home real-space token origin (HRSTKO) 3-45
home region table
designation (HRTD) 3-45
length (HRTL) 3-45
origin (HRTO) 3-45
home segment table
designation (HSTD) 3-45
length (HSTL) 3-45
origin (HSTO) 3-45
home-space mode 3-40
home space-switch-event-control bit 3-45
home storage-alteration-event-control bit 3-45
home virtual address 3-5
effective address-space-control element for 3-52
HRSD (home real-space designation) 3-45
HRTD (home region-table designation) 3-45
HRTL (home region-table length) 3-45
HRTO (home region-table origin) 3-45
HSCH (HALT SUBCHANNEL) instruction 14-6
HSTD (home segment-table designation) 3-45
HSTL (home segment-table length) 3-45
HSTO (home segment-table origin) 3-45

I

I field of instruction 5-8
I instruction format 5-6
I/O (input/output) 2-7
basic functions of 15-1
blocking of data for 15-24
effect on CPU timer 4-62
sense data
See sense data
support functions of 17-1
I/O addressing 13-5
I/O-command words
CCW 13-7
DCW 13-7
I/O commands
See also commands
publication referenced xxix
I/O device
See device
I/O-error alert (in subchannel logout) 16-50
I/O instructions 14-2
deferred condition code for 16-9
operand access by 14-2
role of in I/O operations 13-8
I/O interface
ESCON publication referenced xxix
OEMI publication referenced xxix
I/O interruption 6-15, 16-1
See also interruption
action for 16-6
mask in the PSW 4-6
masking of 13-12
priority of 16-5
program-controlled interruption
See PCI)
I/O-interruption code 6-15
interruption-identification word in 14-20
stored by TPI 14-20
I/O-interruption condition 13-12, 16-2
alert 16-5
intermediate 16-4
primary 13-10, 16-4
secondary 13-10, 16-5
solicited 16-3
unsolicited 16-3
I/O-interruption-identification word 14-20
assigned storage locations for 3-79, 3-85
I/O-interruption parameter
assigned storage locations for 3-79, 3-85
in I/O-interruption code 14-20
in ORB 15-25
in PMCW 15-2
used for IPL 17-17
I/O-interruption request
clearing of 13-12
from subchannels 16-6
I/O-interruption subclass 13-12
I/O-interruption subclass code
See ISC
I/O-interruption subclass mask 6-15, 16-6
relation to priority 16-5
I/O operations 13-8

conclusion of
See conclusion of I/O operations
initiated indication for 16-12
termination of
See conclusion of I/O operations
I/O-system reset 17-13
as part of subsystem reset 4-79
IAC (INSERT ADDRESS SPACE CONTROL) instruction 10-29
IC (INSERT CHARACTER) instruction 7-262
IC (instruction counter)
See instruction address
ICM (INSERT CHARACTERS UNDER MASK) instruction 7-262
examples A-23
ICMH (INSERT CHARACTERS UNDER MASK) instruction 7-262
ICMY (INSERT CHARACTERS UNDER MASK) instruction 7-262
ICY (INSERT CHARACTER) instruction 7-262
IDA (indirect-data address) 15-66
flag in CCW 15-32
IDAW (indirect-data-address word) 15-66
check (in subchannel logout) 16-48
invalid address of 16-25
invalid address specification in 16-25
invalid address specification of 16-26
idle state for subchannel 16-14
IDTE (INVALIDATE DAT TABLE ENTRY) instruction 10-32
IEDTR (INSERT BIASED EXPONENT) DFP instruction 20-40
IEEE exception conditions
summary figure 19-10, 24-2
IEEE-exception data exception 6-25, 9-18
IEEE-exception-simulation facility 1-16
IEEE rounding 9-13
IEEE standard 1-30
IEEE trap action and nontrap action 9-18
IEEE-exception data exception 19-10, 24-2
IEXTR (INSERT BIASED EXPONENT) DFP instruction 20-40
IFCC (interface-control check) 16-29
IIHF (INSERT IMMEDIATE) instruction 7-263
IIHH (INSERT IMMEDIATE) instruction 7-263
IIHL (INSERT IMMEDIATE) instruction 7-263
IILF (INSERT IMMEDIATE) instruction 7-263
IILH (INSERT IMMEDIATE) instruction 7-263
IILL (INSERT IMMEDIATE) instruction 7-263
ILC (instruction-length code) 6-7
assigned storage locations for 3-74, 3-84
for program interruptions 6-16
IML (initial machine loading) controls 12-3
immediate operand 5-8
immediate operation
SLI flag in CCW for 15-60
implicit address translation 3-52
in DFP data 20-5
incorrect length (subchannel status) 16-24, 16-39
for immediate operations 15-60
incorrect-length-indication mode 15-28
incorrect-length-indication-suppression facility 17-24
effect on immediate operation 15-60
incorrect-length-suppression mode 15-28
incorrect state (signal-processor status) 4-96
index
for address generation 5-11
instructions for branching on 7-42
into access list 5-54
into ASN first and second tables 3-30
into authority table 5-29
into entry and linkage tables 5-33
register for 2-4
indicator
check-stop 12-3
CPUs-per-core 12-3
load 12-3
manual 12-4
mode 12-2, 12-3
test 12-6
wait 12-6
indirect-data address
See IDA
indirect-data-address word
See IDAW
indirect storage error 11-20
infinities 9-2, 19-4
classes of DFP data 20-7
information format 3-3
initial-command-response measurement (I/O) 17-9
initial-command-response time (in measurement block) 17-6
initial CPU reset 4-78
signal-processor order 4-85
initial-machine-loading (IML) controls 12-3
initial program loading
See IPL
initial-program-loading CCW1 (assigned storage locations) 3-73, 3-83
initial-program-loading CCW2 (assigned storage locations) 3-74, 3-83
initial-program-loading PSW (assigned storage locations) 3-73, 3-83
initial-status-interruption control 15-27, 16-12
relation to Z bit 16-12
used for IPL 17-17
inoperative (signal-processor status) 4-97
input/output

See I/O
INSERT ADDRESS SPACE CONTROL instruction 10-29
INSERT BIASED EXPONENT DFP instructions 20-40
INSERT CHARACTER instructions 7-262
INSERT CHARACTERS UNDER MASK instruction 7-262
examples A-23
INSERT IMMEDIATE instructions 7-263
INSERT PROGRAM MASK instruction 7-264
INSERT PSW KEY instruction 10-30
INSERT REFERENCE BIT MULTIPLE instruction 10-30
insert-reference-bits-multiple facility 1-16
facility indication 4-101
INSERT STORAGE KEY EXTENDED instruction 10-30
INSERT VIRTUAL STORAGE KEY instruction 10-31
installation 2-1
instruction address
as a type of address 3-5
handling by DAT 3-40
in entry-table entry 5-37
in the PSW 4-7
validity bit for 11-21
instruction-execution protection 3-10
instruction-execution-protection facility 1-16, 3-14
facility indication 4-100
instruction-fetching-nullification-event mask 4-27
instruction format 5-6
instruction-length code
preserved by BRANCH AND LINK 5-15, 7-36
See ILC
instruction-processing damage 11-15
resulting in processing backup 11-18
resulting in processing damage 11-19
instructions
See also instruction lists and page numbers in Appendix B
backing up of 11-18
classes of 2-2
control 10-1
damage to 11-15, 11-19
decimal 8-1
examples A-36
divisible execution of 5-115
ending of 5-22
examples of use A-7
execution of 5-14
fetching of 5-118
access exception for 6-50
PER event for 4-36
PER-event mask for 4-27
floating-point 9-1
format of 5-3
general 7-3
examples A-7
hexadecimal-floating-point
examples A-41
I/O
See I/O Instructions 13-1
interruptible
See interruptible instructions
length of 5-7
list of B-1
modification by execute-type instruction 7-256
prefetching of 5-118
privileged 4-6
for control 10-1
See instruction lists and page numbers in Appendix B
semiprivileged 4-6, 10-2
sequence of execution of 5-3
stepping of (rate control) 12-5
effect on CPU state 4-2
effect on CPU timer 4-63
unprivileged 4-6, 7-3
integer
binary 7-4
address as 5-11
conversion from floating point 18-11, 19-22, 19-25, 20-29, 20-32
conversion to floating point 18-11, 19-19, 20-24, 20-25
examples A-2
decimal 8-3
unsigned binary
conversion to floating point 19-21
integer quotient 19-28
integral boundary 3-3
octoword 3-3
interface
ESCON I/O
publication referenced xxix
parallel-I/O
OEMI publication referenced xxix
serial-I/O
publication referenced xxix
interface-control check (subchannel status) 16-29
interlocked-access facility 1 1-17
facility indication 4-100
interlocked-access facility 2 1-17
facility indication 4-100
interlocked-update storage reference 5-123, 5-124
interlocks for virtual storage references 5-115
intermediate interruption condition (I/O) 16-4
intermediate-status bit (I/O) 16-18, 16-36

intermediate values 9-12
intermittent errors 11-6
International Atomic Time (TAI) related to Coordinated Universal Time 4-51
interpretive execution
publication referenced xxx
interpretive-execution facility 1-32
interrogate TCW
address of 15-36
interrupt-delay time (in measurement block) 17-6
interrupt key 12-3
external interruption 6-13
interruptible instructions 5-24
COMPARE AND FORM CODEWORD 7-137
COMPARE LOGICAL LONG 7-159
COMPARE UNTIL SUBSTRING EQUAL 7-169
MOVE LONG 7-292
PER event affecting the ending of 4-33
stopping of 4-2
TEST BLOCK 10-168
UPDATE TREE 7-419
interruption 6-2
See also masks
action 6-2
I/O 16-6
machine-check 11-11
classes of 6-5
effect on instruction sequence 5-22
external
See external interruption
I/O
See I/O interruption
machine-check
See machine-check interruption
masking of 6-6
pending 6-6
external 6-11
machine-check 11-13
relation to CPU state 4-2
priority of
See priority
program
See program interruption
program-controlled (I/O)
See PCI
restart 6-56
string
See string of interruptions
supervisor-call 6-57
interruption code 6-5
external 6-11
I/O
See I/O-interruption code
machine-check (MCIC) 3-80, 3-86, 11-14
program 6-16
summary of 6-2
interruption conditions 6-2
clearing of 4-77
floating 6-6, 11-25
I/O
See I/O-interruption condition
interruption parameter
external (assigned storage locations) 3-74, 3-84
I/O
See I/O-interruption parameter
interruption-response block
See IRB
interruption subclass
See I/O-interruption subclass
invalid
access-list entry 5-56
address 6-20
bit in ASN-first-table entry 3-31
bit in ASN-second-table entry 3-31
bit in linkage-first-table entry 5-36
bit in linkage-second-table entry 5-36
bit in linkage-table entry 5-36
bit in page-table entry 3-51
bit in region-table entry 3-47
bit in segment-table entry 3-50
CBC 11-2
in registers 11-9
in storage 11-7
in storage keys 11-7
operation code 6-31
order (signal-processor status) 4-97
parameter (signal-processor status) 4-96
translation address 3-62
translation format
exception recognition 3-62
invalid address specification
in channel-program address 16-25, 16-40
in IDAW 16-26
in MIDAW 16-26
of data in CCW 16-26
of IDAW 16-25
of TIC CCW 16-25, 16-40
invalid CCW field
command code 16-25
count 16-25
data address 16-26
MIDA flag 16-27
suspend flag 16-27
invalid format
of CCW 16-27
of ORB 16-27
invalid sequence of CCWs 16-27
INVALIDATE DAT TABLE ENTRY instruction 10-32

INVALIDATE PAGE TABLE ENTRY instruction 10-37
effect of when CPU is stopped 4-3
inverse move
See MOVE INVERSE instruction, move-inverse facility
IPK (INSERT PSW KEY) instruction 10-30
IPL (initial program loading) 4-81, 17-16
IPM (INSERT PROGRAM MASK) instruction 7-264
IPTE (INVALIDATE PAGE TABLE ENTRY) instruction 10-37
IPTE-range facility 1-17
facility indication 4-99
IRB (interruption-response block) 16-6
See also ECW, ERW, ESW, SCSW
storage requirements for 16-13
IRBM (INSERT REFERENCE BIT MULTIPLE) instruction 10-30
ISC (I/O-interruption-subclass code) 15-2
enhanced 14-20
ISKE (INSERT STORAGE KEY EXTENDED) instruction 10-30
isolated state 16-52
IVSK (INSERT VIRTUAL STORAGE KEY) instruction 10-31

K

K (kilo) xxviii
KDB (COMPARE AND SIGNAL) BFP instruction 19-18
KDBR (COMPARE AND SIGNAL) BFP instruction 19-18
KDTR (COMPARE AND SIGNAL) DFP instruction 20-23
KEB (COMPARE AND SIGNAL) BFP instruction 19-18
KEBR (COMPARE AND SIGNAL) BFP instruction 19-18
key
access
See access key
manual
See manual operation
PSW
See PSW key
storage
See storage key
subchannel
See subchannel key
key check (in subchannel logout) 16-47
key-controlled protection 3-11
exception for 6-34
key mask
authorization 5-38
entry 5-38
PSW (PKM) 5-28
KIMD (COMPUTE INTERMEDIATE MESSAGE DIGEST) instruction 7-188
KLMD (COMPUTE LAST MESSAGE DIGEST) instruction 7-201
KM (CIPHER MESSAGE) instruction 7-53
KMA (CIPHER MESSAGE WITH AUTHENTICATION) instruction 7-78
KMAC (COMPUTE MESSAGE AUTHENTICATION CODE) instruction 7-219
KMC (CIPHER MESSAGE WITH CHAINING) instruction 7-53
KMCTR (CIPHER MESSAGE WITH COUNTER) instruction 7-107
KMF (CIPHER MESSAGE WITH CIPHER FEEDBACK) instruction 7-92
KMO (CIPHER MESSAGE WITH OUTPUT FEEDBACK) instruction 7-120
KXBR (COMPARE AND SIGNAL) BFP instruction 19-18
KXTR (COMPARE AND SIGNAL) DFP instruction 20-23

L

L (LOAD) binary instruction 7-264
example A-23
L fields of instruction 5-9
LA (LOAD ADDRESS) instruction 7-266
examples A-23
LAA (LOAD AND ADD) instruction 7-268
LAAG (LOAD AND ADD) instruction 7-268
LAAL (LOAD AND ADD LOGICAL) instruction 7-268
LAALG (LOAD AND ADD LOGICAL) instruction 7-268
LAE (LOAD ADDRESS EXTENDED) instruction 7-266
LAEY (LOAD ADDRESS EXTENDED) instruction 7-266
LAM (LOAD ACCESS MULTIPLE) instruction 7-265
LAMY (LOAD ACCESS MULTIPLE) instruction 7-265
LAN (LOAD AND AND) instruction 7-269
LANG (LOAD AND AND) instruction 7-269
LAO (LOAD AND OR) instruction 7-270
LAOG (LOAD AND OR) instruction 7-270
LARL (LOAD ADDRESS RELATIVE LONG) instruction 7-267
LASP (LOAD ADDRESS SPACE PARAMETERS) instruction 10-41
last-path-used mask
See LPUM

LAT (LOAD AND TRAP) instruction 7-271
late exception recognition 6-10
LATG (LOAD LOGICAL AND TRAP) instruction 7-271
LATH (LOAD HIGH AND TRAP) instruction 7-278
LAX (LOAD AND EXCLUSIVE OR) instruction 7-269
LAXG (LOAD AND EXCLUSIVE OR) instruction 7-269
LAY (LOAD ADDRESS) instruction 7-266
LB (LOAD BYTE) instruction 7-272
LBH (LOAD BYTE HIGH) instruction 7-272
LBR (LOAD BYTE) binary instruction 7-272
LCCB (LOAD COUNT TO BLOCK BOUNDARY) binary instruction 7-273
LCDBR (LOAD COMPLEMENT) BFP instruction 19-31
LCDFR (LOAD COMPLEMENT) floating-point-support instruction 9-31
LCDR (LOAD COMPLEMENT) HFP instruction 18-14
LCEBR (LOAD COMPLEMENT) BFP instruction 19-31
LCER (LOAD COMPLEMENT) HFP instruction 18-14
LCGFR (LOAD COMPLEMENT) binary instruction 7-273
LCGR (LOAD COMPLEMENT) binary instruction 7-273
LCR (LOAD COMPLEMENT) binary instruction 7-272
LCTL (LOAD CONTROL) instruction 10-50
LCTLG (LOAD CONTROL) instruction 10-50
LCXBR (LOAD COMPLEMENT) BFP instruction 19-31
LCXR (LOAD COMPLEMENT) HFP instruction 18-14
LD (LOAD) floating-point-support instruction 9-31
LDE (LOAD LENGTHENED) HFP instruction 18-15
LDEB (LOAD LENGTHENED) BFP instruction 19-34
LDEBR (LOAD LENGTHENED) BFP instruction 19-33
LDER (LOAD LENGTHENED) HFP instruction 18-15
LDETR (LOAD LENGTHENED) DFP instruction 20-45
LDGR (LOAD FPR FROM GR) floating-point-support instruction 9-34
LDR (LOAD) floating-point-support instruction 9-31
LDXBR (LOAD ROUNDED) BFP instruction 19-35
LDXBRA (LOAD ROUNDED) BFP instruction 19-35
LDXR (LOAD ROUNDED) HFP instruction 18-17
LDXTR (LOAD ROUNDED) DFP instruction 20-46
LDY (LOAD) floating-point-support instruction 9-31
LE (LOAD) floating-point-support instruction 9-31
leaf
  DAT-table entry 3-42, 3-53, 3-54, 3-65, 3-46
LEDBR (LOAD ROUNDED) BFP instruction 19-35
LEDBRA (LOAD ROUNDED) BFP instruction 19-35
LEDR (LOAD ROUNDED) HFP instruction 18-17
LEDTR (LOAD ROUNDED) DFP instruction 20-46
left-to-right addressing 3-2
left-units view (LUV) of floating-point numbers 9-2
length
  field 3-3
  instruction 5-7
  of BFP data
    decreasing 19-35
    increasing 19-33
  of DFP data
    decreasing 20-46
    increasing 20-45
  of HFP data
    decreasing 18-17
    increasing 18-15
  register-operand 5-8
  second operand same as first 5-8
  variable (storage operand) 5-9
LER (LOAD) floating-point-support instruction 9-31
LEXBR (LOAD ROUNDED) BFP instruction 19-35
LEXBRA (LOAD ROUNDED) BFP instruction 19-35
LEXR (LOAD ROUNDED) HFP instruction 18-17
LEY (LOAD) floating-point-support instruction 9-31
LFAS (LOAD FPC AND SIGNAL) floating-point-support instruction 9-32
LFH (LOAD HIGH) instruction 7-278
LFPC (LOAD FPC) floating-point-support instruction 9-31
LFT (linkage first table) 5-36
LFTD (linkage-first-table designation) 5-35
LFTL (linkage-first-table length) 5-35
LFTO (linkage-first-table origin) 5-35
LFX (linkage first index)
  invalid bit 5-36
  translation exception 6-29
LG (LOAD) binary instruction 7-264
LGB (LOAD BYTE) instruction 7-272
LGBR (LOAD BYTE) binary instruction 7-272
LGDR (LOAD GR FROM FPR) floating-point-support instruction 9-34
LGF (LOAD) binary instruction 7-264
LGFI (LOAD IMMEDIATE) instruction 7-264
LGFR (LOAD) binary instruction 7-264
LGFRL (LOAD RELATIVE LONG) binary instruction 7-264
LGG (LOAD GUARDED) instruction 7-274
LGH (LOAD HALFWORD) instruction 7-276
LGHI (LOAD HALFWORD IMMEDIATE) instruction 7-276
LGHR (LOAD HALFWORD) binary instruction 7-276

