EREF 2.0: A Programmer’s Reference Manual for Freescale Power Architecture® Processors

Supports
e500 core family
(e500v1, e500v2, e500mc, e5500)
e200 core family

EREF_RM
Rev. 0
09/2011

freescale™

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
+1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Document Number: EREF_RM
Rev. 0, 09/2011

Contents

Paragraph Number | Title | Page Number

How to Use this Book 7
Audience 8
Organization 8
Additional Reading 9
Documentation Set 9
General Information 10
Conventions 10
Acronyms and Abbreviations 11
Terminology Conventions 12

Chapter 1 Overview

1.1 Power Architecture Technology and the Freescale Embedded Implementation Standards (EIS) 1-1
1.1.1 Categories 1-2
1.1.2 Power Architecture Evolution 1-3
1.1.3 Freescale Embedded Implementation Standards (EIS) 1-3
1.1.4 EIS Use of Power ISA Categories 1-4
1.1.4.1 Category Designations in EIS 1-5
1.1.4.2 Categories Supported by All Freescale EIS Processors 1-6
1.1.4.3 Categories Supported by Some Freescale EIS Processors 1-6
1.1.4.4 Categories Not Supported by Freescale EIS Processors 1-7
1.1.5 Special Instruction Changes From Book E 1-8
1.2 Programming Model 1-8
1.2.1 Privilege Model 1-8
1.2.2 Instruction Model 1-9
1.2.3 Register Model 1-10
1.2.4 Exceptions and Interrupt Handling 1-12
1.2.4.1 Exceptions 1-12
1.2.4.2 Interrupts 1-12
1.2.5 Memory (Storage) Model 1-13
1.2.5.1 Memory Addresses 1-13
1.2.5.2 Weakly Ordered Memory 1-14
1.2.5.3 Memory Management 1-14
1.2.5.4 Caches 1-15
1.3 Timers 1-16
1.4 Debug 1-16
1.5 Performance Monitoring 1-16
1.6 Major Additions to this Revision 1-16

Contents

Paragraph Number **Title** **Page Number**

Chapter 2
Computation Modes

2.1 Modes.. 2-1
2.2 64-bit Implementations <64>.. 2-1
2.3 32-bit Implementations.. 2-2

Chapter 3
Register Model

3.1 Overview—Register Model.. 3-1
3.2 Register Model.. 3-2
3.2.1 How Registers are Accessed.. 3-2
3.2.2 Special-Purpose Registers (SPRs) 3-4
3.2.2.1 SPR Register Mapping <E.HV>.. 3-12
3.3 Registers for Integer Operations.. 3-13
3.3.1 General-Purpose Registers (GPRs).. 3-13
3.3.2 Integer Exception Register (XER).. 3-14
3.4 Registers for Floating-Point Operations <FP> 3-15
3.4.1 Floating-Point Registers (FPR0–FPR31) 3-15
3.4.2 Floating-Point Status and Control Register (FPSCR).. 3-15
3.4.3 Floating-Point Data.. 3-18
3.4.3.1 Data Format 3-18
3.4.3.2 Value Representation 3-19
3.4.3.3 Sign of Result 3-21
3.4.3.4 Normalization and Denormalization.. 3-22
3.4.3.5 Data Handling and Precision 3-22
3.4.3.5.1 Single-Precision Operands 3-23
3.4.3.5.2 Integer-Valued Operands 3-24
3.4.3.6 Rounding.. 3-25
3.5 Registers for Branch Operations.. 3-26
3.5.1 Condition Register (CR) 3-26
3.5.1.1 CR Setting for Integer Instructions.. 3-28
3.5.1.2 CR Setting for Store Conditional Instructions.. 3-28
3.5.1.3 CR Setting for Floating-Point Instructions.. 3-28
3.5.1.4 CR Setting for Compare Instructions 3-29
3.5.1.5 CR Bit Settings in VLE Mode <VLE>.. 3-29
3.5.1.5.1 CR Settings for Integer Instructions in VLE <VLE>.. 3-30
3.5.1.5.2 CR Setting for Compare Instructions in VLE <VLE> 3-30
3.5.1.5.3 CR Setting for the VLE Bit Test Instruction <VLE> 3-31
3.5.2 Link Register (LR).. 3-31

Contents

Paragraph Number | Title | Page Number

3.5.2.1 Link Register Usage in VLE Mode <VLE> 3-31
3.5.3 Count Register (CTR) 3-32
3.5.3.1 Count Register Usage in VLE Mode <VLE> 3-32
3.5.4 Branch Unit Control and Status Register (BUCSR) 3-33
3.6 Processor Control Registers 3-33
3.6.1 Machine State Register (MSR) 3-33
3.6.2 Machine State Register Protect Register (MSRP) <E.HV> 3-37
3.6.3 Embedded Processor Control Register (EPCR) 3-39
3.6.4 Hardware Implementation-Dependent Registers 3-41
3.6.4.1 Hardware Implementation-Dependent Register 0 (HID0) 3-41
3.6.4.2 Hardware Implementation-Dependent Register 1 (HID1) 3-44
3.6.5 Processor ID Register (PIR) 3-44
3.6.6 Guest Processor ID Register (GPIR) <E.HV> 3-46
3.6.7 Processor Version Register (PVR) 3-47
3.6.8 System Version Register (SVR) 3-48
3.7 Timer Registers 3-48
3.7.1 Timer Control Register (TCR) 3-49
3.7.2 Timer Status Register (TSR) 3-51
3.7.3 Time Base (TB) 3-52
3.7.4 Decrementer Register (DEC) 3-53
3.7.5 Decrementer Auto-Reload Register (DECAR) 3-53
3.7.6 Alternate Time Base Registers (ATB) 3-54
3.8 Interrupt Registers 3-54
3.8.1 Save/Restore Registers 0 (SRR0, GSRR0, CSRR0, DSRR0, MCSRR0) 3-54
3.8.2 Save/Restore Register 1 (SRR1, GSRR1, CSRR1, DSRR1, MCSRR1) 3-56
3.8.3 Interrupt Vector Prefix Register (IVPR) 3-57
3.8.4 Guest Interrupt Vector Prefix Register (GIVPR) <E.HV> 3-57
3.8.5 Interrupt Vector Offset Registers (IVORs) 3-58
3.8.6 Guest Interrupt Vector Offset Registers (GIVORs) <E.HV> 3-59
3.8.7 Exception Syndrome Register (ESR) 3-60
3.8.8 Guest Exception Syndrome Register (GESR) <E.HV> 3-62
3.8.9 Data Exception Address Register (DEAR) 3-63
3.8.10 Guest Data Exception Address Register (GDEAR) <E.HV> 3-63
3.8.11 Machine Check Syndrome Register (MCSR) 3-64
3.8.12 Machine Check Address Register (MCAR/MCARU) 3-65
3.8.13 External Proxy Register (EPR) <EXP> 3-66
3.8.14 Guest External Proxy Register (GEPR) <EXP,E.HV> 3-67
3.9 Software-Use SPRs (SPRGs, GSPRGs<E.HV>, and USPRGs) 3-67
3.10 L1 Cache Registers 3-68
3.10.1 L1 Cache Control and Status Register 0 (L1CSR0) 3-68
3.10.2 L1 Cache Control and Status Register 1 (L1CSR1) 3-72

Contents

Paragraph Number | Title | Page Number

3.10.3 L1 Cache Control and Status Register 2 (L1CSR2) 3-75
3.10.4 L1 Cache Control and Status Register 3 (L1CSR3) 3-76
3.10.5 L1 Cache Control and Status Register 4 (L1CSR4) 3-76
3.10.6 L1 Cache Configuration Register 0 (L1CFG0) 3-77
3.10.7 L1 Cache Configuration Register 1 (L1CFG1) 3-79
3.10.8 L1 Flush and Invalidate Control Register 0 (L1FINV0) <DCF> 3-79
3.10.9 L1 Flush and Invalidate Control Register 1 (L1FINV1) <DCF> 3-80
3.11 L2 Cache Registers 3-81
3.11.1 L2 Configuration Register (L2CFG0) 3-81
3.11.2 L2 Cache Control and Status Register (L2CSR0) 3-82
3.11.3 L2 Cache Control and Status Register 1 (L2CSR1) 3-85
3.11.4 L2 Error Registers 3-86
3.11.4.1 L2 Cache Error Disable Register (L2ERRDIS) 3-88
3.11.4.2 L2 Cache Error Detect Register (L2ERRDET) 3-89
3.11.4.3 L2 Cache Error Interrupt Enable Register (L2ERRINTEN) 3-90
3.11.4.4 L2 Cache Error Control Register (L2ERRCTL) 3-91
3.11.4.5 L2 Cache Error Attribute Register (L2ERRATTR) 3-92
3.11.4.6 L2 Cache Error Address Capture Registers (L2ERRADDR, L2ERREADDR).... 3-92
3.11.4.7 L2 Cache Error Capture Data Registers (L2ECAPTDATALO, L2ECAPTDATAHI)... 3-93
3.11.4.8 L2 Cache Error Capture ECC Syndrome Register (L2CAPTECC) 3-94
3.11.4.9 L2 Cache Error Injection Control Register (L2ERRINJCTL) 3-94
3.11.4.10 L2 Cache Error Injection Mask Low and High Registers (L2ERRINJLO, L2ERRINJHI) 3-95
3.12 MMU Registers 3-96
3.12.1 Logical Partition ID Register (LPIDR) <E.HV> 3-96
3.12.2 Process ID Register (PID) 3-97
3.12.3 MMU Control and Status Register 0 (MMUCSR0) 3-98
3.12.4 MMU Configuration Register (MMUCFG) 3-99
3.12.5 TLB Configuration Registers (TLB*n*CFG) 3-100
3.12.6 MMU Assist Registers (MAS*n*) 3-102
3.12.6.1 MAS Register 0 (MAS0) 3-102
3.12.6.2 MAS Register 1 (MAS1) 3-104
3.12.6.3 MAS Register 2 (MAS2) 3-105
3.12.6.4 MAS Register 3 (MAS3) 3-106
3.12.6.5 MAS Register 4 (MAS4) 3-107
3.12.6.6 MAS Register 5 (MAS5) <E.HV> 3-108
3.12.6.7 MAS Register 6 (MAS6) 3-109
3.12.6.8 MAS Register 7 (MAS7) 3-110
3.12.6.9 MAS Register 8 (MAS8) <E.HV> 3-110
3.12.6.10 64-bit Access to MAS Register Pairs <64-bit, E.HV> 3-111

Contents

Paragraph Number | Title | Page Number

3.12.7 External PID Registers <E.PD> 3-112
3.12.7.1 External PID Load Context Register (EPLC) <E.PD> 3-112
3.12.7.2 External PID Store Context (EPSC) Register <E.PD> 3-113
3.13 Debug Registers 3-114
3.13.1 Debug Control Registers (DBCR0–DBCR6) 3-115
3.13.1.1 Debug Control Register 0 (DBCR0) 3-115
3.13.1.2 Debug Control Register 1 (DBCR1) 3-117
3.13.1.3 Debug Control Register 2 (DBCR2) 3-119
3.13.1.4 Debug Control Register 3 (DBCR3) 3-121
3.13.1.5 Debug Control Register 4 (DBCR4) 3-122
3.13.1.6 Debug Control Register 5 (DBCR5) 3-122
3.13.1.7 Debug Control Register 6 (DBCR6) 3-124
3.13.1.8 Debug External Control Register 0 (DBECR0) 3-125
3.13.2 Debug Counter Register (DBCNT) 3-125
3.13.3 Debug Status Register (DBSR/DBSRWR) 3-126
3.13.4 Instruction Address Compare Registers (IACn) 3-129
3.13.5 Data Address Compare Registers (DACn) 3-130
3.13.6 Data Value Compare Registers (DVC1 and DVC2) 3-130
3.13.7 Nexus and External Debug Related Registers 3-131
3.13.7.1 Nexus SPR Access Registers 3-131
3.13.7.1.1 Nexus SPR Configuration Register (NSPC) 3-131
3.13.7.1.2 Nexus SPR Data Register (NSPD) 3-132
3.13.7.2 Debug Event Select Register (DEVENT) 3-132
3.13.7.3 Debug Data Acquisition Message (DDAM) 3-133
3.13.7.4 Nexus Process ID Register (NPIDR) 3-133
3.14 Processor Management Registers 3-134
3.14.1 Core Device Control and Status Register 0 (CDCSR0) 3-134
3.15 Performance Monitor Registers (PMRs) <E.PM> 3-136
3.15.1 Global Control Register 0 (PMGC0/UPMGC0) 3-143
3.15.2 Local Control A Registers (PMLCan/UPMLCan) 3-143
3.15.3 Local Control B Registers (PMLCbn/UPMLCbn) 3-145
3.15.4 Performance Monitor Counter Registers (PMCn/UPMCn) 3-146
3.16 SPE Registers 3-147
3.17 AltiVec Registers 3-148
3.18 Device Control Registers (DCRs) 3-148