LGHRL (LOAD HALFWORD RELATIVE LONG) binary instruction 7-276
LGR (LOAD) binary instruction 7-264
LGRL (LOAD RELATIVE LONG) binary instruction 7-264
LGSC (LOAD GUARDED STORAGE CONTROLS) instruction 7-275
LH (LOAD HALFWORD) instruction 7-276
  examples A-24
LHH (LOAD HALFWORD HIGH) instruction 7-277
LHI (LOAD HALFWORD IMMEDIATE) instruction 7-276
LHR (LOAD HALFWORD) binary instruction 7-276
LHRL (LOAD HALFWORD RELATIVE LONG) binary instruction 7-276
LHY (LOAD HALFWORD) instruction 7-276
LIFO (last in first out) queuing
  example for lock and unlock A-48
light
  See indicator
limit mode (I/O) 15-3
link information
  for BRANCH AND LINK instruction 7-35
  for BRANCH AND SAVE AND SET MODE instruction 7-37
  for BRANCH AND SAVE instruction 7-36
linkage first table (LFT) 5-36
  designation (LFTD) 5-35
  length (LFTL) 5-35
  origin (LFTO) 5-35
linkage for subroutines 5-14
linkage index (LX) 5-33
linkage second table (LST) 5-36
  origin (LSTO) 5-36
linkage stack 5-70, 5-79
  associated PER events 5-75
  associated trace entries 5-75
  branch state entry 10-11
  entry address 5-78
  entry descriptor 5-79
  entry-type code 5-79
  handling of information in 5-74
  instructions 5-71
  introduction 5-76
  next-entry size 5-80
  operations 5-76
    control 5-78
  program-call state entry 10-94
  remaining free space 5-79
  section 5-76
    identification 5-79
  state entry 5-81
linkage-stack functions 5-71
linkage table (LT) 5-36
  designation (LTD) 5-35
  length (LTL) 5-35
  origin (LTO) 5-35
list-directed initial-program-load (IPL) function 1-17
list-directed IPL machine-loader execution-space size (assigned-storage locations) 3-74, 3-83
list-directed IPL system-parameter-block address (assigned-storage locations) 3-74, 3-83
list-directed IPL (list-directed initial program loading) 17-19
little endian 7-288, 7-387
LLC (LOAD LOGICAL CHARACTER) instruction 7-279
LLCH (LOAD LOGICAL CHARACTER HIGH) instruction 7-280
LLCR (LOAD LOGICAL CHARACTER) binary instruction 7-279
LLGC (LOAD LOGICAL CHARACTER) instruction 7-279
LLGCR (LOAD LOGICAL CHARACTER) binary instruction 7-279
LLGF (LOAD LOGICAL) instruction 7-278
LLGFR (LOAD LOGICAL) instruction 7-278
LLGFRL (LOAD LOGICAL RELATIVE LONG) binary instruction 7-278
LLGFSG (LOAD LOGICAL AND SHIFT GUARDED) instruction 7-274
LLGH (LOAD LOGICAL HALFWORD) instruction 7-280
LLGHR (LOAD LOGICAL HALFWORD) binary instruction 7-280
LLGHRL (LOAD LOGICAL RELATIVE LONG) binary instruction 7-280
LLGT (LOAD LOGICAL THIRTY ONE BITS) instruction 7-282
LLGTR (LOAD LOGICAL THIRTY ONE BITS) instruction 7-281
LLH (LOAD LOGICAL HALFWORD) instruction 7-280
LLHH (LOAD LOGICAL HALFWORD HIGH) instruction 7-281
LLHR (LOAD LOGICAL HALFWORD) binary instruction 7-280
LLHRL (LOAD LOGICAL RELATIVE LONG) binary instruction 7-280
LLIHF (LOAD LOGICAL IMMEDIATE) instruction 7-281
LLIHH (LOAD LOGICAL IMMEDIATE) instruction 7-281
LLIHL (LOAD LOGICAL IMMEDIATE) instruction 7-281
LLILF (LOAD LOGICAL IMMEDIATE) instruction 7-281

LLILH (LOAD LOGICAL IMMEDIATE) instruction 7-281
LLILL (LOAD LOGICAL IMMEDIATE) instruction 7-281
LLTG (LOAD AND TRAP) instruction 7-271
LLTGF (LOAD LOGICAL AND TRAP) instruction 7-279
LLTGT (LOAD LOGICAL THIRTY ONE BITS AND TRAP) instruction 7-282
LLZRGF (LOAD LOGICAL AND ZERO RIGHTMOST BYTE) binary instruction 7-279
LM (LOAD MULTIPLE) instruction 7-282
LMD (LOAD MULTIPLE DISJOINT) instruction 7-283
LMG (LOAD MULTIPLE) instruction 7-282
LMH (LOAD MULTIPLE HIGH) instruction 7-283
LMY (LOAD MULTIPLE) instruction 7-282
LNDBR (LOAD NEGATIVE) BFP instruction 19-34
LNDFR (LOAD NEGATIVE) floating-point-support instruction 9-34
LNDR (LOAD NEGATIVE) HFP instruction 18-16
LNEBR (LOAD NEGATIVE) BFP instruction 19-34
LNER (LOAD NEGATIVE) HFP instruction 18-16
LNGFR (LOAD NEGATIVE) binary instruction 7-284
LNGR (LOAD NEGATIVE) binary instruction 7-283
LNR (LOAD NEGATIVE) binary instruction 7-283
LNXBR (LOAD NEGATIVE) BFP instruction 19-34
LNXR (LOAD NEGATIVE) HFP instruction 18-16
LOAD ACCESS MULTIPLE instructions 7-265
LOAD ADDRESS EXTENDED instruction 7-266
LOAD ADDRESS instructions 7-266
  examples A-23
LOAD ADDRESS RELATIVE LONG instruction 7-267
LOAD ADDRESS SPACE PARAMETERS instruction 10-41
LOAD AND ADD instructions 7-268
LOAD AND ADD LOGICAL instructions 7-268
LOAD AND AND instructions 7-269
LOAD AND EXCLUSIVE OR instructions 7-269
LOAD AND OR instructions 7-270
LOAD AND TEST BFP instructions 19-31
LOAD AND TEST binary instruction 7-270
LOAD AND TEST DFP instructions 20-41
LOAD AND TEST HFP instructions 18-13
LOAD AND TRAP binary instructions 7-271
load-and-trap facility 1-17
  facility indication 4-100
LOAD AND ZERO RIGHTMOST BYTE binary instructions 7-271
load-and-zero-rightmost-byte facility 1-17
  facility indication 4-100
LOAD binary instructions 7-264
  example A-23
LOAD BYTE HIGH instruction 7-272
LOAD BYTE instructions 7-272
load-clear key 12-4
load-clear-list-directed key 12-4
LOAD COMPLEMENT BFP instructions 19-31
LOAD COMPLEMENT binary instruction 7-272
LOAD COMPLEMENT floating-point-support instruction 9-31
LOAD COMPLEMENT HFP instructions 18-14
LOAD CONTROL instructions 10-50
LOAD COUNT TO BLOCK BOUNDARY binary instruction 7-273
LOAD floating-point-support instructions 9-31
LOAD FP INTEGER BFP instructions 19-32
LOAD FP INTEGER DFP instructions 20-42
LOAD FP INTEGER HFP instructions 18-15
LOAD FPC AND SIGNAL floating-point-support instruction 9-32
LOAD FPC floating-point-support instruction 9-31
LOAD FPR FROM GR floating-point-support instruction 9-34
LOAD GR FROM FPR floating-point-support instruction 9-34
LOAD GUARDED instruction 7-274
LOAD GUARDED STORAGE CONTROLS instruction 7-275
LOAD HALFWORD binary 7-276
LOAD HALFWORD HIGH IMMEDIATE ON CONDITION binary instruction 7-277
LOAD HALFWORD HIGH instruction 7-277
LOAD HALFWORD IMMEDIATE instruction 7-276
LOAD HALFWORD IMMEDIATE ON CONDITION binary instruction 7-277
LOAD HALFWORD instruction
  examples A-24
LOAD HALFWORD RELATIVE LONG instructions 7-276
LOAD HIGH AND TRAP instruction 7-278
LOAD HIGH instruction 7-278
LOAD HIGH ON CONDITION instructions 7-284
LOAD IMMEDIATE instruction 7-264
load indicator 12-3
LOAD LENGTHENED BFP instructions 19-33
LOAD LENGTHENED DFP instructions 20-45
LOAD LENGTHENED HFP instructions 18-15
LOAD LOGICAL AND SHIFT GUARDED instruction 7-274
LOAD LOGICAL AND TRAP instruction 7-279
LOAD LOGICAL AND ZERO RIGHTMOST BYTE binary instruction 7-279
LOAD LOGICAL CHARACTER HIGH instruction 7-280
LOAD LOGICAL CHARACTER instructions 7-279
LOAD LOGICAL HALFWORD HIGH instruction 7-281

LOAD LOGICAL HALFWORD instructions 7-280
LOAD LOGICAL HALFWORD RELATIVE LONG instructions 7-280
LOAD LOGICAL IMMEDIATE instructions 7-281
LOAD LOGICAL instructions 7-278
LOAD LOGICAL RELATIVE LONG binary instruction 7-278
LOAD LOGICAL THIRTY ONE BITS AND TRAP instruction 7-282
LOAD LOGICAL THIRTY ONE BITS instructions 7-281
LOAD MULTIPLE DISJOINT instruction 7-283
LOAD MULTIPLE HIGH instruction 7-283
LOAD MULTIPLE instructions 7-282
LOAD NEGATIVE BFP instructions 19-34
LOAD NEGATIVE binary instruction 7-283
LOAD NEGATIVE floating-point-support instruction 9-34
LOAD NEGATIVE HFP instructions 18-16
load-normal key 12-4
LOAD ON CONDITION instructions 7-284
LOAD PAGE-TABLE-ENTRY ADDRESS instruction 10-50
LOAD PAIR DISJOINT instructions 7-285
LOAD PAIR FROM QUADWORD instruction 7-286
LOAD POSITIVE BFP instructions 19-35
LOAD POSITIVE binary instruction 7-287
LOAD POSITIVE floating-point-support instruction 9-34
LOAD POSITIVE HFP instructions 18-16
LOAD PSW EXTENDED instruction 10-55
LOAD PSW instruction 10-54
LOAD REAL ADDRESS instructions 10-56
LOAD RELATIVE LONG instruction 7-264
LOAD REVERSED instructions 7-287
LOAD ROUNDED BFP instructions 19-35
LOAD ROUNDED DFP instructions 20-46
LOAD ROUNDED HFP instructions 18-17
load state 4-2, 4-3
  during CCW-type IPL 4-81
load/store-on-condition facility 1 1-18
  facility indication 4-100
load-unit-address controls 12-4
LOAD USING REAL ADDRESS instruction 10-60
load-with-dump key 12-4
LOAD ZERO floating-point-support instructions 9-35
load/store-on-condition facility 2 1-18
loading, initial
  See IML, IPL
LOC (LOAD ON CONDITION) instruction 7-284
local-TLB-clearing facility 1-18
  facility indication 4-100
location 3-2
  not available in configuration 6-20
  See also address
LOCFH (LOAD HIGH ON CONDITION) binary instruction 7-284
LOCFHR (LOAD HIGH ON CONDITION) binary instruction 7-284
LOCG (LOAD ON CONDITION) instruction 7-284
LOCGHI (LOAD HALFWORD HIGH IMMEDIATE ON CONDITION) binary instruction 7-277
LOCGR (LOAD ON CONDITION) instruction 7-284
LOCHHI (LOAD HALFWORD HIGH IMMEDIATE ON CONDITION) binary instruction 7-277
LOCHI (LOAD HALFWORD IMMEDIATE ON CONDITION) binary instruction 7-277
lock A-47
  example with FIFO queuing A-49
  example with LIFO queuing A-48
lock used by PERFORM LOCKED OPERATION instruction 7-341
LOCR (LOAD ON CONDITION) instruction 7-284
logical
  arithmetic (unsigned binary) 7-6
  comparison 7-7
  connective
    AND 7-34, 7-35
    EXCLUSIVE OR 7-255, 7-256
    OR 7-311, 7-312
  data 7-3
logical address 3-5
  handling by DAT 3-40
logical-path mask
  See LPM
  I/O-interruption
    See I/O-interruption subclass mask
logical string assist 1-28
logically partitioned (LPAR) mode 1-32, 1-34
logout
  fixed
    assigned storage locations for 3-82
    machine-check 11-27
  subchannel (I/O) 16-47
long binary-floating-point number 19-2
long decimal-floating-point number 20-3
long-displacement facility 1-18
  facility indication 4-99
long hexadecimal-floating-point number 18-2
long I/O block 16-24, 16-39
loop control 5-14
loop of interruptions
  See string of interruptions
low-address protection 3-14
  control bit 3-14
  exception for 6-34
LPAR (logically partitioned) mode 1-32, 1-34
LPD (LOAD PAIR DISJOINT instructions 7-285

LPDBR (LOAD POSITIVE) BFP instruction 19-35
LPDFR (LOAD POSITIVE) floating-point-support instruction 9-34
LPDG (LOAD PAIR DISJOINT instructions 7-285
LPDR (LOAD POSITIVE) HFP instruction 18-16
LPEBR (LOAD POSITIVE) BFP instruction 19-35
LPER (LOAD POSITIVE) HFP instruction 18-16
LPGFR (LOAD POSITIVE) binary instruction 7-287
LPGR (LOAD POSITIVE) binary instruction 7-287
LPM (logical-path mask) 15-5, 15-28, 15-30
  effect on system performance of 15-11
  used for IPL 17-17
LPQ (LOAD PAIR FROM QUADWORD) instruction 7-286
LPR (LOAD POSITIVE) binary instruction 7-287
LPSW (LOAD PSW) instruction 10-54
LPSWE (LOAD PSW EXTENDED) instruction 10-55
LPTEA (LOAD PAGE-TABLE-ENTRY ADDRESS) instruction 10-50
LPUM (last-path-used mask) 15-6
  in ESW 16-49
  field-validity flag for (in subchannel logout) 16-49
LPXBR (LOAD POSITIVE) BFP instruction 19-35
LPXR (LOAD POSITIVE) HFP instruction 18-16
LR (LOAD) binary instruction 7-264
LRA (LOAD REAL ADDRESS) instruction 10-56
LRAG (LOAD REAL ADDRESS) instruction 10-56
LRAY (LOAD REAL ADDRESS) instruction 10-56
LRDR (LOAD ROUNDED) HFP instruction 18-17
LRER (LOAD ROUNDED) HFP instruction 18-17
LRL (LOAD RELATIVE LONG) binary instruction 7-264
LRV (LOAD REVERSED) instruction 7-287
LRVG (LOAD REVERSED) instruction 7-287
LRVGR (LOAD REVERSED) instruction 7-287
LRVH (LOAD REVERSED) instruction 7-287
LRVR (LOAD REVERSED) instruction 7-287
LST (linkage second table) 5-36
LSTO (linkage-second-table origin) 5-36
LSX (linkage first index)
  translation exception 6-29
LSX (linkage second index)
  invalid bit 5-36
LT (linkage table) 5-36
LT (LOAD AND TEST) instruction 7-271
LTD (linkage-table designation) 5-35
LTDBR (LOAD AND TEST) BFP instruction 19-31
LTDR (LOAD AND TEST) HFP instruction 18-13
LTDTR (LOAD AND TEST) DFP instruction 20-41
LTEBR (LOAD AND TEST) BFP instruction 19-31
LTER (LOAD AND TEST) HFP instruction 18-13
LTG (LOAD AND TEST) instruction 7-271
LTGF (LOAD AND TEST) binary instruction 7-271
LTGFR (LOAD AND TEST) binary instruction 7-270
LTGR (LOAD AND TEST) binary instruction 7-270
LTL (linkage-table length) 5-35
LTO (linkage-table origin) 5-35
LTR (LOAD AND TEST) binary instruction 7-270
LTXBR (LOAD AND TEST) BFP instruction 19-31
LTXR (LOAD AND TEST) HFP instruction 18-14
LTXTR (LOAD AND TEST) DFP instruction 20-41
LURA (LOAD USING REAL ADDRESS) instruction 10-60
LURAG (LOAD USING REAL ADDRESS) instruction 10-60
LUV (left-units view) of floating-point numbers 9-2
LX (linkage index) 5-33
  invalid bit 5-36
  translation exception 6-30
LXD (LOAD LENGTHENED) HFP instruction 18-15
LXDB (LOAD LENGTHENED) BFP instruction 19-34
LXDBR (LOAD LENGTHENED) BFP instruction 19-33
LXDR (LOAD LENGTHENED) HFP instruction 18-15
LXDTR (LOAD LENGTHENED) DFP instruction 20-45
LXE (LOAD LENGTHENED) HFP instruction 18-15
LXEB (LOAD LENGTHENED) BFP instruction 19-34
LXEBR (LOAD LENGTHENED) BFP instruction 19-33
LXER (LOAD LENGTHENED) HFP instruction 18-15
LXR (LOAD) floating-point-support instruction 9-31
LY (LOAD) binary instruction 7-264
LZDR (LOAD ZERO) floating-point-support instruction 9-35
LZER (LOAD ZERO) floating-point-support instruction 9-35
LZRF (LOAD AND ZERO RIGHTMOST BYTE) binary instruction 7-271
LZRG (LOAD AND ZERO RIGHTMOST BYTE) binary instruction 7-271
LZXR (LOAD ZERO) floating-point-support instruction 9-35

M

M (mega) xxviii
M (MULTIPLY) binary instruction 7-304, 7-305
  example A-28
machine check
  See also malfunction
  handling of malfunction detected as part of I/O 11-5
  interruption 6-16, 11-10
    action 11-11
    code (MCIC) 3-80, 3-86, 11-14
    floating conditions 11-26
    machine check interruption 11-26

mask in the PSW 4-6
subclass masks in control register 11-26
logout 11-27
mask
in the PSW 4-6
machine-check architectural-mode identification 3-76, 3-85
machine-check extended save area 11-24
machine-type number (in STORE CPU ID result) 10-136, 10-137
MAD (MULTIPLY AND ADD) HFP instruction 18-19
MADB (MULTIPLY AND ADD) BFP instruction 19-38
MADBR (MULTIPLY AND ADD) BFP instruction 19-38
MADR (MULTIPLY AND ADD) HFP instruction 18-19
MAE (MULTIPLY AND ADD) HFP instruction 18-19
MAEB (MULTIPLY AND ADD) BFP instruction 19-38
MAEBR (MULTIPLY AND ADD) BFP instruction 19-38
MAER (MULTIPLY AND ADD) HFP instruction 18-19
main storage 3-2
See also storage
effect of power-on reset on 4-80
See also storage
shared (in multiprocessing) 4-83
malfunction 11-2
at channel subsystem 16-29
at I/O device 16-29
correction of 11-2
effect on manual operation 12-1
from DIAGNOSE instruction 10-23
indication of 11-5
machine-check handling for when detected as part of I/O 11-5
malfunction alert (external interruption) 6-13
when entering check-stop state 11-10
manual indicator 12-4
See also stopped state
manual operation 12-1
controls
address-compare 12-1
alter-and-display 12-2
IML 12-3
load-unit-address 12-4
power 12-4
rate 12-5
TOD-clock 12-6
effect on CPU signaling 4-94
keys
interrupt 12-3
load-clear 12-4
load-clear-list-directed 12-4
load-normal 12-4
load-with-dump 12-4
restart 12-5
start 12-5
stop 12-5
store-status 12-5
system-reset-clear 12-6
system-reset-normal 12-6
masks 6-6
See also I/O interruption, interruption
for IEEE exceptions 9-10
in BRANCH INDIRECT ON CONDITION instruction 7-39
in BRANCH ON CONDITION instruction 7-40
in BRANCH RELATIVE ON CONDITION instruction 7-47
in COMPARE LOGICAL CHARACTERS UNDER MASK instruction 7-157
in INSERT CHARACTERS UNDER MASK instruction 7-263
in PSW 4-5
in STORE CHARACTERS UNDER MASK instruction 7-378
in TEST UNDER MASK instruction 7-393
monitor 6-30
path-management 15-2, 15-28, 15-30
PER-event 4-27
program-interruption 6-16
subclass
See subclass-mask bits
maximum negative number 7-4
MBA (measurement-block address) 15-9
MC (MONITOR CALL) instruction 7-288
MCIC (machine-check-interruption code) 3-80, 3-86, 11-14
MD (MULTIPLY) HFP instruction 18-18
MDB (MULTIPLY) BFP instruction 19-37
MDBR (MULTIPLY) BFP instruction 19-37
MDE (MULTIPLY) HFP instruction 18-18
MDEB (MULTIPLY) BFP instruction 19-37
MDEBR (MULTIPLY) BFP instruction 19-37
MDER (MULTIPLY) HFP instruction 18-17
MDR (MULTIPLY) HFP instruction 18-17
example A-43
MDTR (MULTIPLY) DFP instruction 20-47
MDTRA (MULTIPLY) DFP instruction 20-48
ME (MULTIPLY) HFP instruction 18-18
measurement
block (I/O)
address 17-7
format 17-7
origin 17-7
device-connect-time 17-10
extended-measurement-word 17-11
measurement-block update (I/O) 17-2
measurement-alert external interruption 6-13

measurement block (I/O) 17-3
data check 16-48
index 15-7
key (MBK)
used as access key 3-11
multiple use of 15-11
program check 16-48
protection check 16-48
update enable 15-3
measurement-block-format control (I/O) 15-8
measurement data (I/O)
accumulated 17-3
effect of CSCH on 14-5
effect of HSCH on 14-7
measurement-mode control (I/O) 15-3
MEE (MULTIPLY) HFP instruction 18-18
MEEB (MULTIPLY) BFP instruction 19-37
MEEBR (MULTIPLY) BFP instruction 19-37
MEER (MULTIPLY) HFP instruction 18-17
MER (MULTIPLY) HFP instruction 18-18
message byte (in EDIT) 8-8
message-security assist 1-18
facility indication 4-99
message-security-assist extension 1 1-19
message-security-assist extension 2 1-19
message-security-assist extension 3 1-19
facility indication 4-100
message-security-assist extension 4 1-20
facility indication 4-100
message-security-assist extension 5 1-21
facility indication 4-100
message-security-assist extension 6 1-21
message-security-assist extension 7 1-21
message-security-assist extension 8 1-22
facility indication 4-101
MFY (MULTIPLY) binary instruction 7-304
MG (MULTIPLY) binary instruction 7-304
MGH (MULTIPLY HALFWORD) binary instruction 7-305
MGHI (MULTIPLY HALFWORD IMMEDIATE) instruction 7-305
MGRK (MULTIPLY) binary instruction 7-304
MH (MULTIPLY HALFWORD) instruction 7-305
example A-29
MHI (MULTIPLY HALFWORD IMMEDIATE) instruction 7-305
MHY (MULTIPLY HALFWORD) binary instruction 7-305
MIDA (modified-indirect-data addressing) 15-68
MIDA flag in CCW 15-33
invalid 16-27
MIDAW (modified indirect-data-address word)
invalid address specification of 16-26
MIDAW (modified-indirect-data-address word) 15-68
check (in subchannel logout) 16-48
MIDAW (modified indirect-data-address word)
invalid address of 16-25
MIDAW control 15-28
miscellaneous-instruction-extensions facility 1 1-22
facility indication 4-100
miscellaneous-instruction-extensions facility 2 1-22
facility indication 4-100
ML (MULTIPLY LOGICAL) instruction 7-306
MLG (MULTIPLY LOGICAL) instruction 7-306
MLGR (MULTIPLY LOGICAL) instruction 7-306
MLR (MULTIPLY LOGICAL) instruction 7-306
modal instructions 1-5, 7-8
mode
access-register 3-40
addressing
See addressing mode
architectural
See architectural mode
burst (channel-path operation) 13-3
byte-multiplex mode (channel-path operation) 13-3
home-space 3-40
incorrect-length-indication 15-28
incorrect-length-suppression 15-28
indicator
architectural 12-2, 12-3
multipath
See multipath mode
primary-space 3-40
real 3-40
requirements for semiprivileged instructions 5-28
secondary-space 3-40
single-path 15-4, 15-23
translation 3-40
mode-trace-control bit 4-15
modifiable area (in linkage-stack state entry) 5-83
modification control 15-26
modified-CCW-indirect-data-addressing control 15-28
modified-CCW-indirect-data-addressing facility 1-22
MODIFY STACKED STATE instruction 10-61
MODIFY SUBCHANNEL instruction 14-7
MONITOR CALL instruction 7-288
monitor-class number 6-30, 7-288
assigned storage locations for 3-74, 3-85
monitor code 6-31, 7-288
assigned storage locations for 3-79, 3-85
monitor event 6-30, 7-288
counting operation 7-288
program interruption 7-288
monitor-event
counting operation 5-109
monitor masks 6-30, 7-288

monitoring
See also measurement
channel-subsystem 17-1
for PER events
See PER
with MONITOR CALL 6-30, 7-288
MOVE instructions 7-289
examples A-21, A-24
move-inverse facility 7-290
MOVE INVERSE instruction 7-290
example A-25
MOVE LONG EXTENDED instruction 7-294
MOVE LONG instruction 7-290
examples A-26
MOVE LONG UNICODE instruction 7-297
MOVE NUMERICS instruction 7-301
example A-26
MOVE PAGE instruction 10-62, 10-77
MOVE STRING instruction 7-301
example A-27
MOVE TO PRIMARY instruction 10-64
MOVE TO SECONDARY instruction 10-64
MOVE WITH DESTINATION KEY instruction 10-66
MOVE WITH KEY instruction 10-67
MOVE WITH OFFSET instruction 7-302
example A-27
move-with-optional-specifications facility 1-23
facility indication 4-99
MOVE WITH OPTIONAL SPECIFICATIONS instruction 10-68
MOVE WITH SOURCE KEY instruction 10-71
MOVE ZONES instruction 7-303
example A-28
MP (MULTIPLY DECIMAL) instruction 8-12
example A-39
MR (MULTIPLY) binary instruction 7-304
example A-28
MS (MULTIPLY SINGLE) instruction 7-307
MSC (MULTIPLY SINGLE) instruction 7-307
MSCH (MODIFY SUBCHANNEL) instruction 14-7
MSD (MULTIPLY AND SUBTRACT) HFP instruction 18-19
MSDB (MULTIPLY AND SUBTRACT) BFP instruction 19-38
MSDBR (MULTIPLY AND SUBTRACT) BFP instruction 19-38
MSDR (MULTIPLY AND SUBTRACT) HFP instruction 18-19
MSE (MULTIPLY AND SUBTRACT) HFP instruction 18-19
MSEB (MULTIPLY AND SUBTRACT) BFP instruction 19-38
MSEBR (MULTIPLY AND SUBTRACT) BFP instruction 19-38
MSER (MULTIPLY AND SUBTRACT) HFP instruction 18-19
MSFI (MULTIPLY SINGLE IMMEDIATE) instruction 7-307
MSG (MULTIPLY SINGLE) instruction 7-307
MSGC (MULTIPLY SINGLE) instruction 7-307
MSGF (MULTIPLY SINGLE) instruction 7-307
MSGFI (MULTIPLY SINGLE IMMEDIATE) instruction 7-307
MSGFR (MULTIPLY SINGLE) instruction 7-307
MSGR (MULTIPLY SINGLE) instruction 7-307
MSGRKC (MULTIPLY SINGLE) instruction 7-307
MSR (MULTIPLY SINGLE) instruction 7-307
MSRKC (MULTIPLY SINGLE) instruction 7-307
MSS (multiple subchannel set facility) 17-30
MSS (multiple-subchannel-set) facility 1-23
MSTA (MODIFY STACKED STATE) instruction 10-61
MSY (MULTIPLY SINGLE) instruction 7-307
multipath mode 15-4
entering 15-23
multiple-access storage references 5-125
multiple-epoch facility 1-23
multiple-subchannel-set facility 1-23
multiple subchannel set facility (MSS) 17-30
MULTIPLY AND ADD BFP instructions 19-38
MULTIPLY AND ADD HFP instructions 18-19
MULTIPLY AND SUBTRACT BFP instructions 19-38
MULTIPLY AND SUBTRACT HFP instructions 18-19
MULTIPLY BFP instructions 19-37
MULTIPLY binary instructions 7-304
examples A-28
MULTIPLY DECIMAL instruction 8-12
example A-39
MULTIPLY DFP instructions 20-47
MULTIPLY HALFWORD IMMEDIATE instruction 7-305
MULTIPLY HALFWORD instruction 7-305
example A-29
MULTIPLY HFP instructions 18-17
example A-43
MULTIPLY LOGICAL instructions 7-306
MULTIPLY SINGLE IMMEDIATE instructions 7-307
MULTIPLY SINGLE instructions 7-307
multiprocessing 4-82
manual operations for 12-6
programming considerations for 8-4, A-45
programming examples A-45
timing-facility interruptions for 4-52
TOD clock for 4-46
multiprogramming examples A-45
multithreading
core 2-3
disablement of 4-77, 4-78, 4-79

enablement of 4-91
facility description 1-23
operator considerations 12-7
terminology 2-3
thread 2-3
MVC (MOVE) instruction 7-289
examples A-21, A-24
MVCDK (MOVE WITH DESTINATION KEY) instruction 10-66
MVCIN (MOVE INVERSE) instruction 7-290
example A-25
MVCK (MOVE WITH KEY) instruction 10-67
MVCL (MOVE LONG) instruction 7-290
examples A-26
MVCLE (MOVE LONG EXTENDED) instruction 7-294
MVCLU (MOVE LONG UNICODE) instruction 7-297
MVCOS (MOVE WITH OPTIONAL SPECIFICATIONS) instruction 10-68
MVCP (MOVE TO PRIMARY) instruction 10-64
MVCS (MOVE TO SECONDARY) instruction 10-64
MVCSK (MOVE WITH SOURCE KEY) instruction 10-71
MVGHI (MOVE) instruction 7-289
MVHHI (MOVE) instruction 7-289
MVHI (MOVE) instruction 7-289
MVI (MOVE) instruction 7-289
example A-25
MVIY (MOVE) instruction 7-289
MVN (MOVE NUMERICS) instruction 7-301
example A-26
MVO (MOVE WITH OFFSET) instruction 7-302
example A-27
MVPG (MOVE PAGE) instruction 10-62, 10-77
MVST (MOVE STRING) instruction 7-301
example A-27
MVZ (MOVE ZONES) instruction 7-303
example A-28
MXBR (MULTIPLY) BFP instruction 19-37
MXD (MULTIPLY) HFP instruction 18-18
MXDB (MULTIPLY) BFP instruction 19-37
MXDBR (MULTIPLY) BFP instruction 19-37
MXDR (MULTIPLY) HFP instruction 18-17
MXR (MULTIPLY) HFP instruction 18-17
MXTR (MULTIPLY) DFP instruction 20-47
MXTRA (MULTIPLY) DFP instruction 20-48

N

N (AND) instruction 7-33
N condition (I/O) 16-13
NaN (not-a-number) 9-2, 19-4, 20-7
NC (AND) instruction 7-34
NDNR (AND) binary instruction 7-33
NDXGR (EXCLUSIVE OR) binary instruction 7-254
near-valid CBC 11-2
in storage 11-6
negative zero
binary 7-4
decimal 8-3
example A-5
nested transaction
transactional execution 5-90, 5-91
nesting 5-90, 5-91
new PSW 4-4
assigned storage locations for 3-81
assigned storage locations for (in the z/Architecture architectural mode) 3-83
fetched during interruption 6-2
next-entry size (in linkage stack) 5-80
NG (AND) instruction 7-33
NGR (AND) instruction 7-33
NGRK (AND) binary instruction 7-33
NI (AND) instruction 7-33
example A-8
NIHF (AND IMMEDIATE) instruction 7-34
NIHH (AND IMMEDIATE) instruction 7-34
NIHL (AND IMMEDIATE) instruction 7-34
NILF (AND IMMEDIATE) instruction 7-35
NILH (AND IMMEDIATE) instruction 7-35
NILL (AND IMMEDIATE) instruction 7-35
NIY (AND) instruction 7-34
no-operation
instruction (BRANCH ON CONDITION) 7-40
instruction (BRANCH RELATIVE ON CONDITION) 7-47
node (of tree structure) 7-419
nonconstrained transaction
transactional execution 5-91
nonconstrained transaction-execution mode
transactional execution 5-91
noninterlocked-update storage reference 5-124
nonnumeric entities
decimal 20-7
IEEE 9-2
nonquiescing key-setting facility
facility indication 4-99
NQ control
in SET STORAGE KEY EXTENDED 10-130
storage-key accesses 5-120
1-23
NONTRANSACTION STORE instruction 7-310
nontrap action
IEEE 9-18
nonvolatile storage 3-2
normal numbers
class of BFP data 19-4
class of DFP data 20-7

normalization
of HFP numbers 18-2
not-a-number (NaN) 9-2, 19-4, 20-7
not operational
as channel-path state 16-13
See path-not-operational bit
as CPU state 4-94
as TOD-clock state 4-48
not set (TOD-clock state) 4-47
NR (AND) instruction 7-33
NTSTG (NONTRANSACTION STORE) instruction 7-310
nullification
exceptions to 5-26
for exigent machine-check conditions 11-10
of instruction execution 5-23
of unit of operation 5-25
numbering
of addresses (byte locations) 3-2
of bits 3-3
numbers
binary 7-4
examples A-2
binary-floating-point 19-2
configuration identification (in STORE CPU ID result) 10-136, 10-137
decimal 8-1
examples A-4
device 13-6
hexadecimal 5-7
hexadecimal-floating-point 18-2
examples A-5
machine-type number (in STORE CPU ID result) 10-136, 10-137
numeric bits 8-1
moving of 7-301
Numeric part of PC number 5-83
NY (AND) instruction 7-33

O

O (OR) instruction 7-310
OC (OR) instruction 7-311
octoword 3-3
OEMI (original equipment manufacturers information) for I/O interface
publication referenced xxix
offset of symbol-translation table 7-173
OG (OR) instruction 7-311
OGR (OR) instruction 7-310
OGRK (OR) binary instruction 7-310
OI (OR) instruction 7-311
example A-29
example of problem with A-45
OIHF (OR IMMEDIATE) instruction 7-312
OIHH (OR IMMEDIATE) instruction 7-312
OIHL (OR IMMEDIATE) instruction 7-312
OILF (OR IMMEDIATE) instruction 7-312
OILH (OR IMMEDIATE) instruction 7-312
OILL (OR IMMEDIATE) instruction 7-312
OIY (OR) instruction 7-311
old PSW 6-2
assigned storage locations for 3-80
assigned storage locations for (in the ESA/390-compatibility mode) 3-83
assigned storage locations for (in the z/Architecture architectural mode) 3-83
one's complement binary notation 7-4
used for SUBTRACT LOGICAL instruction 7-390
used for SUBTRACT LOGICAL WITH BORROW instruction 7-391
op code
See operation code
operand 5-3
access identification 3-75
access of 5-123
for I/O instructions 14-2
address generation for 5-11
exception 6-31
immediate 5-8
length of 5-3
overlap of
for decimal instructions 8-3
for general instructions 7-4
register for 5-7
sequence of references for 5-123
storage 5-9
types of (fetch, store, update) 5-123
used for result 5-3
operating state 4-2, 4-3
operation
I/O
See I/O operations
unit of 5-24
operation code (op code) 5-3
invalid 6-31
operation exception 6-31
operation-request block
See ORB
operator facilities 2-8, 12-1
basic 12-1
operator intervening (signal-processor status) 4-96
OR (OR) instruction 7-310
OR IMMEDIATE instructions 7-312
OR instructions 7-310
example of problem with OR immediate A-45
examples A-29
ORB (operation-request block) 15-24

channel-program-type control 15-27
CSS priority 15-29
designating a CCW channel program 15-25
extension control in 15-28, 15-30
interruption parameter in 15-25
invalid 16-27
logical-path mask (LPM) in 15-28, 15-30
order-preserving-compression facility 1-24
facility indication 4-99
orders (I/O) 13-8
orders (signal-processor) 4-84
conditional emergency signal 4-90
conditions precluding response to 4-94
CPU reset 4-85
emergency signal 4-85
emergency signal, conditional 4-90
external call 4-85
initial CPU reset 4-85
restart 4-85
sense 4-85
set architecture 4-89, 4-92
set prefix 4-85
start 4-85
stop 4-85
stop and store status 4-85
store status at address 4-86
set multithreading 4-91
ORK (OR) binary instruction 7-310
outer transaction 5-91
outermost transaction 5-91
overflow
binary 7-5
example A-2
decimal 6-26
exponent
See exponent overflow
fixed-point 6-27, 7-5
in CRW 17-27
overlap
destructive 7-291, 7-296, 7-299
operand 5-116
for decimal instructions 8-3
for general instructions 7-4
operation 5-114
OY (OR) instruction 7-310