Chapter 4
Instruction Model

4.1 Operand Conventions 4-1
4.1.1 Data Organization in Memory and Data Transfers 4-1

Contents

Paragraph Number | Title | Page Number

4.1.2 Alignment and Misaligned Accesses 4-1
4.1.3 Atomic Accesses 4-2
4.2 Instruction Set Characteristics 4-3
4.3 Classes of Instructions 4-5
4.3.1 Defined Instruction Class 4-5
4.3.2 Illegal Instruction Class 4-5
4.3.3 Reserved Instruction Class 4-6
4.3.4 Reserved Fields and Reserved Values 4-6
4.4 Forms of Defined Instructions 4-7
4.4.1 Preferred Instruction Forms 4-7
4.4.2 Invalid Instruction Forms 4-7
4.5 Instruction-Related Exceptions 4-8
4.5.1 Boundedly Undefined 4-9
4.5.2 Instruction Forms 4-10
4.5.2.1 Preferred Instruction Forms (no-op) 4-10
4.5.2.2 Invalid Instruction Forms 4-10
4.5.3 Addressing Modes 4-10
4.5.3.1 Memory Addressing 4-11
4.5.3.2 Memory Operands 4-11
4.5.3.3 Effective Address Calculation 4-11
4.5.3.3.1 Data Memory Addressing Modes 4-11
4.5.3.3.2 Instruction Memory Addressing Modes 4-12
4.5.3.4 Byte Ordering 4-12
4.5.4 Synchronization 4-13
4.5.4.1 Memory Synchronization 4-13
4.5.4.2 Instruction Synchronization 4-14
4.5.4.2.1 Self-Modifying Code 4-15
4.5.4.3 Synchronization Requirements 4-16
4.5.4.4 Context Synchronization 4-19
4.5.4.5 Execution Synchronization 4-20
4.6 Instruction Summary 4-20
4.6.1 User-Level Instructions 4-21
4.6.1.1 Integer Instructions 4-21
4.6.1.1.1 Integer Arithmetic Instructions 4-22
4.6.1.1.2 Integer Compare Instructions 4-23
4.6.1.1.3 Integer Logical Instructions 4-23
4.6.1.1.4 Integer Rotate and Shift Instructions 4-24
4.6.1.2 Load and Store Instructions 4-25
4.6.1.2.1 Load and Store Address Generation 4-26
4.6.1.2.2 Update Forms of Load and Store Instructions 4-28
4.6.1.2.3 Integer Load Instructions 4-28

Contents

Paragraph Number	Title	Page Number
4.6.1.2.4	Integer Store Instructions	4-29
4.6.1.2.5	Integer Load and Store with Byte-Reverse Instructions	4-29
4.6.1.2.6	Integer Load and Store Multiple Instructions	4-30
4.6.1.2.7	Floating-Point Load and Store Instructions <FP>	4-30
4.6.1.2.8	Load and Reserve and Store Conditional Instructions	4-33
4.6.1.2.9	Decorated Load and Store Instructions <DS>	4-34
4.6.1.3	Floating-Point Instructions <FP>	4-35
4.6.1.3.1	Floating-Point Move Instructions	4-36
4.6.1.3.2	Floating-Point Arithmetic Instructions	4-37
4.6.1.3.3	Floating-Point Multiply-Add Instructions	4-37
4.6.1.4	Floating-Point Rounding and Conversion Instructions	4-38
4.6.1.5	Floating-Point Compare Instructions	4-39
4.6.1.6	Floating-Point Status and Control Register Instructions	4-39
4.6.1.7	Floating-Point Exceptions	4-40
4.6.1.7.1	Invalid Operation Exception	4-43
4.6.1.7.2	Zero Divide Exception	4-45
4.6.1.7.3	Overflow Exception	4-46
4.6.1.7.4	Underflow Exception	4-46
4.6.1.7.5	Inexact Exception	4-47
4.6.1.8	Branch and Flow Control Instructions	4-48
4.6.1.8.1	Branch Instruction Address Calculation	4-48
4.6.1.8.2	Branch Relative Addressing Mode	4-48
4.6.1.8.3	Branch Conditional to Relative Addressing Mode	4-49
4.6.1.8.4	Branch to Absolute Addressing Mode	4-50
4.6.1.8.5	Branch Conditional to Absolute Addressing Mode	4-50
4.6.1.8.6	Branch Conditional to Link Register Addressing Mode	4-50
4.6.1.8.7	Branch Conditional to Count Register Addressing Mode	4-51
4.6.1.9	Conditional Branch Control	4-51
4.6.1.9.1	Using a Link Register Stack	4-54
4.6.1.9.2	Branch Instructions	4-55
4.6.1.9.3	Condition Register (CR) Logical Instructions	4-55
4.6.1.9.4	Trap Instructions	4-55
4.6.1.9.5	Integer Select Instruction	4-56
4.6.1.10	System Linkage Instruction	4-57
4.6.1.11	Embedded Hypervisor Privilege Instruction <E.HV>	4-58
4.6.1.12	Debug Instruction	4-58
4.6.1.13	Processor Control Instructions	4-58
4.6.1.13.1	Move to/from Condition Register Instructions	4-58
4.6.1.13.2	Move to/from Special-Purpose Register Instructions	4-59
4.6.1.13.3	Wait for Interrupt Instruction	4-59
4.6.1.14	Performance Monitor Instructions	4-59

Contents

Paragraph Number | Title | Page Number

4.6.1.15 Memory Synchronization Instructions 4-60
4.6.1.16 Atomic Update Primitives Using Load and Reserve and Store Conditional instructions 4-61
4.6.1.16.1 Reservations........ 4-62
4.6.1.16.2 Forward Progress **(MOVE TO MEM CHAP)........ 4-63**
4.6.1.16.3 Reservation Loss Due to Granularity **(MOVE TO MEM CHAP)........ 4-63**
4.6.1.17 Cache Management Instructions........ 4-64
4.6.1.17.1 Cache Target Identifiers........ 4-64
4.6.1.17.2 User-Level Cache Management Instructions........ 4-65
4.6.1.17.3 Cache Locking Instructions <E.CL> 4-67
4.6.2 Hypervisor- and Supervisor-Level Instructions........ 4-68
4.6.2.1 System Linkage and MSR Access Instructions 4-68
4.6.2.2 Supervisor-Level Memory Control Instructions........ 4-70
4.6.2.2.1 Supervisor-Level Cache Management Instructions........ 4-70
4.6.2.2.2 TLB Management Instructions........ 4-70
4.6.2.3 External PID Instructions <E.PD> 4-72
4.6.2.4 Hypervisor-Level Messaging Instructions <E.PC>........ 4-73
4.6.2.4.1 Sending and Receiving Messages........ 4-74
4.6.2.5 Performance Monitor Instructions........ 4-76
4.6.2.6 Supervisor Level Device Control Register Instructions <E.DC>........ 4-76
4.6.3 Recommended Simplified Mnemonics........ 4-76
4.7 Instruction Listing........ 4-76

Chapter 5
Instruction Set

5.1 Notation 5-1
5.1.1 Instruction Bit Numbering........ 5-2
5.2 Instruction Fields 5-3
5.3 Description of Instruction Operations........ 5-5
5.4 Instruction Set 5-11

Chapter 6
Cache and MMU Architecture

6.1 **Overview 6-2**
6.2 Cache Model 6-3
6.2.1 Storage Attributes and Coherency 6-4
6.2.2 Cache Identifiers and CT Values 6-4
6.2.3 Cache Control Instructions 6-5
6.3 Special Cache Management Functionality 6-6

Contents

Paragraph Number | Title | Page Number

6.3.1 Cache Line Locking <E.CL> ... 6-6
6.3.1.1 Cache Line Lock Set, Query, and Clear Instructions ... 6-6
6.3.1.2 Error Conditions ... 6-7
6.3.1.2.1 Overlocking ... 6-8
6.3.1.2.2 Unable to Lock Conditions ... 6-8
6.3.1.2.3 Mode Restricted Cache Locking Exceptions ... 6-8
6.3.2 Direct Cache Flushing <DCF> ... 6-9
6.3.3 Cache Way Partitioning <CWP> ... 6-10
6.3.3.1 Interaction with Cache Locking Operations ... 6-10
6.4 Memory and Cache Coherency ... 6-10
6.4.1 Memory/Cache Access Attributes (WIMGE Bits) ... 6-11
6.4.2 Write-Through-Required Attribute (W) ... 6-12
6.4.3 Caching-Inhibited Attribute (I) ... 6-13
6.4.4 Memory Coherence–Required Attribute (M) ... 6-13
6.4.5 Guarded Attribute (G) ... 6-14
6.4.5.1 Definition of Speculative and Out-of-Order Memory Accesses ... 6-15
6.4.5.2 Performing Operations Speculatively ... 6-15
6.4.5.2.1 Speculative Accesses to Guarded Memory ... 6-16
6.4.5.2.2 Instruction Accesses: Guarded Memory and No-Execute Memory ... 6-16
6.4.6 Byte-Ordering (Endianness) Attribute (E) ... 6-16
6.4.6.1 Big-Endian Pages and Multiple-Byte Scalars ... 6-16
6.4.6.2 Little-Endian Pages and Multiple-Byte Scalars ... 6-16
6.4.6.3 Structure Mapping Examples ... 6-17
6.4.7 Mismatched Memory Cache Attributes ... 6-18
6.4.7.1 Coherency Paradoxes and WIMGE ... 6-19
6.4.7.2 Self-Modifying Code ... 6-19
6.4.8 Shared Memory ... 6-19
6.4.8.1 Memory Access Ordering ... 6-20
6.4.8.2 Architecture Ordering Requirements ... 6-21
6.4.8.3 Forcing Load and Store Ordering (Memory Barriers) ... 6-22
6.4.8.4 Architectural Memory Access Ordering ... 6-23
6.4.8.5 Memory Barrier Example ... 6-24
6.4.8.5.1 Programming Considerations ... 6-25
6.4.8.5.2 Programming Examples ... 6-27
6.4.8.6 Lock Acquisition and Import Barriers ... 6-27
6.4.8.6.1 Acquire Lock and Import Shared Memory ... 6-27
6.4.8.6.2 Acquire Lock and Import Shared Memory Using sync and mbar ... 6-28
6.4.8.6.3 Obtain Pointer and Import Shared Memory ... 6-29
6.4.8.7 Lock Release and Export Barriers ... 6-29
6.4.8.7.1 Export Shared Memory and Release Lock ... 6-30
6.4.8.7.2 Export Shared Memory and Release Lock Using sync or mbar ... 6-30