P

P (peta) xxviii
PACK ASCII instruction 7-313
PACK instruction 7-312
example A-29
PACK UNICODE instruction 7-314
packed decimal numbers 8-1
conversion of to zoned format 7-416
conversion to from zoned format 7-312
padding byte
for COMPARE LOGICAL LONG EXTENDED instruction 7-161
for COMPARE LOGICAL LONG instruction 7-158
for MOVE LONG EXTENDED instruction 7-294
for MOVE LONG instruction 7-290
page 3-39
page-frame real address (PFRA) 3-51
PAGE IN instruction 10-72
page index (PX) 3-39
page-invalid bit (in page-table entry) 3-51
PAGE OUT instruction 10-73
page protection
exception for 6-34
page swapping 3-38
page table 3-51
lookup 3-61
origin (PTO) 3-49
page-translation exception 6-32
as an access exception 6-47, 6-53
PALB (PURGE ALB) instruction 10-116
PAM (path-available mask) 15-7
effect of reconfiguration on 15-11
effect of resetting on 15-11
effect on allegiance of 15-12
parallel-I/O channel-to-channel adapter
publication referenced xxix
parallel-I/O interface 13-3
OEMI publication referenced xxix
parameter
external-interruption 6-11
assigned storage locations for 3-74, 3-84
I/O-interruption
See I/O-interruption parameter
register for SIGNAL PROCESSOR 4-85, 10-134
translation 3-40
parity bit 11-2
parsing-enhancement facility 1-24
facility indication 4-99
partial completion of instruction execution 5-24
PASCE (primary address-space-control element) 3-42
PASN (primary address-space number) 3-24
in trace entry 4-23
PASTE (primary AST entry) 5-35
PASTEIN (primary-AST-entry instance number) 3-26
PASTEO (primary-AST-entry origin) 5-35, 5-54
path
See channel path
path available for selection 15-13
path management 13-9
for clear function 15-14, 15-22

for halt function 15-16
for start function and resume function 15-20
path-management-control word
See PMCW
path-management masks
last-path-used mask
See LPUM
logical-path mask
See LPM
path-available mask
See PAM
path-installed mask
See PIM
path-not-operational mask
See PNOM
path-operational mask
See POM
path-not-operational bit (N) in SCSW 16-13
path-not-operational condition 15-5
path verification required
indicator for (in ERW) 16-52
pattern (in EDIT) 8-8
PC-cp (PROGRAM CALL instruction, to current primary) 10-95
PC number 10-92
in linkage-stack state entry 5-83
in trace entry 4-23
translation 5-33
PC-ss (PROGRAM CALL instruction, with space switching) 10-95
PC-translation-specification exception 6-32
PC-type bit 5-38
PCC (PERFORM CRYPTOGRAPHIC COMPUTATION) instruction 7-315
PCI (program-controlled interruption) 15-64
as flag in CCW 15-32
intermediate interruption condition for 16-18
subchannel status for 16-24
PCKMO (PERFORM CRYPTOGRAPHIC KEY MANAGEMENT OPERATION) instruction 10-74
pending channel reports
effect of I/O-system reset on 17-15
pending interruption
See interruption pending
PER (program-event recording)
access identification 3-75, 3-85, 4-32
address 4-31
assigned storage locations for 3-74, 3-85
address-space-control element (ASCE) identification 4-31
ASCE (address-space-control element) identification 4-31
ATMID (addressing-and-translation-mode identification) 4-30
code 4-30
assigned storage locations for 3-74, 3-85
events 4-26
extensions 1-31
instruction-fetching event 4-36
masks
bit in the PSW 4-5
PER-event 4-27
priority of indication 4-33
program-interruption condition 6-32
storage-alteration event 4-36
storage-area designation 4-34
ending address 4-28
starting address 4-28
wraparound 4-34
store-using-real-address event 4-37
successful-branching event 4-35
PER zero-address-detection facility 1-24
PER-3 facility 1-24
breaking-event-address recording 4-43
PER instruction-fetching nullification 4-26
PERFORM CRYPTOGRAPHIC COMPUTATION instruction 7-315
PERFORM CRYPTOGRAPHIC KEY MANAGEMENT OPERATION instruction 10-74
PERFORM FLOATING-POINT OPERATION floating-point-support instruction 9-35
PERFORM FRAME MANAGEMENT FUNCTION instruction 10-77
PERFORM LOCKED OPERATION instruction 7-331
example A-51
PERFORM PROCESSOR ASSIST instruction 7-345
PERFORM RANDOM NUMBER OPERATION instruction 7-346
PERFORM TIMING FACILITY FUNCTION instruction 10-81
PERFORM TOPOLOGY FUNCTION instruction 10-89
permissible set 9-13
PFD (PREFETCH DATA) instruction 7-359
PFDRL (PREFETCH DATA RELATIVE LONG) instruction 7-359
PFPO (PERFORM FLOATING-POINT OPERATION) floating-point-support instruction 9-35
PFPO facility 1-24
facility indication 4-99
PFRA (page-frame real address) 3-51
PGIN (PAGE IN) instruction 10-72
PGOUT (PAGE OUT) instruction 10-73
piecemeal steps of instruction execution 5-115
PIM (path-installed mask) 15-6
PKA (PACK ASCII) instruction 7-313
PKM (PSW-key mask) 5-28
PKU (PACK UNICODE) instruction 7-314

PLO (PERFORM LOCKED OPERATION) instruction 7-331
example A-51
PMCW (path-management-control word) 15-2
channel-path identifiers (CHPID) 15-8
PNOM (path-not-operational mask) 15-5
effect on POM of 15-11
indicated in SCSW 16-13
point of damage 11-14
point of interruption 5-24
for machine check 11-14
POM (path-operational mask) 15-7
effect on PNOM of 15-11
POPCNT (POPULATION COUNT) instruction 7-359
population-count facility 1-24
facility indication 4-100
POPULATION COUNT instruction 7-359
postnormalization 18-2
power controls 12-4
power-on reset 4-80
powers of 2
table of G-1
PPA (PERFORM PROCESSOR ASSIST) instruction 7-345
PPNO. *See* PRNO
PR (PROGRAM CALL instruction 10-90
PR (PROGRAM RETURN) instruction 10-103
PR-cp (PROGRAM RETURN instruction, to current primary) 10-104
PR-ss (PROGRAM RETURN instruction, with space switching) 10-104
PR/SM (Processor Resource/Systems Manager) 1-32, 1-34
precise intermediate value 9-12
precision (floating-point) 9-1
precision-rounded value 9-12
preferred quantum 20-2
preferred sign codes 8-2
prefetch control 15-27
PREFETCH DATA instruction 7-359
PREFETCH DATA RELATIVE LONG instruction 7-359
prefetching
access exceptions not recognized for 6-50
channel-control check during 16-29, 16-43
channel-data check during 16-28, 16-42, 16-43
handling of invalid CBC in storage during 11-8
of ART-table and DAT-table entries 5-119
of data for I/O 15-59
of instructions 5-118
of operands 5-123
See CCW prefetch control
prefix 3-21
set by signal-processor order 4-85
store-status save area for 3-82, 3-86
prefix area 3-21
prefix register 3-21
prenormalization 18-2
primary address space 3-23
primary ASN (PASN) 3-24
in linkage-stack state entry 5-82
primary-AST entry (PASTE) 5-35
origin (PASTEO) 5-35, 5-54
primary ASTEIN (PASTEIN)
in linkage-stack state entry 5-84
primary authority 3-36
exception 6-33
primary CPU 2-3, 4-81, 12-7
primary designation-type-control bits 3-44
primary interruption condition (I/O) 16-4
primary-list bit 5-54
primary private-space-control bit 3-43
primary real-space-control bit 3-43
primary real-space token origin (PRSTKO) 3-44
primary region table
designation (PRTD) 3-42
length (PRTL) 3-44
origin (PRTO) 3-42
primary segment table
designation (PSTD) 3-42
length (PSTL) 3-44
origin (PSTO) 3-42
primary-space access-list designation (PSALD) 5-56
primary-space mode 3-40
primary space-switch-event-control bit 3-43
primary-status bit (I/O) 16-19
primary storage-alteration-event-control bit 3-43
primary subspace-group-control bit 3-43
primary virtual address 3-5
effective address-space-control element for 3-52
priority
of access exceptions 6-53
of ASN-translation exceptions 6-56
of data exceptions 6-17
of external-interruption conditions 6-11
of I/O interruptions 16-5
of interruptions (CPU) 6-57
of PER events 4-33
of program-interruption conditions 6-50
of subspace-replacement exceptions 6-56
of trace exceptions 6-56
private bit 5-56
private-space control
effect on
fetch-protection override 3-13
low-address protection 3-14
use of common regions 3-47
use of common segments 3-50

private-space-control bit 3-43
home 3-45
primary 3-43
secondary 3-44
privileged instructions 4-6
control 10-1
privileged-operation exception 6-33
PRNO (PERFORM RANDOM NUMBER GENERATION) instruction 7-346
problem state 4-6
bit in entry-table entry 5-38
bit in the PSW 4-6
compatibility 1-35
processing backup (synchronous machine-check condition) 11-18
processing damage (synchronous machine-check condition) 11-19
processor
See CPU 2-2
processor-assist facility 1-24
facility indication 4-100
processor-availability facility 1-32
Processor Resource/Systems Manager (PR/SM) 1-32, 1-34
program 5-48
channel
See channel program
exceptions 6-16
execution of 5-1
fields of SCHIB modifiable by 15-10
initial loading by CCW-type IPL 17-17
initial loading by list-directed IPL 17-19
initial loading of 4-81, 17-16
interruption 6-16
priority of 6-17, 6-50
PROGRAM CALL instruction 10-90
trace entry for 4-23
type of 5-38
program-call state entry 5-81, 10-94
program check
as subchannel status 16-25, 16-40
measurement-block 16-48
program-controlled interruption (I/O)
See PCI
program-event recording
See PER
program events
See PER events
program interruption
filtering 5-103
transaction diagnostic block 3-83
program-interruption filtering
in transactional execution 5-91
program interruption identification
in the transaction diagnostic block 5-96
program mask
in the PSW 4-7, 6-16
inserted by INSERT PROGRAM MASK 7-264
preserved by BRANCH AND LINK 5-15, 7-36
set by SET PROGRAM MASK 7-371
validity bit for 11-21
PROGRAM RETURN instruction 10-103
program-status word
See PSW
PROGRAM TRANSFER instruction 10-107
trace entry for 4-23
PROGRAM TRANSFER WITH INSTANCE instruction 10-107
programmable field of TOD clock 4-51
protection (storage) 3-10
access-list-controlled
See access-list-controlled protection
DAT
See DAT protection
during tracing 4-25
fetch
See fetch protection
key-controlled
See key-controlled protection
low-address
See low-address protection
page
See page protection
protection check
as subchannel status 16-28, 16-42
measurement-block 16-48
protection exception 6-34
as an access exception 6-47, 6-53
PRSD (primary real-space designation) 3-42
PRSTKO (primary real-space token origin) 3-44
PRTD (primary region-table designation) 3-42
PRTL (primary region-table length) 3-44
PRTO (primary region-table origin) 3-42
PSALD (primary-space access-list designation) 5-56
pseudo AST entry 3-24
PSMF(PERFORM FRAME MANAGEMENT FUNCTION) instruction 10-77
PSTD (primary segment-table designation) 3-42
PSTL (primary segment-table length) 3-44
PSTO (primary segment-table origin) 3-42
PSW (program-status word) 2-3, 4-4
current 4-4, 5-14
stored during interruption 6-2
exceptions associated with 6-9
format error 6-9
in linkage-stack state entry 5-82
in program execution 5-14
saved 4-77

store-status save area for 3-82, 3-86
validity bits for 11-20
PSW key 4-6
control bit 5-75
in entry-table entry 5-76
in trace entry 4-23
used as access key 3-11
validity bit for 11-20
PSW-key mask (PKM) 5-28
control bit 5-75
in linkage-stack state entry 5-82
PT (PROGRAM TRANSFER) instruction 10-107
PT-cp (PROGRAM TRANSFER instruction, to current primary) 10-108
PT-ss or PTI-ss (PROGRAM TRANSFER (WITH INSTANCE) instruction, with space switching) 10-108
PTF (PERFORM TOPOLOGY FUNCTION) instruction 10-89
PTFF (PERFORM TIMING FACILITY FUNCTION) instruction 10-81
PTI (PROGRAM TRANSFER WITH INSTANCE) instruction 10-107
PTLB (PURGE TLB) instruction 10-116
PTO (page-table origin) 3-49
publications
other related documents xxix
PURGE ALB instruction 10-116
PURGE TLB instruction 10-116
PX (page index) 3-39

Q

QADTR (QUANTIZE) DFP instruction 20-49
QAXTR (QUANTIZE) DFP instruction 20-49
QNaN (quiet NaN) 9-2, 19-4, 20-7
quadword 3-3
quadword-concurrent storage references 5-126
QUANTIZE DFP instructions 20-49
quantum 20-2
quantum exception 9-21
nontrap action 9-22
trap action 9-22
queuing
FIFO
example for lock and unlock A-49
LIFO
example for lock and unlock A-48
quiescing 5-133
quiet NaN (QNaN) 9-2, 19-4, 20-7

R

R field of instruction 5-7
radix
binary 9-4
decimal 9-4
hexadecimal 9-3
rate control 12-5
RCHP (RESET CHANNEL PATH) instruction 14-9
real address 3-4
real mode 3-40
real space
token origin (RSTKO) 3-42
real-space-control bit
home 3-45
primary 3-43
secondary 3-44
real-space designation (RSD)
home 3-45
primary 3-42
secondary 3-44
real storage 3-5
receiver check (signal-processor status) 4-97
reconfiguration of I/O system 17-23
recovery
as class of machine-check condition 11-10
channel-subsystem 17-24
system 11-15
subclass-mask bit for 11-26
reduced-authority state 10-8
redundancy 11-2
reference
bit in storage key 3-9
multiple-access 5-125
recording 3-19
sequence for storage 5-113
See also sequence
single-access 5-125
region 3-39
region first index (RFX) 3-39
region-first-translation exception 6-35
as an access exception 6-53
region index (RX)
in virtual address 3-39
region-invalid bit (in region-table entry) 3-47
region second index (RSX) 3-39
region-second-translation exception 6-36
as an access exception 6-53
region table
origin (RTO) 3-42
region-table designation (RTD) 3-42
home 3-45
primary 3-42
secondary 3-44
region-table entry (RTE) 3-46
region third index (RTX) 3-39
region-third-translation exception 6-36
as an access exception 6-53

region-translation exception 6-35
register
access 2-6
base-address 2-4
control 2-6
designation of 5-7
floating-point 2-4, 9-8
floating-point-control 9-9
general 2-4
index 2-4
prefix 3-21
save areas for 3-82, 11-23
validation of 11-9
validity bits for 11-21, 11-22
vector 2-5
relative branching 5-13
remainder 19-5
result of DIVIDE TO INTEGER 19-28
remaining free space (in linkage stack) 5-79
remote operating stations 12-1
reporting-source code (RSC) 17-27
reporting-source ID (RSID) 17-28
repressible machine-check conditions 11-10
request-logging-only 16-52
indicator for (in ERW) 16-52
REROUND DFP instructions 20-52
reset 17-13
channel-path 17-13
clear 4-79
CPU 4-77
effect on CPU state 4-2
effect on TOD clock 4-47
I/O-system 17-13
as part of subsystem reset 4-79
initial CPU 4-78
power on 4-80
subsystem 4-79
summary of functions 4-75
summary of functions performed by manual initiation of 4-74
system-reset-clear key 12-6
system-reset-normal key 12-6
RESET CHANNEL PATH instruction 14-9
function initiated by 15-80
See channel-path-reset function
RESET REFERENCE BIT EXTENDED instruction 10-116
RESET REFERENCE BIT MULTIPLE instruction 10-117
reset-reference-bits-multiple facility 1-25
facility indication 4-100
reset signal (I/O) 17-12
in I/O-system reset 17-14
issued as part of RCHP 15-81
in channel-path reset 17-13
resolution
of clock comparator 4-59
of CPU timer 4-62
of TOD clock 4-46
restart
interruption 6-56
key 12-5
signal-processor order 4-85
result operand 5-3
resume function 13-11, 15-20
initiated by RESUME SUBCHANNEL 14-10
path management for 15-20
pending 16-14, 16-36
See also start function
RESUME PROGRAM instruction 10-117
RESUME SUBCHANNEL instruction 14-10
channel-program requirements for 14-11
count of in measurement block 17-4
function initiated by 15-20
See resume function
retry
CPU 11-2
I/O command
See command retry
reusable-ASN bit 3-25
RFX (region first index) 3-39
RI instruction format 5-6
RIE instruction format 5-6
right-units view (RUV) of floating-point numbers 9-2
RIL instruction format 5-6
RIS instruction format 5-6
RISBG (ROTATE THEN INSERT SELECTED BITS) instruction 7-363
RISBG N(ROTATE THEN INSERT SELECTED BITS) instruction 7-363
RISBHG (ROTATE THEN INSERT SELECTED BITS HIGH) instruction 7-364
RISBLG (ROTATE THEN INSERT SELECTED BITS LOW) instruction 7-365
RLL (ROTATE LEFT SINGLE LOGICAL) instruction 7-361
RLLG (ROTATE LEFT SINGLE LOGICAL) instruction 7-361
RNSBG (ROTATE THEN AND SELECTED BITS) instruction 7-361
ROTATE LEFT SINGLE LOGICAL instruction 7-361
ROTATE THEN AND SELECTED BITS instruction 7-361
ROTATE THEN EXCLUSIVE OR SELECTED BITS instruction 7-362
example A-30
ROTATE THEN INSERT SELECTED BITS HIGH instruction 7-364

ROTATE THEN INSERT SELECTED BITS
instruction 7-363
example A-30
ROTATE THEN INSERT SELECTED BITS LOW
instruction 7-365
ROTATE THEN OR SELECTED BITS instruction
7-362
example A-30
rounded intermediate value 9-12
rounding (decimal) 8-12
example A-40
rounding (floating-point)
IEEE 9-13
rounding action
summary of 9-17
rounding methods 9-14
RP (RESUME PROGRAM) instruction 10-117
RR instruction format 5-6
RRBE (RESET REFERENCE BIT EXTENDED)
instruction 10-116
RRBM (RESET REFERENCE BIT MULTIPLE)
instruction 10-117
RRD instruction format 5-6
RRDTR (REROUND) DFP instruction 20-52
RRE instruction format 5-6
RRF instruction format 5-6
RRS instruction format 5-6
RRXTR (REROUND) DFP instruction 20-52
RS instruction format 5-6
RSC (reporting-source code) 17-27
RSCH (RESUME SUBCHANNEL) instruction 14-10
RSI instruction format 5-6
RSID (reporting-source ID) 17-28
RSL instruction format 5-6
RSTKO (real-space token origin) 3-42
RSX (region second index) 3-39
RSY instruction format 5-6
RTE (region-table entry) 3-46
RTO (region-table origin) 3-42
RTX (region third index) 3-39
running (state of TOD clock) 4-47
RUV (right-units view) of floating-point numbers 9-2
RX (region index) 3-39
RX instruction format 5-6
RXE instruction format 5-6, 5-7
RXF instruction format 5-6, 5-7
RXSBG (ROTATE THEN EXCLUSIVE OR
SELECTED BITS) instruction 7-362
RXY instruction format 5-6, 5-7

S

S (SUBTRACT) binary instruction 7-388
S instruction format 5-6, 5-7
SAC (SET ADDRESS SPACE CONTROL) instruction
10-120
SACF (SET ADDRESS SPACE CONTROL FAST)
instruction 10-120
SAL (SET ADDRESS LIMIT) instruction 14-12
SAM24 (SET ADDRESSING MODE) instruction
7-370
SAM31 (SET ADDRESSING MODE) instruction
7-370
SAM64 (SET ADDRESSING MODE) instruction
7-370
sample count (in ESW) 17-4
SAR (SET ACCESS) instruction 7-370
SASCE (secondary address-space-control element)
3-44
SASN (secondary address-space number) 3-24
in trace entry 4-24
SASTEIN (secondary-AST-entry instance number)
3-26
save areas for registers 3-82, 11-23
saved PSW 4-77
scale factor (Ψ) 9-12
scaled preferred quantum 20-3
scaled value 9-12
scaling exponent
signed (Ω) 9-13
unsigned (α) 9-12
SCHIB (subchannel-information block) 15-2
as operand of
MODIFY SUBCHANNEL 14-7
STORE SUBCHANNEL 14-18
Measurement-Block Address (MBA) 15-9
model-dependent area in 15-9
path-management-control word (PMCW) in 15-2
subchannel-status word (SCSW) in 15-9
summary of modifiable fields in 15-10
SCHM (SET CHANNEL MONITOR) instruction
14-13
SCK (SET CLOCK) instruction 10-121
SCKC (SET CLOCK COMPARATOR) instruction
10-122
SCKPF (SET CLOCK PROGRAMMABLE FIELD)
instruction 10-123
SCP-initiated reset 1-31
SCSW (subchannel-status word) 16-7
activity-control field in 16-14, 16-36
CCW address in 16-19
count in 16-31, 16-44
device-status field in 16-24
function-control field in 16-13
in IRB 16-7
in SCHIB 15-9
status-control field in 16-17
subchannel-control field in 16-12, 16-35

subchannel-status field in 16-24, 16-39
SD (SUBTRACT NORMALIZED) HFP instruction 18-24
SDB (SUBTRACT) BFP instruction 19-40
SDBR (SUBTRACT) BFP instruction 19-40
SDR (SUBTRACT NORMALIZED) HFP instruction 18-24
SDTR (SUBTRACT) DFP instruction 20-55
SDTRA (SUBTRACT) DFP instruction 20-55
SE (SUBTRACT NORMALIZED) HFP instruction 18-24
SEARCH STRING instruction 7-366
examples A-31
SEARCH STRING UNICODE instruction 7-368
SEB (SUBTRACT) BFP instruction 19-40
SEBR (SUBTRACT) BFP instruction 19-40
secondary address space 3-23
secondary ASN (SASN) 3-24
control bit 5-76
in linkage-stack state entry 5-82
secondary ASTEIN (SASTEIN)
in linkage-stack state entry 5-83
secondary authority 3-36
exception 6-37
secondary-CCW address validity (in ERW) 16-52
secondary CPU 2-3
secondary designation-type-control bits 3-44
secondary error (in subchannel logout) 16-50
secondary interruption condition (I/O) 16-5
secondary private-space-control bit 3-44
secondary real-space-control bit 3-44
secondary real-space token origin (SRSTKO) 3-44
secondary region table
designation (SRTD) 3-44
length (SRTL) 3-44
origin (SRTO) 3-44
secondary segment table
designation (SSTD) 3-44
length (SSTL) 3-44
origin (SSTO) 3-44
secondary-space-control bit 3-41, 5-29
secondary-space mode 3-40
secondary-status bit (I/O) 16-19
secondary storage-alteration-event-control bit 3-44
secondary subspace-group-control bit 3-44
secondary virtual address 3-5
effective address-space-control element for 3-52
segment 3-39
segment-frame absolute address 3-49
segment index (SX) 3-39
segment-invalid bit (in segment-table entry) 3-50
segment table 3-49
length (STL) 3-42
origin (STO) 3-42
segment-table designation (STD)
home 3-45
obtaining of in access-register translation 5-47
primary 3-42
secondary 3-44
segment-translation exception 6-37
as an access exception 6-47, 6-53
self-describing block of I/O data 15-62
semiprivileged
instructions 4-6
descriptions of 10-2
program authorization 5-27
summary of 5-32
programs 4-6, 5-27
sense
as signal-processor order 4-85
sense-running-status facility
facility indication 4-99
sequence
conceptual 5-113
instruction-execution 5-3
of CCWs that is invalid 16-27
of storage references 5-113
ART-table and DAT-table entries 5-119
for floating-point data 9-8
instructions 5-118
operands 5-123
storage keys 5-120
sequence code (in subchannel logout) 16-50
field-validity flag 16-49
SER (SUBTRACT NORMALIZED) HFP instruction 18-24
serial-I/O channel-to-channel adapter
publication referenced xxix
serial-I/O interface 13-3
publication referenced xxix
serialization 5-130
caused by I/O Instructions 14-2
channel-program 5-133
CPU 5-130
effects of transactional execution 5-130
in completion of store operations 5-124
quiescing 5-133
specific operand 5-132
server-time-protocol facility 1-25
service-call-logical-processor (SCLP) facility 1-33
service-processor damage 11-17
service processor inoperative (signal-processor status) 4-97
service-signal external interruption 6-14
subclass-mask bit for 6-14
SET ACCESS instruction 7-370
SET ADDRESS LIMIT instruction 14-12

SET ADDRESS SPACE CONTROL FAST instruction 10-120
SET ADDRESS SPACE CONTROL instruction 10-120
SET ADDRESSING MODE instruction 7-370
SET ADDRESSING MODE instructions 7-370
set architecture
signal-processor order 4-89, 4-92
SET BFP ROUNDING MODE floating-point-support instructions 9-47
SET CHANNEL MONITOR instruction 14-13
effect on measurement modes of 17-1
SET CLOCK COMPARATOR instruction 10-122
SET CLOCK instruction 10-121
SET CLOCK PROGRAMMABLE FIELD instruction 10-123
SET CPU TIMER instruction 10-123
SET DFP ROUNDING MODE (SRNMT) floating-point-support instruction 9-47
SET FPC AND SIGNAL floating-point-support instruction 9-48
SET FPC floating-point-support instruction 9-47
set multithreading
signal-processor order 4-91
set prefix (signal-processor order) 4-85
SET PREFIX instruction 10-123
SET PROGRAM MASK instruction 7-371
set-program-parameters facility
facility indication 4-99
SET PSW KEY FROM ADDRESS instruction 10-124
SET SECONDARY ASN instruction 10-125
access registers 5-51
SET SECONDARY ASN WITH INSTANCE instruction 10-125
set state (of TOD clock) 4-48
SET STORAGE KEY EXTENDED instruction 10-130
SET SYSTEM MASK instruction 10-133
SFAA (segment-frame absolute address) 3-49
SFASR (SET FPC AND SIGNAL) floating-point-support instruction 9-48
SFPC (SET FPC) floating-point-support instruction 9-47
SG (SUBTRACT) binary instruction 7-388
SGF (SUBTRACT) binary instruction 7-388
SGFR (SUBTRACT) binary instruction 7-387
SGH (SUBTRACT HALFWORD) instruction 7-388
SGR (SUBTRACT) binary instruction 7-387
SGRK (SUBTRACT) binary instruction 7-388
SH (SUBTRACT HALFWORD) instruction 7-388
shared storage
See storage sharing
shared TOD clock 4-46
SHHHR (SUBTRACT HIGH) instruction 7-389
SHHLR (SUBTRACT HIGH) instruction 7-389
SHIFT AND ROUND DECIMAL instruction 8-12
examples A-39
SHIFT LEFT DOUBLE instruction 7-371
example A-31
SHIFT LEFT DOUBLE LOGICAL instruction 7-372
SHIFT LEFT SINGLE instruction 7-372
example A-32
SHIFT LEFT SINGLE LOGICAL instruction 7-373
SHIFT RIGHT DOUBLE instruction 7-374
SHIFT RIGHT DOUBLE LOGICAL instruction 7-374
SHIFT RIGHT SINGLE instruction 7-375
SHIFT RIGHT SINGLE LOGICAL instruction 7-376
SHIFT SIGNIFICAND LEFT DFP instructions 20-54
SHIFT SIGNIFICAND RIGHT DFP instructions 20-54
shifting
floating-point
See normalization
short binary-floating-point number 19-2
short decimal-floating-point number 20-3
short hexadecimal-floating-point number 18-2
short I/O block 16-24, 16-39
SHY (SUBTRACT HALFWORD) instruction 7-388
SI instruction format 5-6, 5-7
SID
See subsystem-identification word
side-effect-access facility 1-25
facility indication 4-100
sign bit
binary 7-4
floating point 9-2
sign codes (decimal) 8-2
signal (I/O) 17-11
clear
See clear signal
halt
See halt signal
reset
See reset signal
SIGNAL PROCESSOR instruction 10-133
orders 4-84
status 4-94
signaling NaN (SNaN) 9-2, 19-4, 20-7
signed binary
arithmetic 7-5
comparison 7-7
integer 7-4
examples A-2
signed exponent 9-2
signed-packed-decimal format 8-2
used in DFP formatting instructions 20-8
signed-packed-decimal numbers
examples A-4
signed scaling exponent (Ω) 9-13
significance

loss 18-2
in HFP addition 18-9
mask (in PSW) 4-7
requested in REROUND 20-52
See also significant digits
starter (in EDIT) 8-8
significand
in BFP data 19-3
in DFP data 20-5
significant digits 20-5
SIGP
See SIGNAL PROCESSOR instruction
SIGP (SIGNAL PROCESSOR) instruction 10-133
SIL instruction format 5-7
simulated IEEE-exception data exception 6-25
single-access reference 5-125
single-path mode 15-4, 15-23
SIY instruction format 5-6, 5-7
size notation xxviii
size of address
controlled by addressing mode 5-10
in CCW 15-32
skip flag in CCW 15-32
effect on data transfer of 15-64
SL (SUBTRACT LOGICAL) instruction 7-390
SLA (SHIFT LEFT SINGLE) instruction 7-372
example A-32
SLAG (SHIFT LEFT SINGLE) instruction 7-372
SLAK (SHIFT LEFT SINGLE) instruction 7-372
SLB (SUBTRACT LOGICAL WITH BORROW) instruction 7-391
SLBG (SUBTRACT LOGICAL WITH BORROW) instruction 7-391
SLBGR (SUBTRACT LOGICAL WITH BORROW) instruction 7-391
SLBR (SUBTRACT LOGICAL WITH BORROW) instruction 7-391
SLDA (SHIFT LEFT DOUBLE) instruction 7-371
example A-31
SLDL (SHIFT LEFT DOUBLE LOGICAL) instruction 7-372
SLDT (SHIFT SIGNIFICAND LEFT) DFP instruction 20-54
SLFI (SUBTRACT LOGICAL IMMEDIATE) instruction 7-390
SLGFI (SUBTRACT LOGICAL IMMEDIATE) instruction 7-390
SLGFR (SUBTRACT LOGICAL) instruction 7-389
SLGR (SUBTRACT LOGICAL) instruction 7-389, 7-390
SLGRK (SUBTRACT LOGICAL) binary instruction 7-389
SLHHHR (SUBTRACT LOGICAL HIGH) instruction 7-391
SLHHLR (SUBTRACT LOGICAL HIGH) instruction 7-391
SLI (suppress-length-indication) flag in CCW 15-32, 15-39
for immediate operations 15-60
SLL (SHIFT LEFT SINGLE LOGICAL) instruction 7-373
SLLG (SHIFT LEFT SINGLE LOGICAL) instruction 7-373
SLLK (SHIFT LEFT SINGLE LOGICAL) instruction 7-373
SLR (SUBTRACT LOGICAL) instruction 7-389
SLRK (SUBTRACT LOGICAL) binary instruction 7-389
SLXT (SHIFT SIGNIFICAND LEFT) DFP instruction 20-54
SLY (SUBTRACT LOGICAL) instruction 7-390
SNaN (signaling NaN) 9-2, 19-4, 20-7
solicited interruption condition (I/O) 16-3
solid errors 11-6
sorting
extended 1-31
sorting instructions
example A-53
See also COMPARE AND FORM CODEWORD instruction, UPDATE TREE instruction
source of interruption
identified by interruption code 6-5
SP (SUBTRACT DECIMAL) instruction 8-13
space-switch event 6-37
control bit
in ASTE 3-32
space-switch-event-control bit
home 3-45
primary 3-43
special-operation exception 6-38
special QNaN 9-3, 19-4, 20-7
special quantum-handling operations 20-3
specific-operand serialization 5-132
specification exception 6-40
SPKA (SET PSW KEY FROM ADDRESS) instruction 10-124
SPM (SET PROGRAM MASK) instruction 7-371
SPT (SET CPU TIMER) instruction 10-123
SPX (SET PREFIX) instruction 10-123
SQD (SQUARE ROOT) HFP instruction 18-23
SQDB (SQUARE ROOT) BFP instruction 19-40
SQDBR (SQUARE ROOT) BFP instruction 19-40
SQDR (SQUARE ROOT) HFP instruction 18-23
SQE (SQUARE ROOT) HFP instruction 18-23
SQEB (SQUARE ROOT) BFP instruction 19-40
SQEBR (SQUARE ROOT) BFP instruction 19-40
SQER (SQUARE ROOT) HFP instruction 18-23
SQUARE ROOT BFP instructions 19-40