Contents

Paragraph Number	Title	Page Number
6.5	MMU Architecture	6-30
6.5.1	MMU Programming Model	6-32
6.5.2	Virtual Address (VA)	6-34
6.5.2.1	Address Space (AS) Value	6-35
6.5.2.2	Process ID (PID) Value	6-36
6.5.2.3	GS Value <E.HV>	6-37
6.5.2.4	LPID Value <E.HV>	6-38
6.5.3	TLB Concept	6-38
6.5.3.1	TLB Entries	6-41
6.5.3.2	TLB Entry Page Size	6-42
6.5.3.3	Reading and Writing TLB Entries	6-42
6.5.3.4	TLB Management using the MAS Registers	6-43
6.5.3.4.1	MAS Registers	6-43
6.5.3.5	Reading TLB Entries	6-43
6.5.3.6	Writing TLB Entries	6-44
6.5.3.6.1	Searching TLB Entries	6-46
6.5.3.7	Invalidating TLB Entries	6-46
6.5.3.7.1	Invalidation of a Single TLB Entry by Clearing MAS1[V]	6-47
6.5.3.7.2	**Invalidations Using tlbivax**	**6-47**
6.5.3.7.3	**Invalidations Using tlbilx <E.HV>**	**6-48**
6.5.3.7.4	Invalidate all Using MMUCSR0	6-49
6.5.3.7.5	Generous Invalidates and Invalidation Protection	6-49
6.5.4	Address Translation	6-50
6.5.4.1	Match Criteria for TLB Entries	6-52
6.5.4.2	Real Address Generation after a TLB Match	6-54
6.5.4.3	Page Size and Effective Address Bits Compared	6-54
6.5.4.4	Permission Attribute Comparison	6-55
6.5.4.5	Translation Algorithms	6-55
6.5.5	Access Control	6-56
6.5.5.1	Page Size and Real Address Generation	6-57
6.5.6	Permission Attributes	6-57
6.5.6.1	Execute Access Permission	6-58
6.5.6.2	Read Access Permission	6-58
6.5.6.3	Write Access Permission	6-58
6.5.6.4	Permission Control and Cache Management Instructions	6-59
6.5.6.5	Use of Permissions to Maintain Page History	6-60
6.5.7	Crossing Page Boundaries	6-60
6.5.8	MMU Exception Handling	6-61
6.5.8.1	TLB Miss Exception Types	6-61
6.5.8.1.1	Instruction TLB Error Interrupt Settings	6-61
6.5.8.1.2	Data TLB Error Interrupt Settings	6-61

Contents

Paragraph Number | Title | Page Number

6.5.8.1.3 TLB Miss Exception MAS Register Settings 6-61
6.5.8.2 Permissions Violation Exception Types 6-62
6.5.8.2.1 Virtualization Fault Exceptions <E.HV> 6-62
6.5.8.2.2 Instruction Storage Interrupt Settings 6-62
6.5.8.2.3 Data Storage Interrupt Settings 6-62
6.5.8.3 **MAS Register Updates for Exceptions, tlbsx, and tlbre 6-63**
6.5.9 MMU Configuration Information 6-66

Chapter 7
Interrupts and Exceptions

7.1 Overview 7-1
7.2 Interrupt Classes 7-2
7.2.1 Recoverability from Interrupts 7-2
7.3 Interrupt Registers 7-3
7.4 Directed Interrupts <E.HV> 7-7
7.5 Exceptions 7-8
7.6 Synchronous and Asynchronous Interrupts 7-10
7.6.1 Requirements for System Reset Generation 7-11
7.7 Interrupt Processing 7-11
7.8 Interrupt Definitions 7-13
7.8.1 Critical Input Interrupt 7-18
7.8.2 Machine Check, NMI, and Error Report Interrupts 7-18
7.8.2.1 General Machine Check, Error Report, and NMI Mechanism 7-19
7.8.2.2 NMI Interrupts 7-21
7.8.2.3 Error Report Synchronous Interrupts 7-22
7.8.2.4 Asynchronous Machine Check Interrupts 7-23
7.8.3 Data Storage Interrupt 7-23
7.8.4 Instruction Storage Interrupt 7-25
7.8.5 External Input Interrupt 7-27
7.8.5.1 External Proxy <EXP> 7-28
7.8.6 Alignment Interrupt 7-29
7.8.7 Program Interrupt 7-30
7.8.8 Floating-Point Unavailable Interrupt <FP> 7-32
7.8.9 System Call Interrupt 7-32
7.8.10 Decrementer Interrupt 7-33
7.8.11 Fixed-Interval Timer Interrupt 7-34
7.8.12 Watchdog Timer Interrupt 7-35
7.8.13 Data TLB Error Interrupt 7-38
7.8.14 Instruction TLB Error Interrupt 7-39
7.8.15 Debug Interrupt 7-40

Contents

Paragraph Number | Title | Page Number

7.8.16 SPE/Embedded Floating-Point/AltiVec Unavailable Interrupt <SP or V> 7-42
7.8.17 Embedded Floating-Point Data Interrupt <SP.FD, SP.FS, SP.FV> 7-43
7.8.18 AltiVec Assist Interrupt <V> 7-44
7.8.19 Embedded Floating-Point Round Interrupt <SP.FD, SP.FS, SP.FV> 7-45
7.8.20 Performance Monitor Interrupt <E.PM> 7-46
7.8.21 Processor Doorbell Interrupt <E.PC> 7-47
7.8.22 Processor Doorbell Critical Interrupt <E.PC> 7-47
7.8.23 Guest Processor Doorbell Interrupt <E.HV> 7-48
7.8.24 Guest Processor Doorbell Critical Interrupt <E.HV> 7-49
7.8.25 Guest Processor Doorbell Machine Check Interrupt <E.HV> 7-49
7.8.26 Embedded Hypervisor System Call Interrupt <E.HV> 7-50
7.8.27 Embedded Hypervisor Privilege Interrupt <E.HV> 7-51
7.8.28 LRAT Error Interrupt <E.HV.LRAT> 7-51
7.9 Partially Executed Instructions 7-52
7.10 Interrupt Ordering and Masking 7-53
7.10.1 Guidelines for System Software 7-54
7.11 Exception Priorities 7-55
7.12 Processor Signaling (msgsnd and msgclr) <E.PC> 7-59
7.12.1 Sending Messages 7-59
7.12.2 Receiving, Filtering, and Accepting Messages 7-61
7.12.3 Message Types and Definitions 7-61

Chapter 8
Timer Facilities

8.1 Timer Facilities 8-1
8.2 Timer Registers 8-2
8.3 Time Base Operation 8-3
8.3.1 Writing the TB 8-3
8.3.2 Reading the TB 8-4
8.3.3 Computing Time of Day from the Time Base 8-4
8.3.4 Nonconstant Update Frequency 8-6
8.4 Decrementer Operation 8-6
8.5 Fixed-Interval Timer 8-7
8.6 Watchdog Timer 8-8
8.7 Alternate Time Base 8-10
8.8 Freezing the Timer Facilities 8-10

Contents

Paragraph Number | Title | Page Number

Chapter 9
Debug Facilities

9.1 Overview 9-1
9.2 Internal and External Debug 9-1
9.3 Programming Model 9-2
9.3.1 Debug Registers 9-2
9.3.2 Debug Instructions 9-2
9.3.3 Debug Interrupt 9-2
9.4 Enabling Internal Debug Mode 9-2
9.5 External Debug Mode 9-3
9.6 Dealing with Debug Resources 9-3
9.6.1 Synchronizing Changes to Debug Facility Registers 9-3
9.6.2 Pre-completion and Post-completion Interrupts 9-4
9.6.2.1 Synchronous Pre-completion Debug Interrupts 9-4
9.6.2.2 Synchronous Post-completion Debug Interrupts 9-5
9.6.2.3 Asynchronous Debug Interrupts 9-5
9.7 Debug Conditions and Debug Events 9-5
9.7.1 Suppressing Debug Events in Hypervisor Mode 9-6
9.7.2 Instruction Address Compare Debug Event 9-6
9.7.3 Data Address Compare Debug Event 9-8
9.7.4 Trap Debug Event 9-11
9.7.5 Branch Taken Debug Event 9-12
9.7.6 Instruction Complete Debug Event 9-12
9.7.7 Interrupt Taken Debug Event 9-13
9.7.8 Critical Interrupt Taken Debug Event <E.ED> 9-14
9.7.9 Return Debug Event 9-14
9.7.10 Critical Interrupt Return Debug Event <E.ED> 9-15
9.7.11 Unconditional Debug Event (UDE) 9-16
9.7.12 DAC and IAC Linking 9-16

Chapter 10
Performance Monitor <E.PM>

10.1 Overview 10-1
10.1.1 Performance Monitor Programming Model 10-2
10.1.1.1 Performance Monitor Interrupt Model 10-3
10.2 Performance Monitor Example 10-4
10.2.1 Event Counting 10-4
10.2.2 Processor Context Configurability 10-4
10.2.3 Event Selection 10-6

Contents

Paragraph Number | Title | Page Number

10.2.4 Chaining Counters 10-6
10.2.5 Thresholds 10-7

Appendix A
Instruction Set Listings

A.1 Instructions Sorted by Mnemonic (Decimal) A-1
A.2 Instructions Sorted by Opcodes (Decimal) A-29
A.3 Instructions Sorted by Mnemonic (Binary) A-57
A.4 Instructions Sorted by Opcode (Binary) A-83

Appendix B
Simplified Mnemonics

B.1 Overview B-1
B.2 Subtract Simplified Mnemonics B-1
B.2.1 Subtract Immediate B-1
B.2.2 Subtract B-2
B.3 Rotate and Shift Simplified Mnemonics B-2
B.3.1 Operations on Words B-2
B.3.2 Operations on Double-words B-3
B.4 Branch Instruction Simplified Mnemonics B-4
B.4.1 Key Facts about Simplified Branch Mnemonics B-5
B.4.2 Eliminating the BO Operand B-6
B.4.3 Incorporating the BO Branch Prediction B-7
B.4.4 The BI Operand—CR Bit and Field Representations B-8
B.4.4.1 BI Operand Instruction Encoding B-9
B.4.5 Simplified Mnemonics that Incorporate the BO Operand B-11
B.4.5.1 Examples that Eliminate the BO Operand B-12
B.4.6 Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with **cr**S) B-15
B.4.6.1 Branch Simplified Mnemonics that Incorporate CR Conditions: Examples B-17
B.4.6.2 Branch Simplified Mnemonics that Incorporate CR Conditions: Listings B-17
B.5 Compare Word Simplified Mnemonics B-19
B.6 Compare Double-word Simplified Mnemonics B-20
B.7 Condition Register Logical Simplified Mnemonics B-21
B.8 Trap Instructions Simplified Mnemonics B-21
B.9 Simplified Mnemonics for Accessing SPRs B-23
B.10 AltiVec Simplified Mnemonics B-24

Contents

Paragraph Number | **Title** | **Page Number**

B.11 SPE Simplified Mnemonics....B-24
B.12 Recommended Simplified Mnemonics....B-24
B.12.1 No-Op (**nop**)....B-24
B.12.2 Load Immediate (**li**)....B-25
B.12.3 Load Address (**la**)....B-25
B.12.4 Move Register (**mr**)....B-25
B.12.5 Complement Register (**not**)....B-25
B.12.6 Move to Condition Register (**mtcr**)....B-25
B.12.7 Sync (**sync**)....B-26
B.12.8 Integer Select (**isel**)....B-26
B.12.9 System Call (**sc**)....B-26
B.12.10 TLB Invalidate Local Indexed....B-26

Appendix C
Programming Examples

C.1 Synchronization....C-1
C.1.1 Synchronization Primitives....C-2
C.1.1.1 Memory Barriers....C-2
C.1.1.2 Fetch and No-op....C-2
C.1.1.3 Fetch and Store....C-3
C.1.1.4 Fetch and Add....C-3
C.1.1.5 Fetch and AND....C-3
C.1.1.6 Test and Set....C-3
C.1.1.7 Compare and Swap....C-4
C.1.2 Lock Acquisition and Release....C-4
C.1.3 List Insertion....C-5
C.1.4 Synchronization Notes....C-6
C.2 Multiple-Precision Shifts....C-7
C.3 Floating-Point Conversions <FP>....C-9
C.3.1 Conversion from a Floating-Point Number to Floating-Point Integer....C-9
C.3.2 Conversion from Floating-Point Number to Signed Integer Doubleword <64>....C-10
C.3.3 Conversion from Floating-Point Number to Unsigned Integer Doubleword <64>....C-10
C.3.4 Conversion from Floating-Point Number to Signed Integer Word....C-10
C.3.5 Conversion from Floating-Point Number to Unsigned Integer Word....C-10
C.3.6 Conversion from Signed Integer Doubleword to Floating-Point Number <64>....C-11
C.3.7 Conversion from Unsigned Integer Doubleword to Floating-Point Number <64>....C-11
C.3.8 Conversion from Signed Integer Word to Floating-Point Number <64>....C-12
C.3.9 Conversion from Unsigned Integer Word to Floating-Point Number <64>....C-12
C.4 Floating-Point Selection....C-12
C.4.1 Notes....C-13