SQUARE ROOT HFP instructions 18-20
SQXBR (SQUARE ROOT) BFP instruction 19-40
SQXR (SQUARE ROOT) HFP instruction 18-23
SR (SUBTRACT) binary instruction 7-387
SRA (SHIFT RIGHT SINGLE) instruction 7-375
SRAG (SHIFT RIGHT SINGLE) instruction 7-375
SRAK (SHIFT RIGHT SINGLE) instruction 7-375
SRDA (SHIFT RIGHT DOUBLE) instruction 7-374
SRDL (SHIFT RIGHT DOUBLE LOGICAL) instruction 7-374
SRDT (SHIFT SIGNIFICAND RIGHT) DFP instruction 20-54
SRK (SUBTRACT) binary instruction 7-387
SRL (SHIFT RIGHT SINGLE LOGICAL) instruction 7-376
SRLG (SHIFT RIGHT SINGLE LOGICAL) instruction 7-376
SRLK (SHIFT RIGHT SINGLE LOGICAL) instruction 7-376
SRNM (SET BFP ROUNDING MODE) floating-point-support instruction 9-47
SRNMB (SET BFP ROUNDING MODE) floating-point-support instruction 9-47
SRNMT (SET DFP ROUNDING MODE) floating-point-support instruction 9-47
SRP (SHIFT AND ROUND DECIMAL) instruction 8-12
  examples A-39
SRSD (secondary real-space designation) 3-44
SRST (SEARCH STRING) instruction 7-366
  examples A-31
SRSTKO (secondary real-space token origin) 3-44
SRSTU (SEARCH STRING UNICODE) instruction 7-368
SRTD (secondary region-table designation) 3-44
SRTL (secondary region-table length) 3-44
SRTO (secondary region-table origin) 3-44
SRXT (SHIFT SIGNIFICAND RIGHT) DFP instruction 20-54
SS instruction format 5-6, 5-7
SSAIR (SET SECONDARY ASN WITH INSTANCE) instruction 10-125
SSAR (SET SECONDARY ASN) instruction 10-125
  access registers 5-51
SSAR-cp or SSAIR-cp (SET SECONDARY ASN (WITH INSTANCE) instruction, to current primary) 10-125
SSAR-ss or SSAIR-ss (SET SECONDARY ASN (WITH INSTANCE) instruction, with space switching) 10-125
SSASTEO (subspace-AST-entry origin) 5-67
SSASTESN (subspace-AST-entry sequence number) 5-67
SSCH (START SUBCHANNEL) instruction 14-15
SSE instruction format 5-6, 5-7
SSF instruction format 5-7
SSID (subchannel-set identifier) 13-5
SSKE (SET STORAGE KEY EXTENDED) instruction 10-130
SSM (SET SYSTEM MASK) instruction 10-133
SSM-suppression-control bit 6-38, 10-133
SSTD (secondary segment-table designation) 3-44
SSTL (secondary segment-table length) 3-44
SSTO (secondary segment-table origin) 3-44
ST (STORE) binary instruction 7-376
stack-empty exception 6-44
stack-full exception 6-44
stack-operation exception 6-44
stack-specification exception 6-45
stack-type exception 6-45
stacking process 5-84
stacking PROGRAM CALL 5-72
STAM (STORE ACCESS MULTIPLE) instruction 7-377
STAMY (STORE ACCESS MULTIPLE) instruction 7-377
standalone dump 12-5
standard epoch (for TOD clock) 4-51
STAP (STORE CPU ADDRESS) instruction 10-136
start (CPU)
  function 4-3
  key 12-5
  signal-processor order 4-85
start function (I/O) 13-8, 15-20
  bit in SCSW for 16-13
  initiated by START SUBCHANNEL 14-15
  path management for 15-20
  pending 16-15, 16-36
START SUBCHANNEL instruction 14-15
  See also start function for I/O
  count of in measurement block 17-4
  deferred condition code for (in SCSW) 16-9
  function initiated by 15-20
  operation-request block (ORB) used by 15-24
state
  CPU
    See CPU state
  TOD-clock 4-47
state entry 5-81
status
  alert 16-17
  device 16-24
    effect of clear function on 15-16
    field-validity flag for (in subchannel logout) 16-49
    with inappropriate bit combination 16-50
  device-status check 16-50
  for SIGNAL PROCESSOR 4-85, 10-134

initial-status interruption
See initial-status-interruption control
intermediate 16-18, 16-36
primary 16-19
program
See PSW
resulting from signal-processor orders 4-94
secondary 16-19
storing of 4-81
manual key for 12-5
subchannel 16-24
status-control field (in SCSW) 16-17
status modifier (device status)
effect of in command chaining 15-63
status pending 16-19
status-verification facility 17-23
STC (STORE CHARACTER) instruction 7-378
STCH (STORE CHARACTER HIGH) instruction 7-378
STCK (STORE CLOCK) instruction 7-379
STCKC (STORE CLOCK COMPARATOR) instruction 10-135
STCKE (STORE CLOCK EXTENDED) instruction 7-380
STCKF (STORE CLOCK FAST) instruction 7-379
STCM (STORE CHARACTERS UNDER MASK) instruction 7-378
examples A-32
STCMH (STORE CHARACTERS UNDER MASK) instruction 7-378
STCMY (STORE CHARACTERS UNDER MASK) instruction 7-378
STCPS (STORE CHANNEL PATH STATUS) instruction 14-16
STCRW (STORE CHANNEL REPORT WORD) instruction 14-17
STCTG (STORE CONTROL) instruction 10-135
STCTL (STORE CONTROL) instruction 10-135
STCY (STORE CHARACTER) instruction 7-378
STD
See segment-table designation
STD (STORE) floating-point-support instruction 9-48
STDY (STORE) floating-point-support instruction 9-49
STE (STORE) floating-point-support instruction 9-48
STEY (STORE) floating-point-support instruction 9-49
STFH (STORE HIGH) instruction 7-384
STFL (STORE FACILITY LIST) instruction 10-138
STFL facility list 3-79, 3-85
STFLE (STORE FACILITY LIST EXTENDED) instruction 7-382
STFPC (STORE FPC) floating-point-support instruction 9-49
STG (STORE) binary instruction 7-377
STGRL (STORE RELATIVE LONG) instruction 7-377
STGSC (STORE GUARDED STORAGE CONTROLS) instruction 7-383
STH (STORE HALFWORD) instruction 7-383
STHH (STORE HALFWORD HIGH) instruction 7-384
STHHRL (STORE HALFWORD RELATIVE LONG) instruction 7-384
STHY (STORE HALFWORD) instruction 7-384
STIDP (STORE CPU ID) instruction 10-136
configuration identification 10-137
environment 10-136
format indication 10-136, 10-137
machine-type number 10-136, 10-137
STL (segment-table length) 3-42
STM (STORE MULTIPLE) instruction 7-385
example A-32
STMG (STORE MULTIPLE) instruction 7-385
STMH (STORE MULTIPLE HIGH) instruction 7-385
STMY (STORE MULTIPLE) instruction 7-385
STNSM (STORE THEN AND SYSTEM MASK) instruction 10-164
STO (segment-table origin) 3-42
STOC (STORE ON CONDITION) instruction 7-385
STOCFH (STORE HIGH ON CONDITION) binary instruction 7-386
STOCG (STORE ON CONDITION) instruction 7-385
stop
function 4-2
key 12-5
signal-processor order 4-85
stop and store status (signal-processor order) 4-85
stopped (signal-processor status) 4-96
stopped state
of CPU 4-2
effect on completion of store operations 5-124
of TOD clock 4-48
storage 3-1, 3-43
absolute 3-4
address wraparound
See wraparound
addressing 3-2
See also address
alteration
space-control bit 4-28
alteration manual controls 12-2
alteration PER event 4-36
bits for 3-43
mask for 4-27
assigned locations in 3-73
auxiliary 3-2, 3-38
block 3-4

testing for usability of 10-167
buffer (cache) 3-2
clearing of
See clearing operation
concurrency of access for references to 5-126
configuration of 3-4
direct-access 3-2
display 12-2
error 11-19
indirect 11-20
expanded 2-2
failing address in
See failing-storage address
interlocked update 5-123, 5-124
interlocks for virtual references 5-115
main 3-2
noninterlocked update of 5-124
nonvolatile 3-2
operand 5-9
reference to (fetch, store, update) 5-123
update reference 5-124
operand consistency 5-125
examples A-49, A-51
prefixing for 3-21
real 3-5
sequence of references to 5-113
for floating-point data 9-8
size
notation for xxviii
validation of 11-7
virtual 3-38
volatile 3-2
effect of power-on reset on 4-80
storage-access code (in subchannel logout) 16-50
storage-alteration-event bit 4-28
storage-alteration-event-control bit 3-43
home 3-45
primary 3-43
secondary 3-44
storage-area designation
for PER events 4-34
storage degradation (machine-check condition) 11-19
storage key 3-9
error in 11-19
sequence of references to 5-120
testing for usability of 10-167
validation of 11-7
storage-key function 1-32
storage-logical-validity bit 11-21
storage protection 3-10
during tracing 4-25
storage-protection-override-control bit 3-12
storage reconfiguration 1-31
storage sharing
by address spaces 3-38
by CPUs and the channel subsystem 3-4
examples A-45
in multiprocessing 4-83
STORE ACCESS MULTIPLE instructions 7-377
STORE binary instructions 7-376
STORE CHANNEL PATH STATUS instruction 14-16
STORE CHANNEL REPORT WORD instruction 14-17
channel-report word (CRW) stored by 17-26
STORE CHARACTER HIGH instruction 7-378, 7-384
STORE CHARACTER instructions 7-378
STORE CHARACTERS UNDER MASK instruction 7-378
examples A-32
STORE CLOCK COMPARATOR instruction 10-135
STORE CLOCK EXTENDED instruction 7-380
store-clock-fast facility 1-25, 4-24
facility indication 4-99
STORE CLOCK FAST instruction 7-379
STORE CLOCK instruction 7-379
STORE CONTROL instruction 10-135
STORE CPU ADDRESS instruction 10-136
store-CPU-counter-multiple facility
facility indication 4-101
STORE CPU ID instruction 10-136
configuration identification 10-136, 10-137
environment 10-136
format indication 10-136, 10-137
machine-type number 10-136, 10-137
STORE CPU TIMER instruction 10-138
store-facility-list-extended facility 1-25
facility indication 4-99
STORE FACILITY LIST EXTENDED instruction 7-382
STORE FACILITY LIST instruction 10-138
STORE floating-point-support instructions 9-48
STORE FPC floating-point-support instruction 9-49
STORE GUARDED STORAGE CONTROLS instruction 7-383
STORE HALFWORD HIGH instruction 7-384
STORE HALFWORD instructions 7-383
STORE HALFWORD RELATIVE LONG instruction 7-384
STORE HIGH instruction 7-384
STORE HIGH ON CONDITION instructions 7-386
store-hypervisor-information facility
facility indication 4-100
STORE MULTIPLE HIGH instruction 7-385
STORE MULTIPLE instruction
example A-32
STORE MULTIPLE instructions 7-385

STORE ON CONDITION instructions 7-385
STORE PAIR TO QUADWORD instruction 7-386
STORE PREFIX instruction 10-139
STORE REAL ADDRESS instruction 10-139
store reference 5-123
  access exceptions for 6-50
STORE RELATIVE LONG instructions 7-377
STORE REVERSED instructions 7-387
store status 4-81
  key 12-5
  signal-processor order for 4-85
store-status architectural-mode identification 3-76, 3-85
store status at address (signal-processor order) 4-86
STORE SUBCHANNEL instruction 14-18
STORE SYSTEM INFORMATION instruction 10-140
STORE THEN AND SYSTEM MASK instruction 10-164
STORE THEN OR SYSTEM MASK instruction 10-164
store using real address (PER event) 4-37
store-using-real-address-event mask 4-27
STORE USING REAL ADDRESS instruction 10-165
STOSM (STORE THEN OR SYSTEM MASK) instruction 10-164
STP clock source error (machine-check condition) 11-24
STP sync check (machine-check condition) 11-23
STP-timing-alert condition
  See timing alert
STPQ (STORE PAIR TO QUADWORD) instruction 7-386
STPT (STORE CPU TIMER) instruction 10-138
STPX (STORE PREFIX) instruction 10-139
STRAG (STORE REAL ADDRESS) instruction 10-139
streaming-mode control 15-25
string of interruptions 4-3, 6-57
  caused by clock comparator 4-60
  caused by CPU timer 4-63
STRV (STORE REVERSED) instruction 7-387
STRVG (STORE REVERSED) instruction 7-387
STRVH (STORE REVERSED) instruction 7-387
STSCH (STORE SUBCHANNEL) instruction 14-18
STSI (STORE SYSTEM INFORMATION) instruction 10-140
STURA (STORE USING REAL ADDRESS) instruction 10-165
STURG (STORE USING REAL ADDRESS) instruction 10-165
STY (STORE) binary instruction 7-377
SU (SUBTRACT UNNORMALIZED) HFP instruction 18-25
subchannel
  active allegiance for 15-13
  dedicated allegiance for 15-13
  effect of I/O-system reset on 17-15
  idle 16-14
  working allegiance for 15-12
subchannel-active bit 16-16, 16-36
subchannel addressing 13-5
subchannel control information in SCSW 16-12, 16-35
subchannel enabled bit in PMCW 15-3
subchannel-information block
  See SCHIB
subchannel key 15-25, 15-30, 16-9
  used as access key 3-11
  used for IPL 17-17
subchannel key check (in subchannel logout) 16-47
subchannel logout 16-47
subchannel number 13-5
subchannel operation mode
  command mode 13-9
  transport mode 13-9
subchannel-set identifier
  See SSID
Subchannel sets 13-2
subchannel status 16-24, 16-39
subchannel-status word
  See SCSW
subchannels 13-2
subclass-mask bits
  I/O-interruption
    See I/O-interruption subclass mask
  external-interruption 6-11
  machine-check 11-26
subnormal numbers
  class of DFP data 20-6
subroutine linkage 5-14
subspace-active bit 5-67
subspace-AST-entry origin (SSASTEO) 5-67
subspace-AST-entry sequence number (SSASTESN) 5-67
subspace-group control 3-43
subspace-group-control bit
  primary 3-43
  secondary 3-44
subspace groups
  introduction to 5-18
subspace-replacement
  exceptions 6-56
  operations 5-70
subsystem-identification word 14-1
subsystem-identification word (SID)
  assigned storage locations for 3-79, 3-85
subsystem-linkage-control bit 5-29, 5-35
subsystem reset 4-79

SUBTRACT BFP instructions 19-40
SUBTRACT binary instructions 7-387
SUBTRACT DECIMAL instruction 8-13
SUBTRACT DFP instructions 20-55
SUBTRACT HALFWORD instructions 7-388
SUBTRACT HIGH instruction 7-389
SUBTRACT LOGICAL HIGH instructions 7-391
SUBTRACT LOGICAL IMMEDIATE instructions 7-390
SUBTRACT LOGICAL instructions 7-389
SUBTRACT LOGICAL WITH BORROW instructions 7-391
SUBTRACT NORMALIZED
See SUBTRACT BFP instructions
SUBTRACT NORMALIZED HFP instructions 18-24
SUBTRACT UNNORMALIZED HFP instructions 18-25
successful-branching PER event 4-35
mask for 4-27
SUPERVISOR CALL instruction 7-392
supervisor-call interruption 6-57
supervisor state 4-6
support functions (I/O) 17-1
suppress-length-indication flag in CCW
See SLI
suppress-suspended-interruption control (I/O) 15-27, 16-12
used for IPL 17-17
suppression
exceptions to 5-26
of instruction execution 5-22
of unit of operation 5-25
SUR (SUBTRACT UNNORMALIZED) HFP instruction 18-25
suspend control 15-25
suspend-control bit 16-9
used for IPL 17-17
suspend flag in CCW 15-32
invalid 16-27
suspend function 13-11
suspended bit (in SCSW) 16-17, 16-36
suspension of channel-program execution 15-73
effect on DCTI of 15-75
intermediate interruption condition for 16-18
SVC (SUPERVISOR CALL) instruction 7-392
SW (SUBTRACT UNNORMALIZED) HFP instruction 18-25
swapping
by COMPARE (DOUBLE) AND SWAP instructions 7-144
by EXCLUSIVE OR instruction 7-255
SWR (SUBTRACT UNNORMALIZED) HFP instruction 18-25
SX (segment index) 3-39
SXBR (SUBTRACT) BFP instruction 19-40
SXR (SUBTRACT NORMALIZED) HFP instruction 18-24
SXTR (SUBTRACT) DFP instruction 20-55
SXTRA (SUBTRACT) DFP instruction 20-55
SY (SUBTRACT) binary instruction 7-388
symbol translation
use by VTAM 7-180
symbol-translation-option bit 7-173
synchronization
checkpoint 11-3
of CPU timer with TOD clock 4-62
of TOD clocks 4-48, 4-53
synchronization control 15-26
synchronous machine-check-interruption conditions 11-18
system
manual control of 12-1
organization of 2-1
system check stop 11-10
system damage 11-15
system mask (in PSW) 4-5
validity bit for 11-20
system recovery 11-15
system reset
I/O
See I/O-system reset
See reset
system-reset-clear key 12-6
system-reset-normal key 12-6
System/360 and System/370 I/O interface
See parallel-I/O interface

T

T (tera) xxviii
table of powers of 2 G-1
table-type bits 3-48, 3-50
in region-table entry 3-48
in segment-table entry 3-50
tables
ASN
See ASN first table, ASN second table
authority
See authority table
DAT
See page table, segment table, region table
entry
See entry table
linkage
See linkage tables
page
See page table
region

See region table
segment
See segment table
translation 3-45
TABORT (TRANSACTION ABORT) instruction 7-394
TAI (International Atomic Time) related to UTC 4-51
TAM (TEST ADDRESSING MODE) instruction 7-392
TAR (TEST ACCESS) instruction 10-165
target instruction 7-256
target-precision-constrained rounding 9-12
TB (TEST BLOCK) instruction 10-167
TBDR (CONVERT HFP TO BFP) floating-point-support instruction 9-28
TBDER (CONVERT HFP TO BFP) floating-point-support instruction 9-28
TBEGIN (TRANSACTION BEGIN) instruction 7-395
TBEGINC (TRANSACTION BEGIN CONSTRAINED) instruction 7-399
TCCB (transport-command-control block)
address of 15-35
contents of 15-37
TCDB (TEST DATA CLASS) BFP instruction 19-41
TCEB (TEST DATA CLASS) BFP instruction 19-41
DCW (device-command word) 15-38
TCW (transport-command word) 15-33
contents of 15-33
TCW channel program 13-7
TCW (transport-command word)
chaining 15-63
TCXB (TEST DATA CLASS) BFP instruction 19-41
TDCDT (TEST DATA CLASS) DFP instruction 20-56
TDCET (TEST DATA CLASS) DFP instruction 20-56
TDCXT (TEST DATA CLASS) DFP instruction 20-56
TDGDT (TEST DATA GROUP) DFP instruction 20-57
TDGET (TEST DATA GROUP) DFP instruction 20-57
TDGXT (TEST DATA GROUP) DFP instruction 20-57
TEND (TRANSACTION END) instruction 7-401
termination
of I/O operations
See conclusion of I/O operations
of instruction execution 5-23
for exigent machine-check conditions 11-10
of unit of operation 5-25
for exigent machine-check conditions 11-10
termination code (in subchannel logout) 16-50
field-validity flag for 16-49
TEST ACCESS instruction 10-165
TEST ADDRESSING MODE instruction 7-392
TEST AND SET instruction 7-392
TEST BLOCK instruction 10-167
TEST DATA CLASS BFP instructions 19-41
TEST DATA CLASS DFP instructions 20-56
TEST DATA GROUP DFP instructions 20-57
TEST DECIMAL instruction 8-14
test indicator 12-6
test-pending-external-interruption facility 1-25
facility indication 4-101
TEST PENDING EXTERNAL INTERRUPTION instruction 10-169
TEST PENDING INTERRUPTION instruction 14-19
interruption code stored by 14-20
TEST PROTECTION instruction 10-170
TEST SUBCHANNEL instruction 14-21
interruption-response block (IRB)used by 16-6
TEST UNDER MASK HIGH instruction 7-393
TEST UNDER MASK instruction 7-393
examples A-33
TEST UNDER MASK LOW instruction 7-393
testing for storage-block and storage-key usability 10-167
THDER (CONVERT BFP TO HFP) floating-point-support instruction 9-27
THDR (CONVERT BFP TO HFP) floating-point-support instruction 9-27
thread
identification (ID) 2-3, 4-83, 4-84, 4-92, 12-3, 12-7
program-specified maximum 2-3, 4-83, 4-91, 4-92, 12-3
width 4-83, 4-84
multithreading 2-3
TIC (transfer in channel) 15-77
invalid sequence of 16-27
TID
See thread identification
TID width
See thread identification width
ties 9-13
time-of-day clock
See TOD clock
timer
See CPU timer
timing
channel-subsystem 17-2
timing alert
external interruption 6-14
timing facilities 4-46
timing-facility bit (in PMCW) 15-4
timing-facility damage 11-16
for TOD clock 4-48
timing-alert condition
ETR-timing-alert condition 6-14
STP-timing-alert condition 6-14
TLB (translation-lookaside buffer) 3-62