Contents

Paragraph Number | Title | Page Number

Appendix D
Floating-Point Models

D.1 Execution Model for IEEE-754 Operations.... D-1
D.2 Multiply-Add Type Instruction Execution Model D-3
D.3 Floating-Point Conversions D-4
D.3.1 Conversion from Floating-Point Number to Signed Fixed-Point Integer Word.... D-5
D.3.2 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word D-5
D.4 Floating-Point Models D-5
D.4.1 Floating-Point Round to Single-Precision Model.... D-5
D.4.2 Floating-Point Convert to Integer Model.... D-9
D.4.3 Floating-Point Convert from Integer Model.... D-12
D.5 Floating-Point Selection D-13
D.5.1 Comparison to Zero D-13
D.5.2 Minimum and Maximum.... D-13
D.5.3 Simple If-Then-Else Constructions D-14
D.5.4 Notes.... D-14
D.6 Floating-Point Load Instructions D-14
D.7 Floating-Point Store Instructions.... D-15

Figures

Figure Number | Title | Page Number

1-1 Power ISA 1-2
1-2 Power Architecture Relationships 1-3
1-3 Extended GPRs as Implemented to Support SPE Instructions 1-4
3-1 Integer Exception Register (XER) 3-14
3-2 Floating-Point Status and Control Register (FPSCR) 3-15
3-3 Floating-Point Single Format 3-19
3-4 Floating-Point Double Format 3-19
3-5 Approximation to Real Numbers 3-20
3-6 Selection of Z1 and Z2 3-26
3-7 Condition Register (CR) 3-26
3-8 Link Register (LR) 3-31
3-9 Count Register (CTR) 3-32
3-10 Branch Unit Control and Status Register (BUCSR) 3-33
3-11 Machine State Register (MSR) 3-34
3-12 Machine State Register Protect (MSRP) Format 3-38
3-13 Embedded Processor Control Register (EPCR) 3-39
3-14 Hardware Implementation-Dependent Register 0 (HID0) 3-42
3-15 Hardware Implementation-Dependent Register 1 (HID1) 3-44
3-16 Processor ID Register (PIR) 3-45
3-17 Guest Processor ID Register (GPIR) 3-46
3-18 Processor Version Register (PVR) 3-47
3-19 System Version Register (SVR) 3-48
3-20 Timer Control Register (TCR) 3-49
3-21 Timer Status Register (TSR) 3-51
3-22 Time Base Upper/Lower Registers (TBU/TBL) 3-52
3-23 Decrementer Register (DEC) 3-53
3-24 Decrementer Auto-Reload Register (DECAR) 3-54
3-25 Alternate Time Base Register Upper/Lower (ATBU/ATBL) 3-54
3-26 Save/Restore Register 0 (*x*SRR0) 3-55
3-27 Save/Restore Register 1(*x*SRR1) 3-56
3-28 Interrupt Vector Prefix Register (IVPR) 3-57
3-29 Guest Interrupt Vector Prefix Register (GIVPR) Format 3-57
3-30 Interrupt Vector Offset Registers (IVOR) 3-58
3-31 Guest Interrupt Vector Offset Registers G(IVOR) 3-59
3-32 Exception Syndrome Register (ESR) 3-60
3-33 Guest Exception Syndrome Register (ESR) 3-62
3-34 Data Exception Address Register (DEAR) 3-63
3-35 Data Exception Address Register (DEAR) 3-64
3-36 Machine Check Syndrome Register 1 (MCSR) 3-65
3-37 Machine Check Address Register (MCAR/MCARU) 3-66
3-38 External Proxy Register (EPR) 3-66

Figures

Figure Number | Title | Page Number

3-39 Guest External Proxy Register (GEPR) 3-67
3-40 Software-Use SPRs (SPRGs, GSPRGs, and USPRG0) 3-68
3-41 L1 Cache Control and Status Register 0 (L1CSR0) 3-69
3-42 L1 Cache Control and Status Register 1 (L1CSR1) 3-73
3-43 L1 Cache Control and Status Register 2 (L1CSR2) 3-75
3-44 L1 Cache Control and Status Register 3 (L1CSR3) 3-76
3-45 L1 Cache Control and Status Register 4 (L1CSR4) 3-77
3-46 L1 Cache Configuration Register 0 (L1CFG0) 3-78
3-47 L1 Cache Configuration Register 1 (L1CFG1) 3-79
3-48 L1 Flush and Invalidate Control Register 0 (L1FINV0) 3-80
3-49 L1 Flush and Invalidate Control Register 0 (L1FINV0) 3-80
3-50 L2 Cache Configuration Register (L2CFG0) 3-81
3-51 L2 Cache Control and Status Register (L2CSR0) 3-83
3-52 L2 Cache Control and Status Register 1 (L2CSR1) 3-86
3-53 L2 Cache Error Disable Register (L2ERRDIS) 3-88
3-54 L2 Cache Error Detect Register (L2ERRDET) 3-89
3-55 L2 Cache Error Interrupt Enable Register (L2ERRINTEN) 3-90
3-56 L2 Cache Error Control Register (L2ERRCTL) 3-91
3-57 L2 Cache Error Attribute Register (L2ERRATTR) 3-92
3-58 L2 Cache Error Address Capture Register (L2ERRADDR) 3-93
3-59 L2 Cache Error Extended Address Capture Register (L2ERREADDR) 3-93
3-60 L2 Cache Error Capture Data Low Register (L2CAPTDATALO) 3-93
3-61 L2 Cache Error Capture Data High Register (L2CAPTDATAHI) 3-94
3-62 L2 Cache Error Capture ECC Syndrome Register (L2CAPTECC) 3-94
3-63 L2 Cache Error Injection Control Register (L2ERRINJCTL) 3-95
3-64 L2 Cache Error Injection Mask Low Register (L2ERRINJLO) 3-95
3-65 L2 Cache Error Injection Mask High Register (L2ERRINJHI) 3-95
3-66 L 3-96
3-67 L 3-96
3-68 Logical Partition ID Register (LPIDR) 3-97
3-69 Process ID Registers (PID, PID1, PID2) 3-97
3-70 MMU Control and Status Register 0 for MMU V1 (MMUCSR0) 3-98
3-71 MMU Configuration Register (MMUCFG) 3-99
3-72 TLB Configuration Register n (TLB0CFG) 3-100
3-73 MAS Register 0 for MMU V1 (MAS0) 3-102
3-74 MAS Register 1 (MAS1) Format 3-104
3-75 MAS Register 2 (MAS2) 3-105
3-76 MAS Register 3 (MAS3) 3-106
3-77 MAS Register 4 (MAS4) 3-107
3-78 MAS Register 5 (MAS5) 3-109
3-79 MAS Register 6 for MMU V1 (MAS6) 3-109

Figures

Figure Number **Title** **Page Number**

3-80 MAS Register 7 (MAS7) 3-110
3-81 MAS Register 8 (MAS8) 3-111
3-82 External PID Load Context (EPLC) Format 3-112
3-83 External PID Store Context (EPSC) Format 3-113
3-84 Debug Control Register 0 (DBCR0) 3-115
3-85 Debug Control Register 1 (DBCR1) 3-117
3-86 Debug Control Register 2 (DBCR2) 3-120
3-87 Debug Control Register 3 (DBCR3) 3-122
3-88 Debug Control Register 4 (DBCR4) 3-122
3-89 Debug Control Register 5 (DBCR5) 3-122
3-90 Debug Control Register 6 (DBCR6) 3-125
3-91 Debug External Control Register 0 (DBECR0) 3-125
3-92 Debug Counter Register (DBCNT) 3-125
3-93 Debug Status Register (DBSR/DBSRWR) 3-126
3-94 Instruction Address Compare Registers (IAC1–IAC8) 3-130
3-95 Data Address Compare Registers (DACn) 3-130
3-96 Data Value Compare Registers (DVC1–DVC2) 3-131
3-97 Nexus SPR Configuration Register (NSPC) 3-132
3-98 Nexus SPR Data Register (NSPD) 3-132
3-99 Debug Event Register (DEVENT) 3-133
3-100 Debug Data Acquisition Message (DDAM) 3-133
3-101 Nexus Process ID Register (NPIDR) 3-133
3-102 Core Device Control and Status Register 0 (CDCSR0) 3-134
3-103 CDCSR0 Device Field Format 3-135
3-104 Performance Monitor Global Control Register 0 (PMGC0)/
User Performance Monitor Global Control Register 0 (UPMGC0) 3-143
3-105 Local Control A Registers (PMLCan)/
User Local Control A Registers (UPMLCan) 3-144
3-106 Local Control B Registers (PMLCbn)/
User Local Control B Registers (UPMLCbn) 3-145
3-107 Performance Monitor Counter Registers (PMCn)/
User Performance Monitor Counter Registers (UPMCn) 3-146
4-1 Register Indirect with Immediate Index Addressing for Loads/Stores 4-26
4-2 Register Indirect with Index Addressing for Loads/Stores 4-27
4-3 Register Indirect Addressing for Integer Loads/Stores 4-27
4-4 Branch Relative Addressing 4-49
4-5 Branch Conditional Relative Addressing 4-49
4-6 Branch to Absolute Addressing 4-50
4-7 Branch Conditional to Absolute Addressing 4-50
4-8 Branch Conditional to Link Register Addressing 4-51
4-9 Branch Conditional to Count Register Addressing 4-51

Figures

Figure Number	Title	Page Number
4-10	BO Bit Descriptions	4-52
5-1	Instruction Description	5-11
5-2	**evlddepx** Results in Big- and Little-Endian Modes	5-85
5-3	**evstddepx** Results in Big- and Little-Endian Modes	5-86
6-1	Big-Endian Mapping of Structure S	6-17
6-2	Little-Endian Mapping of Structure S—Alternate View	6-18
6-3	Memory Barrier when Coherency is Required (M = 1)	6-24
6-4	Cumulative Memory Barrier	6-25
6-5	MMU Registers	6-32
6-6	Virtual Address Space	6-35
6-7	TLBs Accessed Through MAS Registers and TLB Instructions	6-43
6-8	EA Format for **tlbivax**	6-47
6-9	Virtual Address and TLB-Entry Comparison	6-53
6-10	Effective-to-Real Address Translation	6-54
6-11	Granting of Access Permission	6-55
7-1	Interrupt Registers	7-4
7-2	Watchdog State Machine	7-37
8-1	Time/Decrementer Registers Comparison	8-2
8-2	Relationship of Timer Facilities to Time Base	8-3
8-3	Decrementer Operation	8-6
8-4	Watchdog State Machine	8-9
10-1	Performance Monitor Registers	10-2

Tables

Table Number | Title | Page Number

1-1 Register sets in EIS 1-11
3-1 How Registers are Accessed 3-3
3-2 Special-Purpose Registers (by SPR Number) 3-5
3-3 Register Mapping in Guest Supervisor State 3-12
3-4 XER Field Descriptions 3-14
3-5 FPSCR Field Descriptions 3-16
3-6 Floating-Point Result Flags 3-18
3-7 IEEE-754 Floating-Point Fields 3-19
3-8 Rounding Mode Rules 3-25
3-9 BI Operand Settings for CR Fields 3-27
3-10 CR0 Bit Descriptions 3-28
3-11 CR Setting for Floating-Point Instructions 3-28
3-12 CR Setting for Compare Instructions 3-29
3-13 CR0 Encodings 3-30
3-14 Condition Register Setting for Compare Instructions 3-30
3-15 Branch to Link Register Instruction Comparison 3-31
3-16 Branch to Count Register Instruction Comparison 3-32
3-17 BUCSR Field Descriptions 3-33
3-18 MSR Field Descriptions 3-34
3-19 MSRP Field Descriptions 3-38
3-20 EPCR Field Descriptions 3-40
3-21 HID0 Field Descriptions 3-42
3-22 PIR Reset Value <E.PC> 3-45
3-23 PIR Access Privilege <E.HV> 3-46
3-24 GPIR Access Privilege 3-47
3-25 PVR Field Descriptions 3-47
3-26 TCR Field Descriptions 3-50
3-27 TSR Field Descriptions 3-51
3-28 IVOR Assignments 3-58
3-29 GIVOR Assignments 3-59
3-30 Exception Syndrome Register (ESR) Definition 3-60
3-31 MCSR Field Descriptions 3-65
3-32 L1CSR0 Field Descriptions 3-69
3-33 L1CSR1 Field Descriptions 3-73
3-34 L1CSR2 Field Descriptions 3-75
3-35 L1CSR3 Field Descriptions 3-76
3-36 L1CSR4 Field Descriptions 3-77
3-37 L1CFG0 Field Descriptions 3-78
3-38 L1CFG1 Field Descriptions 3-79
3-39 L1FINV0 Fields—L1 Direct Cache Flush 3-80
3-40 L1FINV0 Fields—L1 Direct Cache Flush 3-81

Tables

Table Number | Title | Page Number

3-41 L2CFG0 Field Descriptions 3-82
3-42 L2CSR0 Field Descriptions 3-83
3-43 L2CSR1 Field Descriptions 3-86
3-44 L2ERRDIS Field Descriptions 3-88
3-45 L2ERRDET Field Descriptions 3-89
3-46 L2ERRINTEN Field Descriptions 3-90
3-47 L2ERRCTL Field Descriptions 3-91
3-48 L2ERRATTR Field Descriptions 3-92
3-49 L2CAPTECC Field Descriptions 3-94
3-50 MMUCSR0 Field Descriptions 3-98
3-51 MMUCFG Field Descriptions 3-99
3-52 TLBnCFG Field Descriptions 3-101
3-53 MAS0 Field Descriptions 3-102
3-54 MAS1 Field Descriptions—Descriptor Context and Configuration Control 3-104
3-55 MAS2 Field Descriptions—EPN and Page Attributes 3-105
3-56 MAS3 Field Descriptions–RPNL and Access Control 3-107
3-57 MAS4 Field Descriptions—Hardware Replacement Assist Configuration 3-108
3-58 MAS5 Field Descriptions—Hypervisor Search PIDs 3-109
3-59 MAS 6 Field Descriptions—Search PIDs and Search AS 3-110
3-60 MAS 7 Field Descriptions—High Order RPNU 3-110
3-61 MAS8 Field Descriptions 3-111
3-62 MAS Register Pairs 3-111
3-63 EPLC Fields—External PID Load Context 3-112
3-64 EPSC Fields—External PID Store Context 3-114
3-65 DBCR0 Field Descriptions 3-115
3-66 DBCR1 Field Descriptions 3-118
3-67 DBCR2 Field Descriptions 3-120
3-68 DBCR5 Field Descriptions 3-123
3-69 DBSR Field Descriptions 3-127
3-70 IAC Bits Used to Compare to Instruction Address 3-129
3-71 NSPC Field Descriptions 3-132
3-72 CDCSR0 Fields 3-134
3-73 CDCSR0 Device Field Format Fields 3-135
3-74 CDCSR0 Device Architectural State 3-136
3-75 Behavior of CDCSR0 Devices Based on State 3-136
3-76 Freescale EIS PMRs 3-137
3-77 PMGC0/UPMGC0 Field Descriptions 3-143
3-78 PMLCan/UPMLCan Field Descriptions 3-144
3-79 PMLCbn/UPMLCbn Field Descriptions 3-146
3-80 PMCn/UPMCn Field Descriptions 3-147
4-1 Address Characteristics of Aligned Operands 4-2

Tables

Table Number	Title	Page Number
4-2	Data Access Synchronization Requirements	4-17
4-3	Instruction Fetch and/or Execution Synchronization Requirements	4-18
4-4	Special Synchronization Requirements	4-19
4-5	Integer Arithmetic Instructions	4-22
4-6	Integer Compare Instructions	4-23
4-7	Integer Logical Instructions	4-23
4-8	Integer Rotate Instructions	4-24
4-9	Integer Shift Instructions	4-25
4-10	Integer Load Instructions	4-28
4-11	Integer Store Instructions	4-29
4-12	Integer Load and Store with Byte-Reverse Instructions	4-30
4-13	Integer Load and Store Multiple Instructions	4-30
4-14	Floating-Point Load and Store Instructions	4-32
4-15	Load and Reserve and Store Conditional Instructions	4-33
4-16	Decorated Load and Store Instructions	4-35
4-17	Floating-Point Move Instructions	4-36
4-18	Floating-Point Arithmetic Instructions	4-37
4-19	Floating-Point Multiply-Add Instructions	4-38
4-20	Floating-Point Rounding and Conversion Instructions	4-38
4-21	CR Field Settings	4-39
4-22	Floating-Point Compare and Select Instructions	4-39
4-23	Floating-Point Status and Control Register Instructions	4-40
4-24	Power ISA FE0 and FE1 Configuration Descriptions	4-42
4-25	BO Operand Encodings	4-52
4-26	at Bit Encodings	4-52
4-27	BH Field Encodings	4-53
4-28	Branch Instructions	4-55
4-29	Condition Register Logical Instructions	4-55
4-30	Trap Instructions	4-56
4-31	Integer Select Instruction	4-56
4-32	Recoding with **isel**	4-57
4-33	System Linkage Instruction	4-57
4-34	Embedded Hypervisor Privilege Instruction	4-58
4-35	Debug Instruction	4-58
4-36	Move to/from Condition Register Instructions	4-59
4-37	Move to/from Special-Purpose Register Instructions	4-59
4-38	Wait for Interrupt Instruction	4-59
4-39	Performance Monitor Instructions	4-59
4-40	Memory Synchronization Instructions	4-60
4-41	User-Level Cache Instructions	4-65
4-42	Cache Locking Instructions	4-67

Tables

Table Number | Title | Page Number

4-43 Instruction Execution Based on Privilege Level 4-68
4-44 System Linkage Instructions—Supervisor-Level 4-69
4-45 Move to/from Machine State Register Instructions 4-69
4-46 Supervisor-Level Cache Management Instruction 4-70
4-47 TLB Management Instructions 4-71
4-48 External PID Load Store Instructions 4-73
4-49 Processor Message Types 4-74
4-50 Processor Message Types <E.HV> 4-75
4-51 Supervisor Performance Monitor Instructions 4-76
4-52 Device Control Register Instructions 4-76
4-53 Instruction Set 4-76
5-1 Notation Conventions 5-1
5-2 Instruction Field Descriptions 5-3
5-3 RTL Notation 5-6
5-4 Operator Precedence 5-10
5-5 BI Operand Settings for CR Fields 5-25
5-6 BI Operand Settings for CR Fields 5-27
5-7 BI Operand Settings for CR Fields 5-29
5-8 Operations with Special Values 5-110
5-9 Operations with Special Values 5-112
5-10 mfpmr Execution 5-190
5-11 mfspr Execution 5-191
5-12 mtpmr Execution 5-205
5-13 mtspr Execution 5-207
5-14 or rx,rx,rx Performance Hints 5-219
5-15 Extended Mnemonics for sync 5-313
5-16 Extended Mnemonics for tlbilx 5-318
6-1 CT Values and Target Cache 6-4
6-2 Load and Store Ordering 6-22
6-3 Architectural Memory Access Ordering 6-24
6-4 TLB Fields 6-39
6-5 TLB Page Sizes 6-42
6-6 Privilege Level of Executing tlbwe 6-45
6-7 Fields for EA Format of **tlbivax** 6-48
6-8 Real Address Generation 6-57
6-9 Permission Control for Instruction, Data Read, and Data Write Accesses 6-58
6-10 Permission Control and Cache Instructions 6-59
6-11 MMU Assist Register Field Updates 6-63
7-1 Interrupt Classes 7-2
7-2 Asynchronous and Synchronous Interrupts xSRR0 Setting 7-10
7-3 Interrupt Summary by IVOR 7-13

Tables

Table Number | Title | Page Number

7-4 Critical Input Interrupt Register Settings 7-18
7-5 Machine Check Interrupt Settings 7-21
7-6 Data Storage Interrupt Exception Conditions 7-23
7-7 Data Storage Interrupt Register Settings (directed to guest) 7-24
7-8 Data Storage Interrupt Register Settings (directed to hypervisor) 7-25
7-9 Instruction Storage Interrupt Exception Conditions 7-25
7-10 Instruction Storage Interrupt Register Settings (directed to guest) 7-26
7-11 Instruction Storage Interrupt Register Settings (directed to hypervisor) 7-26
7-12 External Input Interrupt Register Settings (directed to guest) 7-27
7-13 External Input Interrupt Register Settings (directed to hypervisor) 7-27
7-14 Alignment Interrupt Register Settings 7-30
7-15 Program Interrupt Exception Conditions 7-31
7-16 Program Interrupt Register Settings 7-31
7-17 Floating-Point Unavailable Interrupt Register Settings 7-32
7-18 System Call Interrupt Register Settings (directed to guest) 7-33
7-19 System Call Interrupt Register Settings (directed to hypervisor) 7-33
7-20 Decrementer Interrupt Register Settings 7-33
7-21 Fixed-Interval Timer Interrupt Register Settings 7-34
7-22 Watchdog Timer Interrupt Register Settings 7-35
7-23 Data TLB Error Interrupt Register Settings (directed to guest) 7-38
7-24 Data TLB Error Interrupt Register Settings (directed to hypervisor) 7-39
7-25 Instruction TLB Error Interrupt Register Settings (directed to guest) 7-39
7-26 Instruction TLB Error Interrupt Register Settings (directed to hypervisor) 7-40
7-27 Debug Interrupt Register Settings <E.ED> 7-41
7-28 Debug Interrupt Register Settings <E.ED> 7-42
7-29 SPE/Embedded Floating-Point/Vector Unavailable Interrupt Register Settings 7-43
7-30 Embedded Floating-Point Data Interrupt Register Settings 7-43
7-31 AltiVec Assist Interrupt Register Settings 7-44
7-32 Embedded Floating-Point Round Interrupt Register Settings 7-45
7-33 Performance Monitor Interrupt Register Settings 7-47
7-34 Processor Doorbell Interrupt Register Settings 7-47
7-35 Processor Doorbell Critical Interrupt Register Settings 7-48
7-36 Guest Processor Doorbell Interrupt Register Settings 7-48
7-37 Guest Processor Doorbell Critical Interrupt Register Settings 7-49
7-38 Guest Processor Doorbell Machine Check Interrupt Register Settings 7-50
7-39 Embedded Hypervisor System Call Interrupt Register Settings 7-50
7-40 Embedded Hypervisor Privilege Interrupt Register Settings 7-51
7-41 Embedded Hypervisor Privilege Interrupt Register Settings 7-52
7-42 Interrupt Hierarchy 7-54
7-43 Asynchronous Exception Priorities 7-57
7-44 Synchronous Exception Priorities 7-57

Tables

Table Number	Title	Page Number
7-45	msgsnd rB Field Descriptions	7-62
7-46	Processor Actions When a Message is Accepted	7-62
8-1	Watchdog Timer Controls	8-9
10-1	Processor States and PMLCan Bit Settings	10-5

About This Book

The primary objective of this reference is to provide software and hardware designers with the ability to design and program to the instruction set architecture (ISA) defined for embedded environment processors by the Power ISA™ and by Freescale’s implementation standards (EIS).

How to Use this Book

This book describes the resources defined for the Power ISA embedded environment. Although it provides a reference to all architecturally defined instructions implemented on Freescale Power ISA processors, the following supplementary volumes give detailed descriptions to the major extensions to the architecture:

- *AltiVec™ Technology Programming Environments Manual* (in this document referred to as the AltiVec PEM)
- *Signal Processing Engine (SPE) Programming Environments Manual* (in this document referred to as the SPE PEM)
- *Variable-Length Encoding (VLE) Programming Environments Manual* (in this document referred to as the VLE PEM)

Ordering information and for these documents can be found in Suggested Reading on page xxxi.

In particular, these books provide the full instruction descriptions in the format shown in Chapter 5, “Instruction Set,” as well as any additional information regarding how this functionality extends the register, interrupt, cache, and MMU models.