entries 3-63
attachment of 3-63
clearing of 3-67
effect of translation changes on 3-67
usable state 3-65
TM (TEST UNDER MASK) instruction 7-393
examples A-33
TMH (TEST UNDER MASK HIGH) instruction 7-393
TMHH (TEST UNDER MASK) instruction 7-393
TMHL (TEST UNDER MASK) instruction 7-393
TML (TEST UNDER MASK LOW) instruction 7-393
TMLH (TEST UNDER MASK) instruction 7-393
TMLL (TEST UNDER MASK) instruction 7-393
TMY (TEST UNDER MASK) instruction 7-393
TOD clock 4-46
effect of power-on reset on 4-80
effect on clock-comparator interruption 6-12
effect on CPU-timer decrementing 4-63
effect on CPU-timer interruption 6-12
manual control of 4-48, 12-6
offset-based steering
overview 4-56
steering 4-54
episodes 4-56
facility indication 4-99
hardware based 4-54
overview 4-55
offset based 4-54
overview 4-55
offset update events 4-56
registers 4-56
UTC information block (UIB) 4-57
unique values of 4-49
validation of 11-9
value in trace entry 4-24
TOD-clock-control-override control 4-48
TOD-clock epoch-extension facility
facility indication 4-101
TOD-clock programmable field 4-51
TOD-clock programmable register 4-51
save areas for 3-82
validity bit for 11-22
TOD-clock-steering facility 1-26
TOD-clock-sync-control bit 4-48, 4-53
TOD-clock-synchronization facility 4-53
TP (TEST DECIMAL) instruction 8-14
TPEI (TEST PENDING EXTERNAL INTERRUPTION) instruction 10-169
TPI (TEST PENDING INTERRUPTION) instruction 14-19
interruption code stored by 14-20
TPROT (TEST PROTECTION) instruction 10-170
TR (TRANSLATE) instruction 7-402
example A-33
trace 4-12
entries 4-15
entry address 4-15
exceptions 6-56
table exception 6-45
TRACE instruction 10-173
trace entry for 4-24
TRACG (TRACE) instruction 10-173
transaction
definition of 5-91
transaction abort code
in the transaction diagnostic block 5-95
TRANSACTION ABORT instruction 7-394
transaction-abort PSW 5-93
TRANSACTION BEGIN CONSTRAINED instruction 7-399
TRANSACTION BEGIN instruction 7-395, 7-399
processing of 7-396, 7-399
transaction constraint exception 6-45
transaction diagnostic block
aborted-transaction-instruction-address field 5-95
breaking-event-address field 5-96
conflict-token field 5-95
data-exception-code field 5-96
exception-access-identification field 5-96
flags field 5-95
for a program interruption 3-83
format field 5-94
program-interruption-identification field 5-96
transaction-abort-code field 5-95
transaction-nesting-depth field 5-95
translation-exception-identification field 5-96
transaction-diagnostic-block address 5-93
transaction diagnostic control
in control register 2 5-92
transaction diagnostic scope
in control register 2 5-92
TRANSACTION END instruction 7-401
transaction-end PER event
mask for 4-27
transaction initiation 5-99
transaction nesting depth 5-93
in the transaction diagnostic block 5-95
transactional access 5-91
transactional execution
abort 5-89
and breaking events 4-45
and PER events 4-29, 4-40
commit 5-90
conflict 5-90
constrained transaction 5-90
constrained transaction-execution mode 5-90
controls 5-92
effects on CPU serialization 5-130

nested transaction 5-90, 5-91
nonconstrained transaction 5-91
nonconstrained transaction-execution mode 5-91
outer transaction 5-91
outermost transaction 5-91
program-interruption filtering 5-91
restricted instruction
tracing 4-25
transaction
definition of 5-91
transactional access 5-91
transactional-execution mode 5-91
transactional-execution-aborted event 6-46
transactional-execution control
in control register 0 4-10, 5-92
transactional-execution facility 1-26, 5-89
facility indication 4-100
instructions 5-97
restricted 5-97
operation 5-99
execution in the transactional-execution mode 5-99
normal transaction ending 5-99
program-interruption filtering 5-103
transaction abort conditions 5-99
transaction abort processing 5-102
transaction initiation 5-99
terminology 5-89
transactional-execution program-interruption filtering override
in control register 0 5-92, 4-10
transfer in channel
See TIC
transferring program control 5-71
TRANSLATE AND TEST instruction 7-403
example A-34
TRANSLATE AND TEST REVERSE instruction 7-408
TRANSLATE EXTENDED instruction 7-409
TRANSLATE instruction 7-402
example A-33
TRANSLATE ONE TO ONE instruction 7-411
TRANSLATE ONE TO TWO instruction 7-411
TRANSLATE TWO TO ONE instruction 7-411
TRANSLATE TWO TO TWO instruction 7-411
translation
address 3-38
See dynamic address translation
exception identification 3-76, 3-77
lookaside buffer
See TLB
PC-number 5-33
specification exception 6-46
tables for 3-45
translation exception identification
in the transaction diagnostic block 5-96
translation modes 3-40
translation parameters 3-40
translation path 3-64
TCCB (transport-command-control block) 15-36
transport-command-control block
See TCCB
transport-command word
See TCW
transport-status block
See TSB
trap action
IEEE 9-18
trap control block 10-175
TRAP instruction 10-174
trap save area 10-176
TRAP2 (TRAP) instruction 10-174
TRAP4 (TRAP) instruction 10-174
TRE (TRANSLATE EXTENDED) instruction 7-409
tree structure for sorting 7-419
example A-53
trial execution
for editing instructions and TRANSLATE instruction 5-27
for PER 4-29
trimodal addressing 5-10, 7-8
TROO (TRANSLATE ONE TO ONE) instruction 7-411
TROT (TRANSLATE ONE TO TWO) instruction 7-411
TRT (TRANSLATE AND TEST) instruction 7-403
example A-34
TRTO (TRANSLATE TWO TO ONE) instruction 7-411
TRTR (TRANSLATE AND TEST REVERSE) instruction 7-408
TRTT (TRANSLATE TWO TO TWO) instruction 7-411
true zero (HFP number) 18-1
TS (TEST AND SET) instruction 7-392
TSB (transport-status block) 15-47
address of 15-35
contents of 15-49, 15-56
TSCH (TEST SUBCHANNEL) instruction 14-21
two's complement binary notation 7-4
examples A-2
type of PROGRAM CALL 5-38

U

UIB
See UTC information block 4-57
ulp (unit in the last place) 9-7, 9-8, 19-3

underflow
See exponent underflow
unit check (device status)
in establishing dedicated allegiance 15-13
unit of operation 5-24
unlock A-47
example with FIFO queuing A-50
example with LIFO queuing A-49
unnormalized floating-point number 18-2
HFP data only 9-3
unordered (comparison to a NaN) 9-24
UNPACK ASCII instruction 7-417
UNPACK instruction 7-416
example A-35
UNPACK UNICODE instruction 7-418
UNPK (UNPACK) instruction 7-416
example A-35
UNPKA (PACK ASCII) instruction 7-417
UNPKU (PACK UNICODE) instruction 7-418
unprivileged instructions 4-6, 7-3
unsigned binary
arithmetic 7-6
integer 7-4
examples A-3
in address generation 5-11
unsigned binary integer
conversion to floating point 19-21
unsigned-packed-decimal format 8-2
used in DFP formatting instructions 20-8
unsigned scaling exponent (α) 9-12
unsolicited interruption condition (I/O) 16-3
unstack-suppression bit 5-79
unstacking process 5-86
update reference 5-124
UPDATE TREE instruction 7-419
example A-53
UPT (UPDATE TREE) instruction 7-419
example A-53
usable ALB entry 5-65
UTC (Coordinated Universal Time) used in TOD epoch 4-51
UTC information block (UIB) 4-57

V

VA (VECTOR ADD) vector instruction 22-3
VAC (VECTOR ADD WITH CARRY) vector instruction 22-4
VACC (VECTOR ADD COMPUTE CARRY) vector instruction 22-4
VACCC (VECTOR ADD WITH CARRY COMPUTE CARRY) vector instruction 22-5
valid ART-table entry 5-65
valid CBC 11-2
valid floating-point-register numbers 9-9
valid region-table, segment-table, or page-table entry 3-63
validation 11-6
of registers 11-9
of storage 11-7
of storage key 11-7
of TOD clock 11-9
validity bit for backward stack-entry address 5-80
validity bit for forward-section-header address 5-81
validity bits
in machine-check-interruption code 11-20
in subchannel logout 16-49
VAP (VECTOR ADD DECIMAL) instruction 25-3
variable-length field 3-3
VAVG (VECTOR AVERAGE) vector instruction 22-6
VAVGL (VECTOR AVERAGE LOGICAL) vector instruction 22-6
VBPERM (VECTOR BIT PERMUTE) vector instruction 21-4
VCDG (VECTOR FP CONVERT FROM FIXED 64-BIT) vector instruction 24-15
VCDLG (VECTOR FP CONVERT FROM LOGICAL 64-BIT) vector instruction 24-16
VCEQ (VECTOR COMPARE EQUAL) vector instruction 22-7
VCGD (VECTOR FP CONVERT TO FIXED 64-BIT) vector instruction 24-17
VCH (VECTOR COMPARE HIGH) vector instruction 22-8
VCHL (VECTOR COMPARE HIGH LOGICAL) vector instruction 22-9
VCKSM (VECTOR CHECKSUM) vector instruction 22-6
VCLGD (VECTOR FP CONVERT TO LOGICAL 64-BIT) vector instruction 24-19
VCLZ (VECTOR COUNT LEADING ZEROS) vector instruction 22-10
VCP (VECTOR COMPARE DECIMAL) instruction 25-4
VCTZ (VECTOR COUNT TRAILING ZEROS) vector instruction 22-10
VCVB (VECTOR CONVERT TO BINARY) instruction 25-5
VCVBG (VECTOR CONVERT TO BINARY) instruction 25-5
VCVD (VECTOR CONVERT TO DECIMAL) instruction 25-6
VCVDG (VECTOR CONVERT TO DECIMAL) instruction 25-6
VDP (VECTOR DIVIDE DECIMAL) instruction 25-8
VEC (VECTOR ELEMENT COMPARE) vector instruction 22-7

VECL (VECTOR ELEMENT COMPARE LOGICAL) vector instruction 22-7
vector
registers 2-5
validity bit for 11-21, 11-22
VECTOR ADD COMPUTE CARRY vector instruction 22-4
VECTOR ADD DECIMAL instruction 25-3
VECTOR ADD vector instruction 22-3
VECTOR ADD WITH CARRY COMPUTE CARRY vector instruction 22-5
VECTOR ADD WITH CARRY vector instruction 22-4
VECTOR AND vector instruction 22-5
VECTOR AND WITH COMPLEMENT vector instruction 22-5
VECTOR AVERAGE LOGICAL vector instruction 22-6
VECTOR AVERAGE vector instruction 22-6
VECTOR BIT PERMUTE vector instruction 21-4
VECTOR CHECKSUM vector instruction 22-6
VECTOR COMPARE DECIMAL instruction 25-4
VECTOR COMPARE EQUAL vector instruction 22-7
VECTOR COMPARE HIGH LOGICAL vector instruction 22-9
VECTOR COMPARE HIGH vector instruction 22-8
VECTOR CONVERT TO BINARY instructions 25-5
VECTOR CONVERT TO DECIMAL instructions 25-6
VECTOR COUNT LEADING ZEROS vector instruction 22-10
VECTOR COUNT TRAILING ZEROS vector instruction 22-10
VECTOR DIVIDE DECIMAL instruction 25-8
VECTOR ELEMENT COMPARE LOGICAL vector instruction 22-7
VECTOR ELEMENT COMPARE vector instruction 22-7
VECTOR ELEMENT ROTATE AND INSERT UNDER MASK vector instruction 22-22
VECTOR ELEMENT ROTATE LEFT LOGICAL vector instruction 22-21
VECTOR ELEMENT SHIFT LEFT vector instruction 22-23
VECTOR ELEMENT SHIFT RIGHT ARITHMETIC vector instruction 22-23
VECTOR ELEMENT SHIFT RIGHT LOGICAL vector instruction 22-24
vector-enhancements facility 1
facility indication 4-101
vector-exception code (VXC) 6-20
VECTOR EXCLUSIVE OR vector instruction 22-11
vector facility for z/Architecture
facility indication 4-100
vector facility for z/architecture 1-26
VECTOR FIND ANY ELEMENT EQUAL vector instruction 23-1
VECTOR FIND ELEMENT EQUAL vector instruction 23-3
VECTOR FIND ELEMENT NOT EQUAL vector instruction 23-4
VECTOR FP ADD vector instruction 24-3
VECTOR FP COMPARE AND SIGNAL SCALAR vector instruction 24-8
VECTOR FP COMPARE EQUAL vector instruction 24-9
VECTOR FP COMPARE HIGH OR EQUAL vector instruction 24-13
VECTOR FP COMPARE HIGH vector instruction 24-11
VECTOR FP COMPARE SCALAR vector instruction 24-7
VECTOR FP CONVERT FROM FIXED 64-BIT vector instruction 24-15
VECTOR FP CONVERT FROM LOGICAL 64-BIT vector instruction 24-16
VECTOR FP CONVERT TO FIXED 64-BIT vector instruction 24-17
VECTOR FP CONVERT TO LOGICAL 64-BIT vector instruction 24-19
VECTOR FP DIVIDE vector instruction 24-21
VECTOR FP LOAD LENGTHENED vector instruction 24-25
VECTOR FP LOAD ROUNDED vector instruction 24-26
VECTOR FP MAXIMUM vector instruction 24-27
VECTOR FP MINIMUM vector instruction 24-33
VECTOR FP MULTIPLY AND ADD vector instruction 24-41
VECTOR FP MULTIPLY AND SUBTRACT vector instruction 24-41
VECTOR FP MULTIPLY vector instruction 24-39
VECTOR FP NEGATIVE MULTIPLY AND ADD vector instruction 24-41
VECTOR FP NEGATIVE MULTIPLY AND SUBTRACT vector instruction 24-41
VECTOR FP PERFORM SIGN OPERATION vector instruction 24-43
VECTOR FP SQUARE ROOT vector instruction 24-44
VECTOR FP SUBTRACT vector instruction 24-45
VECTOR FP TEST DATA CLASS IMMEDIATE vector instruction 24-46
VECTOR GALOIS FIELD MULTIPLY SUM AND ACCUMULATE vector instruction 22-12
VECTOR GALOIS FIELD MULTIPLY SUM vector instruction 22-11
VECTOR GATHER ELEMENT vector instruction 21-4

VECTOR GENERATE BYTE MASK vector instruction 21-5
VECTOR GENERATE MASK vector instruction 21-5
VECTOR ISOLATE STRING vector instruction 23-5
VECTOR LOAD AND REPLICATE vector instruction 21-6
VECTOR LOAD COMPLEMENT vector instruction 22-12
VECTOR LOAD ELEMENT IMMEDIATE vector instruction 21-7
VECTOR LOAD ELEMENT vector instruction 21-6
VECTOR LOAD FP INTEGER vector instruction 24-23
VECTOR LOAD GR FROM VR ELEMENT vector instruction 21-7
VECTOR LOAD IMMEDIATE DECIMAL instruction 25-9
VECTOR LOAD LOGICAL ELEMENT AND ZERO vector instruction 21-8
VECTOR LOAD MULTIPLE vector instruction 21-9
VECTOR LOAD POSITIVE vector instruction 22-12
VECTOR LOAD RIGHTMOST WITH LENGTH instruction 21-9
VECTOR LOAD TO BLOCK BOUNDARY vector instruction 21-10
VECTOR LOAD vector instruction 21-6
VECTOR LOAD VR ELEMENT FROM GR vector instruction 21-11
VECTOR LOAD VR FROM GRS DISJOINT vector instruction 21-11
VECTOR LOAD WITH LENGTH vector instruction 21-11
VECTOR MAXIMUM LOGICAL vector instruction 22-13
VECTOR MAXIMUM vector instruction 22-13
VECTOR MERGE HIGH vector instruction 21-12
VECTOR MERGE LOW vector instruction 21-12
VECTOR MINIMUM LOGICAL vector instruction 22-14
VECTOR MINIMUM vector instruction 22-13
VECTOR MULTIPLY AND ADD EVEN vector instruction 22-15
VECTOR MULTIPLY AND ADD HIGH vector instruction 22-15
VECTOR MULTIPLY AND ADD LOGICAL EVEN vector instruction 22-15
VECTOR MULTIPLY AND ADD LOGICAL HIGH vector instruction 22-15
VECTOR MULTIPLY AND ADD LOGICAL ODD vector instruction 22-16
VECTOR MULTIPLY AND ADD LOW vector instruction 22-14
VECTOR MULTIPLY AND ADD ODD vector instruction 22-16
VECTOR MULTIPLY AND SHIFT DECIMAL instruction 25-10
VECTOR MULTIPLY DECIMAL instruction 25-10
VECTOR MULTIPLY EVEN vector instruction 22-18
VECTOR MULTIPLY HIGH vector instruction 22-16
VECTOR MULTIPLY LOGICAL EVEN vector instruction 22-18
VECTOR MULTIPLY LOGICAL HIGH vector instruction 22-17
VECTOR MULTIPLY LOGICAL ODD vector instruction 22-18
VECTOR MULTIPLY LOW vector instruction 22-17
VECTOR MULTIPLY ODD vector instruction 22-18
VECTOR NAND vector instruction 22-20
VECTOR NOR vector instruction 22-19
VECTOR NOT EXCLUSIVE OR vector instruction 22-20
VECTOR OR vector instruction 22-20
VECTOR OR WITH COMPLEMENT vector instruction 22-21
VECTOR PACK LOGICAL SATURATE vector instruction 21-14
VECTOR PACK SATURATE vector instruction 21-13
VECTOR PACK vector instruction 21-13
VECTOR PACK ZONED instruction 25-12
vector-packed-decimal facility 1-26
  facility indication 4-101
VECTOR PERFORM SIGN OPERATION DECIMAL instruction 25-13
VECTOR PERMUTE DOUBLEWORD IMMEDIATE vector instruction 21-15
VECTOR PERMUTE vector instruction 21-15
VECTOR POPULATION COUNT vector instruction 22-21
VECTOR REMAINDER DECIMAL instruction 25-14
VECTOR REPLICATE IMMEDIATE vector instruction 21-16
VECTOR REPLICATE vector instruction 21-15
VECTOR SCATTER ELEMENT vector instruction 21-16
VECTOR SELECT vector instruction 21-17
VECTOR SHIFT AND DIVIDE DECIMAL instruction 25-16
VECTOR SHIFT AND ROUND DECIMAL instruction 25-17
VECTOR SHIFT LEFT BY BYTE vector instruction 22-25
VECTOR SHIFT LEFT DOUBLE BY BYTE vector instruction 22-25
VECTOR SHIFT LEFT vector instruction 22-25
VECTOR SHIFT RIGHT ARITHMETIC BY BYTE vector instruction 22-26
VECTOR SHIFT RIGHT ARITHMETIC vector instruction 22-25