These books and the EREF together describe functionality at an architectural level, and they include indications of where implementation details may be left up to the designs of the individual cores and the integrated devices that implement them. For such critical details, the user must consult the following documentation:

- Core reference manuals—These books describe the features and behavior of individual microprocessor cores and provide specific information about how functionality described in the EREF is implemented by a particular core. They also describe implementation-specific features and micro-architectural details, such as instruction timing and cache hardware details, that lie outside the architecture specification.
- Integrated device reference manuals—These manuals describe the features and behavior of integrated devices that implement a Power ISA processor core. It is important to understand that some features defined for a core may not be supported on all devices that implement that core.

  Also, some features are defined in a general way at the core level and have meaning only in the context of how the core is implemented. For example, any implementation-specific behavior of register fields can be described only in the reference manual for the integrated device.

  Each of these documents include the following two chapters that are pertinent to the core:

— A core overview. This chapter provides a general overview of how the core works and indicates which of a core's features are implemented on the integrated device.

— A register summary chapter. This chapter gives the most specific information about how register fields can be interpreted in the context of the implementation.

These reference manuals also describe how the core interacts with other blocks on the integrated device, especially regarding topics such as reset, interrupt controllers, memory and cache management, debug, and global utilities.

Information in these books is subject to change without notice, as described in the disclaimers on the title page of this book, and documentation errata is made available through the Freescale website. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation.

Audience

It is assumed that the reader has the appropriate general knowledge regarding operating systems, microprocessor system design, and the basic principles of RISC processing to use the information in this manual.

Organization

Following is a summary and a brief description of the major parts of this reference manual:

- Chapter 1, "Overview," provides a general discussion of the programming, interrupt, cache, and memory management models as they are defined by the architecture.
- Chapter 2, "Computation Modes," provides a description of the computational modes for 32-bit and 64-bit computation.
- Chapter 3, "Register Model," is useful for software engineers who need to understand the programming model in general and the functionality of each register.
- Chapter 4, "Instruction Model," provides a description of the addressing modes and an overview of the instructions. Instructions are organized by function.
- Chapter 5, "Instruction Set," functions as a handbook for the instruction set. Instructions are sorted by mnemonic. Each instruction description includes the instruction formats and an individualized legend that provides such information as the level or levels of the architecture in which the instruction may be found and the privilege level of the instruction.
- Chapter 6, "Cache and MMU Architecture," focusses on how Freescale devices manage cache implementations and memory translation and protection. Note, however, that cache and MMU implementations vary from device to device, particularly with respect to the implementation of the caches and TLBs. For such information, refer to the core reference manuals.
- Chapter 7, "Interrupts and Exceptions," provides a description of the interrupts and exception conditions that can cause them.
- Chapter 8, "Timer Facilities," describes the architecture-defined timer resources: the time base, alternate time base, decrementer, watchdog timer, and fixed-interval timer.
- Chapter 9, "Debug Facilities."

- Chapter 10, “Performance Monitor <E.PM>.”

The following appendixes are also included:

- Appendix B, “Instruction Set Listings,” lists all instructions except those defined by the VLE extension instructions by both mnemonic and opcode, and includes a quick reference table with general information, such as the architecture level, privilege level, form, and whether the instruction is optional. VLE instruction opcodes are listed in the VLE PEM.
- Appendix C, “Simplified Mnemonics,” describes simplified mnemonics, which are provided for easier coding of assembly language programs. Simplified mnemonics are defined for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions defined by the Power ISA architecture and by implementations of and extensions to the Power ISA architecture.
- Appendix D, “Programming Examples,” shows how to use memory synchronization instructions to emulate various synchronization primitives and to provide more complex forms of synchronization. It also describes multiple precision shifts and floating-point conversions.
- Appendix E, “Floating-Point Models.”

This book also includes a glossary.

Suggested Reading

This section lists additional reading that provides background for the information in this manual as well as general information about the architecture.

Documentation Set

The following set of documents is provided to support Freescale embedded cores. These documents are available from the sources listed on the back cover of this manual; the document order numbers are included in parentheses for ease in ordering:

- Category-specific programming environments manuals. These books describe the three major extensions to the Power ISA embedded environment of the Power ISA. These include the following:
  — *AltiVec™ Technology Programming Environments Manual* (ALTIVECPEM)
  — *Signal Processing Engine (SPE) Programming Environments Manual: A Supplement to the EREF* (SPEPEM)
  — *Variable-Length Encoding (VLE) Programming Environments Manual: A Supplement to the EREF* (VLEPEM)
- Core reference manuals—These books describe the features and behavior of individual microprocessor cores and provide specific information about how functionality described in the EREF is implemented by a particular core. They also describe implementation-specific features and micro-architectural details, such as instruction timing and cache hardware details, that lie outside the architecture specification.
- Integrated device reference manuals—These manuals describe the features and behavior of integrated devices that implement a Power ISA processor core.

- Addenda/errata to reference manuals—When processors have follow-on parts, often an addendum is provided that describes the additional features and functionality changes. These addenda are intended for use with the corresponding reference manuals.
- Hardware specifications—Hardware specifications provide specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations.
- Technical summaries—Each device has a technical summary that provides an overview of its features. This document is roughly the equivalent to the overview (Chapter 1) of an implementation’s reference manual.
- Application notes—These short documents address specific design issues useful to programmers and engineers working with Freescale processors.

Additional literature is published as new processors become available. For a current list of documentation, refer to http://www.freescale.com.

General Information

The following documentation provides useful information about Power Architecture technology and computer architecture in general:

- *Power ISA™ Version 2.06,* by Power.org™, 2009, available at the Power.org website.
- *PowerPC Architecture Book,* by Brad Frey, IBM, 2005, available at the IBM website.
- *Computer Architecture: A Quantitative Approach*, Fourth Edition, by John L. Hennessy and David A. Patterson, Morgan Kaufmann Publishers, 2006.
- *Computer Organization and Design: The Hardware/Software Interface*, Third Edition, by David A. Patterson and John L. Hennessy, Morgan Kaufmann Publishers, 2007.

Conventions

This document uses the following notational conventions:

cleared/set	When a bit takes the value zero, it is said to be cleared; when it takes a value of one, it is said to be set.
mnemonics	Instruction mnemonics are shown in lowercase bold.
italics	Italics indicate variable command parameters, for example, **bcctr***x*.
	Book titles in text are set in italics.
	Internal signals are set in italics, for example, $\overline{\textit{qual BG}}$.
0x	Prefix to denote hexadecimal number
0b	Prefix to denote binary number
rA, **r**B	Instruction syntax used to identify what is typically a source GPR
rD	Instruction syntax used to identify a destination GPR
frA, **fr**B, **fr**C	Instruction syntax used to identify a source FPR
frD	Instruction syntax used to identify a destination FPR
vA, **v**B, **v**C	Instruction syntax used to identify a source VR

vD — Instruction syntax used to identify a destination VR

REG[FIELD] — Abbreviations for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register.

x — In some contexts, such as signal encodings, an unitalicized x indicates a don’t care.

x — An italicized *x* indicates an alphanumeric variable.

n — An italicized *n* indicates an numeric variable.

¬ — NOT logical operator

& — AND logical operator

| — OR logical operator

|| — Concatenation operator; for example, 010 || 111 is the same as 010111.

[—] — Indicates a reserved field in a register. Although these bits can be written to as ones or zeros, they are always read as zeros.

Additional conventions used with instruction encodings are described in Table 5-1.

Acronyms and Abbreviations

Table i contains acronyms and abbreviations that are used in this document.

Table i. Acronyms and Abbreviated Terms

Term	Meaning
DTLB	Data translation lookaside buffer
EA	Effective address
ECC	Error checking and correction
FPR	Floating-point register
FPU	Floating-point unit
GPR	General-purpose register
IEEE	Institute of Electrical and Electronics Engineers
ISA	Instruction set architecture
ITLB	Instruction translation lookaside buffer
L2	Secondary cache
LA	Logical Address
LIFO	Last-in-first-out
LR	Link register
LRU	Least recently used
LSB	Least-significant byte

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
lsb	Least-significant bit
MMU	Memory management unit
MSB	Most-significant byte
msb	Most-significant bit
MSR	Machine state register
NaN	Not a number
NIA	Next instruction address
No-op	No operation
OEA	Operating environment architecture
PTE	Page table entry
RISC	Reduced instruction set computing
RTL	Register transfer language
SIMM	Signed immediate value
SPE	Signal processing engine
SPR	Special-purpose register
TLB	Translation lookaside buffer
UIMM	Unsigned immediate value
UISA	User instruction set architecture
VA	Virtual address
VEA	Virtual environment architecture
VLE	Variable length encoding
VR	Vector register

Terminology Conventions

Table ii lists certain terms used in this manual that differ from the architecture terminology conventions.

Table ii. Terminology Conventions

The Architecture Specification	This Manual
Extended mnemonics	Simplified mnemonics
Privileged mode (or privileged state)	Supervisor level
Problem mode (or problem state)	User level
Real address	Physical address
Relocation	Translation

Table ii. Terminology Conventions (continued)

The Architecture Specification	This Manual
Out-of-order memory accesses	Speculative memory accesses
Storage (locations)	Memory
Storage (the act of)	Access

Table iii describes instruction field notation conventions used in this manual.

Table iii. Instruction Field Conventions

The Architecture Specification	Equivalent to:
BA, BB, BT	**crb**A, **crb**B, **crb**D (respectively)
BF, BFA	**crD**, **crf**S (respectively)
D	d
DS	ds
FLM	FM
FRA, FRB, FRC, FRT, FRS	**fr**A, **fr**B, **fr**C, **fr**D, **fr**S (respectively)
FXM	CRM
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S (respectively)
SI	SIMM
U	IMM
UI	UIMM
/, //, ///	0...0 (shaded)

Chapter 1
Overview

This document describes the programming resources common among those Freescale embedded processors that implement the Power ISA™ (embedded category) and the Freescale embedded implementation standards (EIS).

Note that the PowerPC™ architecture, as implemented on e600 and e300 cores, is described in the *Programming Environments Manual for 32-bit Implementations of the PowerPC™ Architecture.*

This chapter includes overviews of the following:

- Power Architecture® Technology and the Freescale Embedded Implementation Standards (EIS)
- EREF Programming Model
- Exceptions and Interrupt Handling
- Memory (Storage) Model
- Special Instruction Differences Between Book E and EREF

1.1 Power Architecture® Technology and the Freescale Embedded Implementation Standards (EIS)

The functionality described in this book was originally defined as the PowerPC Book E architecture and the Freescale EIS. It is now defined as the resources defined for the Base and Embedded categories by the Power ISA specification as well as for additional categories. Although the structure of the architecture specification has changed substantially, the functionality itself is essentially unchanged. The PowerPC architecture and the Power ISA now make up the Power Architecture technology.

1.1.1 Understanding Differences in the Book E and EREF Specification Structures

The reorganization makes the structure of the specification quite different from the PowerPC 1.10, Power ISA, and Book E architecture specifications.

Book I and Book II have been reorganized and amended, with features largely unchanged from Book E, the EIS, and IBM's PowerPC 2.02 specification for server processors. Although Book III-E and Book III-S still bear a family resemblance to the original PowerPC Book III definition, they differ in very significant ways both from one another and from the PowerPC 1.10 specification.

The Power ISA definition organizes the specification into shared Books I and II, which largely consists of the Base category (see Section 1.1.2, "Introducing the Power ISA Category"). Book III-S and Book III-E describe the resources for the privileged modes of the architecture and, although very similar in general appearance, contain significant differences. Book III-S is derived from PowerPC 2.02 and describes the privileged modes of the Server environment. Book III-E is derived from an unpublished version of Book E (which redefined the 64-bit instruction set to be compatible with PowerPC 2.02) and describes the privileged modes of the Embedded environment.

This figure shows the structure of Power ISA.

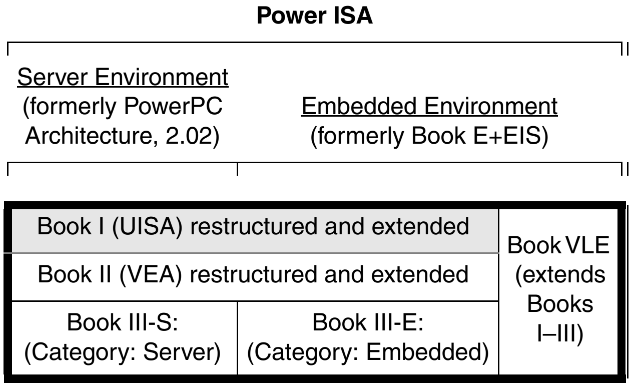

Power ISA
Server Environment
(formerly PowerPC Architecture, 2.02)
Embedded Environment
(formerly Book E+EIS)
Book I (UISA) restructured and extended
Book II (VEA) restructured and extended
Book III-S: (Category: Server)
Book III-E: (Category: Embedded)
Book VLE (extends Books I–III)

Figure 1-1. Power ISA

1.1.2 Introducing the Power ISA Category

The Power ISA category describes distinct modules within the architecture.

1.1.2.1 Understanding the Base Category

The Base category is common to both Embedded and Server categories, and it was originally derived from the Book I and Book II of the original PowerPC architecture. As Power ISA has evolved, the Base category has had features added to make the Base category common to all implementations of Power ISA.

1.1.2.2 Understanding the Embedded and Server Categories

In addition to the Base category, the Embedded and Server categories define functionality restricted to those respective environments. For example, these two categories include separate Book IIIs. In the Power ISA, categories also replace the concept of Auxiliary Processing Units (APUs), which were previously described by Book E and older versions of EREF, as modules of functionality that were extensions to Book E. In some places, this document continues to use the concept of Auxiliary Processing Units (APUs). In general, an APU reasonably maps to a category of Power ISA; however, some categories in Power ISA do not map to APUs. The concept ‘category’ is useful primarily as an arbitrary convention for structuring of the architecture specification itself; it is not particularly useful in understanding the behavior of the functionality.

1.1.2.3 Benefits of Categories

Categories extend the Base and Embedded category programming model. The modularity of categories facilitates the development of resources that are useful to some embedded environments but not others. Because of the mutually-exclusive nature of such extensions, instructions from multiple categories may be assigned the same opcode numbers. Such categories are mutually exclusive for a given implementation unless the implementation provides an implementation-specific method for using one category or the other.

Categories may consist of any combination of instructions, optional behavior of Power ISA-defined instructions, registers, register files, fields within Power ISA-defined registers, interrupts, or exception conditions within Power ISA-defined interrupts.

1.1.3 Power Architecture Evolution

The PowerPC architecture is the grandfather to the latest generation of the Power Architecture technology.

This figure shows the relationship between the different environments.

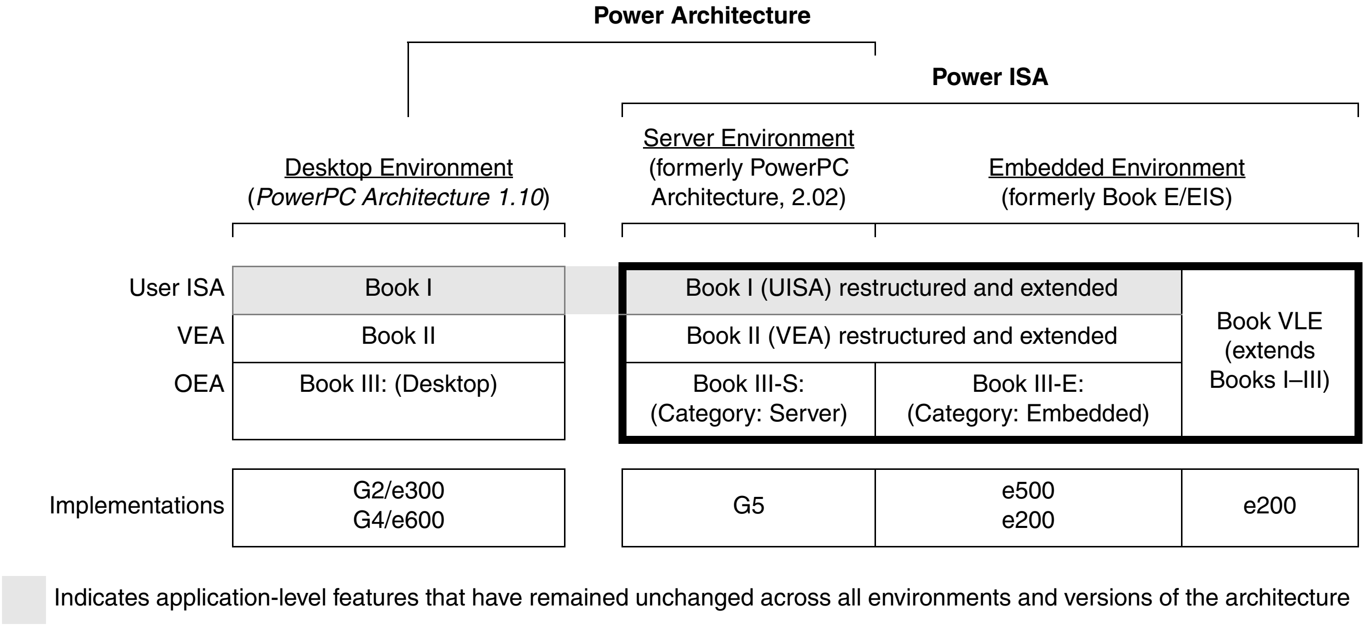


Figure 1-2. Power Architecture Relationships

The definitions that consist of the Power Architecture technology are as follows:

- The PowerPC Instruction Set Architecture (ISA) 1.10. The original architecture defined in the 1990s by Apple, IBM, and Motorola's Semiconductor Products Sector (SPS) (now Freescale). This mature architecture continues to form the basis for developing PowerPC processors that use Freescale's G2, e300, and e600 processor cores. This version of the architecture is described in *Programming Environments Manual for 32-bit Implementations of the PowerPC™ Architecture*.
- The Power ISA specification. The Power ISA specification brings together the embedded features defined in Book E and the Freescale EIS with the server and desktop resources defined by IBM's PowerPC architecture 2.02 definition. This architecture continues to evolve.

1.1.4 Freescale Embedded Implementation Standards (EIS)

The EIS refers to the architecture common to all Freescale embedded devices. It includes the following:

- Power ISA architecture features defined for the embedded environment
- Additional Power ISA features (categories such as Floating-Point, Embedded.Hypervisor, Decorated Storage, etc.)
- Freescale-specific features that, if implemented, are implemented consistently among Freescale embedded devices

Under Book E, programming model extensions took the form of APUs. A few APUs were strictly specific to individual devices, but most, such as the signal processing engine (SPE) and variable-length encoding (VLE), were designed to be used by multiple implementations and so were defined as part of the EIS. Most of the APUs formerly defined by the EIS are now part of the Power ISA (where they are referred to as categories).

As Figure 1-2 shows, processors designed under the Book E/EIS and PowerPC advanced server 2.02 architectures remain compliant with the restructured Power ISA architecture.

1.1.5 EIS Use of Power ISA Categories

Many features formerly defined by the EIS have become part of the Power ISA. These include Freescale's embedded MMU model and many former APUs, such as SPE and VLE. What is new is how these different architectures have been joined under a new name that reflects the expansive reach of the diversified architecture. Some of these features, such as the Integer Select instruction (**isel**), have become part of the basic programming model (category Base); others that are available to, but not required by, embedded processors, are defined by the Power ISA as separate categories.

This table provides references for categories described briefly in the EREF.

Table 1-1. References for Related Categories

Category	Definition	Reference
Variable length encoding (VLE)	Variable-length encoding facility, which re-encodes some of the basic instruction set to fit into 16-bit and 32-bit instructions.	The VLE category is implemented in many e200 cores. The VLE category in Power ISA is specified in Book VLE.
Signal processing engine (SPE)	A comprehensive set of 64-bit, two-element, SIMD instructions that share the Book I–defined GPRs extended by the SPE to 64 bits in 32-bit implementations, as shown in Figure 1-3.	The SPE category extends Book III–defined features, in particular, the interrupt model. The EREF provides an overview of the SPE, but a full description is provided in the SPE PEM.
Altivec[1]	This comprehensive 128-bit, four-operand SIMD ISA consists of instructions, a set of 32, 128-bit vector registers (**v**Rs), the vector save register (VRSAVE), and the vector status and control register (VSCR), which is analogous to the FPSCR.	The vector category extends the Book III interrupt model. For more information, see the AltiVec PEM.

Note:

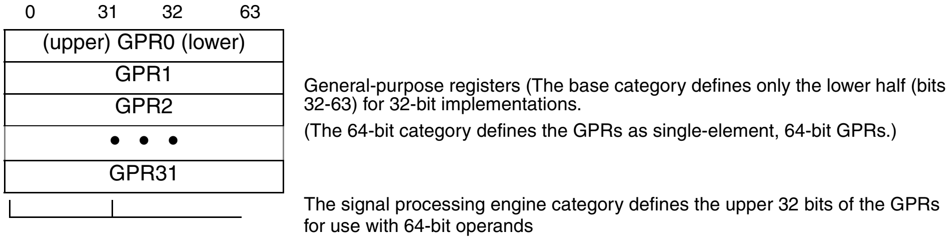
[1] AltiVec technology was introduced in 1998 as an extension to (but not formally a part of) the PowerPC architecture.

This figure provides an illustration of the GPRs extended by the SPE to 64 bits in 32-bit implementations.

NOTE

The SPE defines three dependent Embedded Float categories:

- Embedded Float Scalar Double:
  GPR-based floating-point double-precision
- Embedded Float Scalar Single:
  GPR-based floating-point single-precision
- Embedded Float Vector:
  GPR-based floating-point vector single-precision



General-purpose registers (The base category defines only the lower half (bits 32-63) for 32-bit implementations.

(The 64-bit category defines the GPRs as single-element, 64-bit GPRs.)

The signal processing engine category defines the upper 32 bits of the GPRs for use with 64-bit operands

Figure 1-3. Extended GPRs as Implemented to Support SPE Instructions

Freescale processors compliant with EREF implement several categories; although, in some instances, implementations may not support all instructions, instruction variants, registers, or other parts of the architectural definition of the category in Power ISA. This is because:

- Power ISA continues to evolve. New versions of the architecture may contain additions to existing categories; Freescale cores were implemented to conform to earlier versions of Power ISA.
- The EIS may designate that some features of Power ISA categories are not implemented by Freescale cores. Features of Power ISA categories that are not defined in the EREF, but appear in Power ISA, are examples of this.

1.1.5.1 Category Designations in EIS

In this document, category designations appear between angled brackets. Text associated with the designation applies only if the processor implements the category, or the category abbreviation, inside the angled brackets. Category designations appear in other places in the document as well, and are documented accordingly; for example, instruction definitions contain a category definition. Any definitions or descriptions that appear in this document and have a category designation are considered part of the Base category, or as overall descriptions of the architecture itself.

1.1.5.1.1 Category Designation Considerations

The following example of a category indicates that the sentence applies only to category 64-bit and to processors that implement that category:

- The upper 32 bits of the register are set to 0. <64-bit>

When a category designation appears at the end of a section title (for example, “2.8.3 External Proxy Register <EXP>”), this means the category designation applies to the entire section (or chapter, if the category designation appears at the end of a chapter title).

When a category designation appears on a separate line, preceding a paragraph, this means that the category designation applies to the paragraph that follows. The following example illustrates this point:

- <Embedded.Hypervisor>:
  If the processor is in the guest supervisor state...

When a category designation appears inside a sentence or phrase (for example, “SPE <SP>/embedded floating-point <SP.FD,SP.FV>/vector unavailable interrupt <V>,” this means the category designation applies to the previous word or clause.

1.1.5.2 Categories Supported by All Freescale EIS Processors

All EREF-compliant Freescale processors implement at least the following categories from Power ISA 2.06:

- Base
- Embedded
- Cache Specification

In addition to these categories, other features that are not associated with a category but appear in this document are also assumed to be implemented by processors compliant with EREF.