VECTOR SHIFT RIGHT LOGICAL BY BYTE vector instruction 22-26
VECTOR SHIFT RIGHT LOGICAL vector instruction 22-26
VECTOR SIGN EXTEND TO DOUBLEWORD vector instruction 21-17
VECTOR STORE ELEMENT vector instruction 21-18
VECTOR STORE MULTIPLE vector instruction 21-19
VECTOR STORE RIGHTMOST WITH LENGTH instruction 21-19
VECTOR STORE vector instruction 21-18
VECTOR STORE WITH LENGTH vector instruction 21-20
VECTOR STRING RANGE COMPARE vector instruction 23-6
VECTOR SUBTRACT COMPUTE BORROW INDICATION vector instruction 22-27
VECTOR SUBTRACT DECIMAL instruction 25-18
VECTOR SUBTRACT vector instruction 22-26
VECTOR SUBTRACT WITH BORROW COMPUTE BORROW INDICATION vector instruction 22-28
VECTOR SUBTRACT WITH BORROW INDICATION vector instruction 22-27
VECTOR SUM ACROSS DOUBLEWORD vector instruction 22-28
VECTOR SUM ACROSS QUADWORD vector instruction 22-29
VECTOR SUM ACROSS WORD vector instruction 22-29
VECTOR TEST DECIMAL instruction 25-20
VECTOR TEST UNDER MASK vector instruction 22-30
VECTOR UNPACK HIGH vector instruction 21-20
VECTOR UNPACK LOGICAL HIGH vector instruction 21-21
VECTOR UNPACK LOGICAL LOW vector instruction 21-22
VECTOR UNPACK LOW vector instruction 21-21
VECTOR UNPACK ZONED instruction 25-20
VERIM (VECTOR ELEMENT ROTATE AND INSERT UNDER MASK) vector instruction 22-22
VERLL (VECTOR ELEMENT ROTATE LEFT LOGICAL) vector instruction 22-21
VERLLV (VECTOR ELEMENT ROTATE LEFT LOGICAL) vector instruction 22-21
VESL (VECTOR ELEMENT SHIFT LEFT) vector instruction 22-23
VESLV (VECTOR ELEMENT SHIFT LEFT) vector instruction 22-23
VESRA (VECTOR ELEMENT SHIFT RIGHT ARITHMETIC) vector instruction 22-23
VESRAV (VECTOR ELEMENT SHIFT RIGHT ARITHMETIC) vector instruction 22-23
VESRL (VECTOR ELEMENT SHIFT RIGHT LOGICAL) vector instruction 22-24
VESRLV (VECTOR ELEMENT SHIFT RIGHT LOGICAL) vector instruction 22-24
VFA (VECTOR FP ADD) vector instruction 24-3
VFAE (VECTOR FIND ANY ELEMENT EQUAL) vector instruction 23-1
VFCE (VECTOR FP COMPARE EQUAL) vector instruction 24-9
VFCH (VECTOR FP COMPARE HIGH) vector instruction 24-11
VFCHE (VECTOR FP COMPARE HIGH OR EQUAL) vector instruction 24-13
VFD (VECTOR FP DIVIDE) vector instruction 24-21
VFEE (VECTOR FIND ELEMENT EQUAL) vector instruction 23-3
VFENE (VECTOR FIND ELEMENT NOT EQUAL) vector instruction 23-4
VFI (VECTOR LOAD FP INTEGER) vector instruction 24-23
VFLL (VECTOR FP LOAD LENGTHENED) vector instruction 24-25
VFLR (VECTOR FP LOAD ROUNDED) vector instruction 24-26
VFM (VECTOR FP MULTIPLY) vector instruction 24-39
VFMA (VECTOR FP MULTIPLY AND ADD) vector instruction 24-41
VFMAX (VECTOR FP MAXIMUM) vector instruction 24-27
VFMIN (VECTOR FP MINIMUM) vector instruction 24-33
VFMS (VECTOR FP MULTIPLY AND SUBTRACT) vector instruction 24-41
VFNMA (VECTOR FP NEGATIVE MULTIPLY AND ADD) vector instruction 24-41
VFNMS (VECTOR FP NEGATIVE MULTIPLY AND SUBTRACT) vector instruction 24-41
VFPSO (VECTOR FP PERFORM SIGN OPERATION) vector instruction 24-43
VFS (VECTOR FP SUBTRACT) vector instruction 24-45
VFSQ (VECTOR FP SQUARE ROOT) vector instruction 24-44
VFTCI (VECTOR FP TEST DATA CLASS IMMEDIATE) vector instruction 24-46
VGBM (VECTOR GENERATE BYTE MASK) vector instruction 21-5
VGEF (VECTOR GATHER ELEMENT) vector instruction 21-4
VGEG (VECTOR GATHER ELEMENT) vector instruction 21-4

VGFM (VECTOR GALOIS FIELD MULTIPLY SUM) vector instruction 22-11
VGFMA (VECTOR GALOIS FIELD MULTIPLY SUM AND ACCUMULATE) vector instruction 22-12
VGM (VECTOR GENERATE MASK) vector instruction 21-5
view (of floating-point numbers) 9-2
  common-rounding-point 9-13
  fraction 9-2
  left-units view 9-2
  right-units view 9-2
virtual address 3-5
virtual machine
  extensions for 1-32
virtual storage 3-38
VISTR (VECTOR ISOLATE STRING) vector instruction 23-5
VL (VECTOR LOAD) vector instruction 21-6
VLBB (VECTOR LOAD TO BLOCK BOUNDARY) vector instruction 21-10
VLC (VECTOR LOAD COMPLEMENT) vector instruction 22-12
VLEB (VECTOR LOAD ELEMENT) vector instruction 21-6
VLEF (VECTOR LOAD ELEMENT) vector instruction 21-7
VLEG (VECTOR LOAD ELEMENT) vector instruction 21-7
VLEH (VECTOR LOAD ELEMENT) vector instruction 21-7
VLEIB (VECTOR LOAD ELEMENT IMMEDIATE) vector instruction 21-7
VLEIF (VECTOR LOAD ELEMENT IMMEDIATE) vector instruction 21-7
VLEIG (VECTOR LOAD ELEMENT IMMEDIATE) vector instruction 21-7
VLEIH (VECTOR LOAD ELEMENT IMMEDIATE) vector instruction 21-7
VLGV (VECTOR LOAD GR FROM VR ELEMENT) vector instruction 21-7
VLIP (VECTOR LOAD IMMEDIATE DECIMAL) instruction 25-9
VLL (VECTOR LOAD WITH LENGTH) vector instruction 21-11
VLLEZ (VECTOR LOAD LOGICAL ELEMENT AND ZERO) vector instruction 21-8
VLM (VECTOR LOAD MULTIPLE) vector instruction 21-9
VLP (VECTOR LOAD POSITIVE) vector instruction 22-12
VLR (VECTOR LOAD) vector instruction 21-6
VLREP (VECTOR LOAD AND REPLICATE) vector instruction 21-6
VLRL (VECTOR LOAD RIGHTMOST WITH LENGTH) vector instruction 21-9
VLRLR (VECTOR LOAD RIGHTMOST WITH LENGTH) vector instruction 21-9
VLVG (VECTOR LOAD VR ELEMENT FROM GR) vector instruction 21-11
VLVGP (VECTOR LOAD VR FROM GRS DISJOINT) vector instruction 21-11
VM-data-space facility 1-32
VMAE (VECTOR MULTIPLY AND ADD EVEN) vector instruction 22-15
VMAH (VECTOR MULTIPLY AND ADD HIGH) vector instruction 22-15
VMAL (VECTOR MULTIPLY AND ADD LOW) vector instruction 22-14
VMALE (VECTOR MULTIPLY AND ADD LOGICAL EVEN) vector instruction 22-15
VMALH (VECTOR MULTIPLY AND ADD LOGICAL HIGH) vector instruction 22-15
VMALO (VECTOR MULTIPLY AND ADD LOGICAL ODD) vector instruction 22-16
VMAO *VECTOR MULTIPLY AND ADD ODD) vector instruction 22-16
VME (VECTOR MULTIPLY EVEN) vector instruction 22-18
VMH (VECTOR MULTIPLY HIGH) vector instruction 22-16
VML (VECTOR MULTIPLY LOW) vector instruction 22-17
VMLE (VECTOR MULTIPLY LOGICAL EVEN) vector instruction 22-18
VMLH (VECTOR MULTIPLY LOGICAL HIGH) vector instruction 22-17
VMLO (VECTOR MULTIPLY LOGICAL ODD) vector instruction 22-18
VMN (VECTOR MINIMUM) vector instruction 22-13
VMNL (VECTOR MINIMUM LOGICAL) vector instruction 22-14
VMO (VECTOR MULTIPLY ODD) vector instruction 22-18
VMP (VECTOR MULTIPLY DECIMAL) instruction 25-10
VMRH (VECTOR MERGE HIGH) vector instruction 21-12
VMRL (VECTOR MERGE LOW) vector instruction 21-12
VMSL (VECTOR MULTIPLY SUM LOGICAL) vector instruction 22-19
VMSP (VECTOR MULTIPLY AND SHIFT DECIMAL) instruction 25-11
VMX (VECTOR MAXIMUM) vector instruction 22-13
VMXL (VECTOR MAXIMUM LOGICAL) vector instruction 22-13
VN (VECTOR AND) vector instruction 22-5

VNC (VECTOR AND WITH COMPLEMENT) vector instruction 22-5
VNN (VECTOR NAND) vector instruction 22-20
VNO (VECTOR NOR) vector instruction 22-20
VNX (VECTOR NOT EXCLUSIVE OR) vector instruction 22-20
VO (VECTOR OR) vector instruction 22-20
VOC (VECTOR OR WITH COMPLEMENT) vector instruction 22-21
volatile storage 3-2
  effect of power-on reset on 4-80
voting digit 9-13
VPDI (VECTOR PERMUTE DOUBLEWORD IMMEDIATE) vector instruction 21-15
VPERM (VECTOR PERMUTE) vector instruction 21-15
VPK (VECTOR PACK) vector instruction 21-13
VPKLS (VECTOR PACK LOGICAL SATURATE) vector instruction 21-14
VPKS (VECTOR PACK SATURATE) vector instruction 21-13
VPKZ (VECTOR PACK ZONED) instruction 25-12
VPOPCT (VECTOR POPULATION COUNT) vector instruction 22-21
VPSOP (VECTOR PERFORM SIGN OPERATION DECIMAL) instruction 25-13
VREP (VECTOR REPLICATE) vector instruction 21-15
VREPI (VECTOR REPLICATE IMMEDIATE) vector instruction 21-16
VRP (VECTOR REMAINDER DECIMAL) instruction 25-14
VS (VECTOR SUBTRACT) vector instruction 22-26
VSBCBI (VECTOR SUBTRACT WITH BORROW COMPUTE BORROW INDICATION) vector instruction 22-28
VSBI (VECTOR SUBTRACT WITH BORROW INDICATION) vector instruction 22-27
VSCBI (VECTOR SUBTRACT COMPUTE BORROW INDICATION) vector instruction 22-27
VSCEF (VECTOR SCATTER ELEMENT) vector instruction 21-16
VSCEG (VECTOR SCATTER ELEMENT) vector instruction 21-16
VSDP (VECTOR SHIFT AND DIVIDE DECIMAL) instruction 25-16
VSEG (VECTOR SIGN EXTEND TO DOUBLEWORD) vector instruction 21-17
VSEL (VECTOR SELECT) vector instruction 21-17
VSL (VECTOR SHIFT LEFT) vector instruction 22-25
VSLB (VECTOR SHIFT LEFT BY BYTE) vector instruction 22-25
VSLDB (VECTOR SHIFT LEFT DOUBLE BY BYTE) vector instruction 22-25
VSP (VECTOR SUBTRACT DECIMAL) instruction 25-18
VSRA (VECTOR SHIFT RIGHT ARITHMETIC) vector instruction 22-25
VSRAB (VECTOR SHIFT RIGHT ARITHMETIC BY BYTE) vector instruction 22-26
VSRL (VECTOR SHIFT RIGHT LOGICAL) vector instruction 22-26
VSRLB (VECTOR SHIFT RIGHT LOGICAL) vector instruction 22-26
VSRP (VECTOR SHIFT AND ROUND DECIMAL) instruction 25-17
VST (VECTOR STORE) vector instruction 21-18
VSTEB (VECTOR STORE ELEMENT) vector instruction 21-18
VSTEF (VECTOR STORE ELEMENT) vector instruction 21-18
VSTEG (VECTOR STORE ELEMENT) vector instruction 21-18
VSTEH (VECTOR STORE ELEMENT) vector instruction 21-18
VSTL (VECTOR STORE WITH LENGTH) vector instruction 21-20
VSTM (VECTOR STORE MULTIPLE) vector instruction 21-19
VSTRC (VECTOR STRING RANGE COMPARE) vector instruction 23-6
VSTRLR (VECTOR LOAD RIGHTMOST WITH LENGTH) vector instruction 21-19
VSTRLR (VECTOR STORE RIGHTMOST WITH LENGTH) vector instruction 21-19
VSUM (VECTOR SUM ACROSS WORD) vector instruction 22-29
VSUMG (VECTOR SUM ACROSS DOUBLEWORD) vector instruction 22-28
VSUMQ (VECTOR SUM ACROSS QUADWORD) vector instruction 22-29
VTAM use of symbol translation 7-180
VTM (VECTOR TEST UNDER MASK) vector instruction 22-30
VTP (VECTOR TEST DECIMAL) instruction 25-20
VUPH (VECTOR UNPACK HIGH) vector instruction 21-20
VUPKZ (VECTOR UNPACK ZONED) instruction 25-20
VUPL (VECTOR UNPACK LOW) vector instruction 21-21
VUPLH (VECTOR UNPACK LOGICAL) vector instruction 21-21
VUPLL (VECTOR UNPACK LOGICAL LOW) vector instruction 21-22

VX (VECTOR EXCLUSIVE OR) vector instruction 22-11
VXC (vector-exception code) 6-20

W

wait indicator 12-6
wait-state bit
in the PSW 4-6
warning (machine-check condition) 11-17
subclass-mask bit for 11-27
warning-track-interruption facility 1-27
WFC (VECTOR FP COMPARE SCALAR) vector instruction 24-7
WFK (VECTOR FP COMPARE AND SIGNAL SCALAR) vector instruction 24-8
word 3-3
word-concurrent storage references 5-126
working allegiance (I/O) 15-12
wraparound
of instruction addresses 5-10
of PER addresses 4-34
of register numbers
for LOAD MULTIPLE instruction 7-282
for STORE MULTIPLE instruction 7-385
of storage addresses 3-7
controlled by addressing mode 3-7
for MOVE INVERSE instruction 7-290
for MOVE LONG EXTENDED instruction 7-296
for MOVE LONG instruction 7-292
for MOVE LONG UNICODE instruction 7-299
for MOVE WITH OPTIONAL SPECIFICATIONS instruction 10-69
of TOD clock 4-47

X

X (EXCLUSIVE OR) instruction 7-254
X field of instruction 5-11
XA (extended architecture)
See 370-XA architecture
XC (EXCLUSIVE OR) instruction 7-255
examples A-20
XG (EXCLUSIVE OR) instruction 7-254
XGR (EXCLUSIVE OR) instruction 7-254
XI (EXCLUSIVE OR) instruction 7-255
example A-21
XIHF (EXCLUSIVE OR IMMEDIATE) instruction 7-256
XILF (EXCLUSIVE OR IMMEDIATE) instruction 7-256
XIY (EXCLUSIVE OR) instruction 7-255
XR (EXCLUSIVE OR) instruction 7-254
XRK (EXCLUSIVE OR) binary instruction 7-254
XSCH (CANCEL SUBCHANNEL) instruction 14-3
XY (EXCLUSIVE OR) instruction 7-254

Z

Z bit (zero condition-code bit) 16-12
as cause of intermediate interruption condition 16-18
z/Architecture architecture
additions to 1-2, 1-7
original highlights of 1-2
ZAP (ZERO AND ADD) instruction 8-14
example A-40
zero
instruction-length code 6-7
negative
See negative zero
normal meaning for byte value xxix
setting floating-point register to 9-35
true (HFP number) 18-1
zero-address-detection PER event
mask for 4-27
ZERO AND ADD instruction 8-14
example A-40
zero condition code (Z bit in SCSW) 16-12
zero-padding control 7-173
zone bits 8-1
moving of 7-303
zoned decimal numbers 8-1
zoned-decimal numbers
examples A-4

Communicating Your Comments to IBM

Title: z/Architecture Principles of Operation

Publication No: SA22-7832-11

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a reader's comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - FAX: (International Access Code)+1+845+432-9405
- If you prefer to send comments electronically, use one of these network IDs:
  - Internet e-mail: mhvrcfs@us.ibm.com
  - World Wide Web: http://www.ibm.com/servers/eserver/zseries/zos/webqs.html

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies

Optionally, if you include your telephone number, we will be able to respond to your comments by phone.

Reader's Comments — We'd Like to Hear from You

Title: z/Architecture Principles of Operation

Publication No: SA22-7832-11

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Today's date: _____

What is your occupation?

Newsletter number of latest Technical Newsletter (if any) concerning this publication:

How did you use this publication?

[ ] As an introduction

[ ] As a reference manual

[ ] For another purpose (explain)

[ ] As a text (student)

[ ] As a text (instructor)

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: Comment:

Name

Address

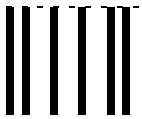
Company or Organization

Phone No.

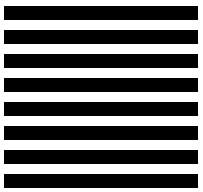
Cut or Fold
Along Line

Reader's Comments — We'd Like to Hear from You
SA22-7832-11

Fold and Tape **Please do not staple** Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY 12601-5400

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

SA22-7832-11