1.1.5.3 Categories Supported by Some Freescale EIS Processors

Many Freescale processors implement additional categories to provide additional functionality or acceleration capabilities that are useful for a given solution space, but which may not be applicable to all solution spaces. While these categories may be considered “optional,” many processors share a common subset of these categories. The mix of supported categories also occurs because the architecture evolves, and processor cores developed prior to a new version of the architecture may not contain a category developed as part of the new architecture version. In general, Power ISA introduces new functionality as categories, and, in later versions, moves them to Base, if so warranted.

Pieces of the architecture that are not part of a non-Base category may be implemented optionally on a given processor. Text that describes the feature indicates whether or not it is optional.

NOTE

> Power ISA also defines the Embedded category, reflecting the embedded environment of Power ISA. The Embedded category is assumed by EIS and is considered to be the same as Base.

1.1.5.3.1 PowerISA Category Abbreviations and Acronyms

This table shows PowerISA categories, and their associated abbreviations, which are considered to be a part of EREF and appear on some Freescale processors.

Table 1-2. PowerISA Category Abbreviations and Acronyms

Acronym	PowerISA Category
ATB	Alternate Time Base
DS	Decorated Storage
E.DC	Embedded.Device Control
E.ED	Embedded.Enhanced Debug
E.PD	Embedded.External PID
E.HV	Embedded.Hypervisor
E.LE	Embedded.Little-Endian
E.PM	Embedded.Performance Monitor
E.PC	Embedded.Processor Control
E.CL	Embedded.Cache Locking
EXP	External Proxy
FP	Floating-Point
FP.R	Floating-Point.Record
SP	SPE
SP.FD	SPE.Embedded Float Scalar Double
SP.FS	SPE.Embedded Float Scalar Single
SP.FV	SPE.Embedded Float Vector
VLE	Variable Length Encoding
V	Vector
WT	Wait
64	64-bit

1.1.5.3.2 EIS Category Abbreviations and Acronyms

EIS includes categories that are not defined by PowerISA. The previous version of EIS called these Auxiliary Processing Units (APUs).

This table shows categories defined only by EIS.

Table 1-3. EIS Category Abbreviations and Acronyms

Acronym	EIS Category
DCF	Direct Cache Flush
CWP	Cache Way Partitioning

Table 1-3. EIS Category Abbreviations and Acronyms (continued)

Acronym	EIS Category
DEO	Data Cache Block Extended Operations
ER	Enhanced Reservations
WS	Write Shadow Mode
CS	Cache Stashing

1.1.5.4 Categories Not Supported by Freescale EIS Processors

Some categories present in Power ISA are not part of EREF and are not implemented on Freescale devices. The following categories defined by Power ISA 2.06 are not part of EREF:

- Server (and any dependent categories)
- Decimal Floating-Point
- Embedded.Cache.Debug
- Embedded.Cache.Initialization
- Embedded.Hypervisor.LRAT
- Embedded.Multi-Threading
- Embedded.Page Table
- Embedded.TLB Write Conditional
- Legacy Integer Multiply-Accumulate
- Legacy Move Assist
- Load/Store Quadword
- Move Assist
- Processor Compatibility
- Stream
- Strong Access Order
- Trace
- Vector-Scalar Extension
- Vector.Little Endian

1.1.6 Special Instruction Differences Between Book E and EREF

Some instructions present in Book E have had their definition altered slightly from the Book E specification in order to effectively merge these instructions in to Power ISA. In particular, the following changes occurred:

- The Book E–defined **msync** instruction has reverted to being the Synchronize instruction **sync** defined by the PowerPC architecture. In the embedded category, **msync** is a simplified mnemonic for the **sync** instruction to ensure compatibility with the Book E **msync**. The instruction encoding remains the same, and the semantics of **sync** (without operands) remains the same. The **mbar** instruction, defined in Book E, remains as part of the embedded category; the equivalent PowerPC architecture 1.10 instruction **eieio**, which shares the opcode, is defined as part of the Server category.

- The CT field in the **dcbt** and **dcbtst** instructions is now renamed as the TH field. The semantic of the field remains the same for values less than 8, which is a cache identifier. Category Stream, which is not supported in EREF, uses values of the TH field which are 8 or greater.

1.2 EREF Programming Model

1.2.1 RISC Architecture Overview

Power Architecture processors are fundamentally a classic load/store RISC architecture. In general, such RISC architectures have fixed-width instruction length, and contain a substantial number of selectable registers to provide inputs and outputs of computations. The instructions that are defined are generally less complex, with the expectation that those instructions can be implemented simply, and that the implementation will more easily be able to execute instructions out of order, resulting in an implementation that is smaller, lower in power usage, and less complex.

The programming model of EREF follows accordingly. Memory accesses load or store data to/from memory via registers. Computation instructions use the data in registers as inputs to the computation and produce results that are placed in another register. In general, a single instruction does not perform computation and memory operations with the exception that in some forms of loads and stores (those with “Update” forms), the effective address computed in the instruction is also placed into another register.

1.2.2 EIS Architecture Overview

EIS is defined as a 64-bit architecture which allows for 32-bit and 64-bit implementations. 64-bit implementations can run software in either 64-bit mode or 32-bit mode and can easily handle running a mix of both 32-bit and 64-bit contexts when controlled by an appropriately designed operating system or hypervisor. It is not possible to run a 64-bit context from an operating system or hypervisor which is running in 32-bit mode. Registers in EIS are defined using 64-bit numbering, even though 32-bit implementations use bits 32–63 of such registers. Instructions use 32-bit numbering since instructions are at most 32 bits long (all instructions are exactly 32 bits except for some VLE mode instructions which are 16 bits).

A more complete discussion of 32-bit and 64-bit modes is presented in Chapter 2, “Computation Modes.”

1.2.3 Privilege Model

EIS provides three privilege levels to provide different levels of protection for software to operate under. This allows software to efficiently build systems that provide partitioning, virtualization environments, and protection.

This table provides privilege level definitions.

Table 1-4. Privilege Level Definitions

State	Definition	Use
Hypervisor state	This is the highest privilege level; it has access to all instructions and resources. Also called “hypervisor privileged,” “hypervisor mode,” or “hypervisor.”	The hypervisor state is used to provide partitioning and virtualization for operating systems software, which runs as a guest to the hypervisor software.
Guest supervisor state	This is the supervisor state at which operating systems which are hosted by a hypervisor run. Also called the “guest supervisor,” or “supervisor mode.”	The guest supervisor state is allowed to perform several privileged operations, and access any resources that are not hypervisor privileged.
User state	This is the unprivileged state in which a program runs under the control of an operating system. Also called user mode, user, or in Power ISA, called problem state.	The vast majority of instructions are allowed in the user state.

1.2.3.1 Hypervisor State Considerations

A software program running in the hypervisor state that is hosting guests in a partitioned environment is called a hypervisor. The hypervisor program partitions the system among guest operating systems and manages any shared state between the guests in such a manner that prevents the guests from interfering with each other. The way in which the hypervisor implements policy decisions with regard to managing these shared resources determines the amount of interferance that can occur between guests.

When the Embedded.Hypervisor category is implemented, the hypervisor state can run a guest operating system in guest supervisor state (see Section 1.2.3.2, “Guest Supervisor State Considerations”). If the Embedded.Hypervisor category is not implemented on a core, the operating system software runs in the equivalent of hypervisor state, and there is no guest supervisor state. Even when the Embedded.Hypervisor category is implemented, it is still practical to run an operating system in hypervisor state. This is termed running an operating system on “bare metal.” The programming model of running an operating system bare metal on a core that implements the Embedded.Hypervisor category is the same as if the category is not implemented.

1.2.3.2 Guest Supervisor State Considerations

Attempting to access hypervisor privileged resources and instructions will cause embedded hypervisor privilege interrupts which the hypervisor can service on behalf of the guest to provide appropriate emulation. What programming model the guest supervisor state should use depends on how the hypervisor is implemented. In general it is practical for the hypervisor to implement the same model (or a model very similar to) the programming model of running the guest operating system on bare metal.

1.2.3.3 User State Considerations

NOTE

> Attempting to access hypervisor or guest supervisor resources or instructions causes privilege exceptions, which the operating system can handle appropriately.

There is no difference in user state privilege when the user state program is running under control of an operating system hosted as a guest to a hypervisor, running under control of a hypervisor, or running under control of an operating system which is running bare metal. The cases differ only in the manner in which some interrupts are directed when the user state program attempts to access a privileged resource. In each case, the interrupts are directed to the state that controls the user state program or to the hypervisor state.

1.2.4 Instruction Model

Except for category VLE, which describes a variable length set of instructions, the instruction set contains the following features:

- Instructions are 32 bits in length and contain a primary opcode in the first six bits. Instructions are described using 32-bit numbering. For most primary opcode values, bits 21–30 contain the secondary opcode. Instructions which encode large immediate values generally occupy a single primary opcode, and use bits 16–31 to provide a signed or unsigned immediate field.
- Most computational instructions are triadic (that is, they contain three register operands). One of the register operands specifies a destination or target (where the output of the computation is placed). The other register operands specify inputs to the computation.

Chapter 4, “Instruction Model” details how instructions operate and briefly presents the instruction set organized by function. The major sections are:

- Section 4.1, “Operand Conventions”
- Section 4.2, “Instruction Set Characteristics”
- Section 4.3, “Classes of Instructions”
- Section 4.4, “Forms of Defined Instructions”
- Section 4.5, “Instruction-Related Exceptions”
- Section 4.6, “Instruction Summary”
- Section 4.7, “Instruction Listing”

Full descriptions of each instruction, including opcode, syntax, affected registers, related interrupts, and RTL (register transfer language) are provided in Chapter 5, “Instruction Set”; such descriptions for the VLE, SPE, and AltiVec instructions are provided in their respective programming environments manuals (PEMs).

1.2.4.1 VLE Instruction Set Overview

The VLE instruction set is decoded when the processor is in VLE mode. VLE mode is provided by cores that implement category VLE. VLE instructions are variable length and provide a significant improvement in code density over the standard instruction set, with an increase in code path length. VLE mode is generally only implemented on cores that are targeted for markets where code density is important.

1.2.4.2 VLE Instruction Set Features

The VLE instruction set contains the following features:

- Instructions are 16-bit or 32-bit in length and instruction decoding is more dynamic in how fields are interpreted.
- Instructions from primary opcode 4 and 31 in the standard instruction set decode the same for the VLE instruction set, making all the instructions in those two primary opcodes available in VLE mode. Note that opcode 4 is used for category SPE instructions as well as category Vector (Altivec) instructions.
- Most instructions are dyadic (that is, they contain two register operands). One of the operands specifies a register as both the destination or target as well as one of the inputs.
- Not all of the registers that are available in the standard instruction set are available to be used for all VLE instructions (that is, some VLE instructions cannot address all of the 32 General Purpose Registers).
- Since VLE instructions can be placed on 16-bit boundaries, the core is capable of branching to a 16-bit boundary.

1.2.5 Register Model

This section summarizes architecture-defined registers implemented by Freescale embedded devices.

NOTE

> Not all registers are implemented by all processor cores. Whether a register is implemented depends on what architectural categories the core implements, and what version of the architecture that the core implements.

The sets of registers that are provided are briefly shown in this table.

Table 1-5. Register sets in EIS

Register Set	Category	Number	Bits per Register	Description
GPR	Base	32	32 or 64	General Purpose Registers. The primary register set. GPRs are used for a variety of functions including load/store, arithmetic, logical, and address calculation operations. Some categories (SPE.Embedded Floating Point *) also make use of floating-point data. The number of bits per register are 64-bits if the implementation is a 64-bit implementation or if category SPE is implemented. Otherwise, the number of bits per register is 32. See Section 3.3.1, “General-Purpose Registers (GPRs).”
CR	Base	1	32	Condition Register. The condition register is a single 32-bit register, however it is generally used as a set of eight condition register fields (CR*n*) of four bits each. Condition register bits are used by conditional branch instructions to determine if a branch is to be taken (or used for the **isel** instruction which input register is to be written to the target register). Condition register fields or bits can be set by compare instructions, condition register logical instructions, or as the result of a record form (instruction mnemonic ending with “**.**”). See Section 3.5.1, “Condition Register (CR).”
FPR <FP>	Floating Point	32	64	Floating Point Registers. FPRs are used in a variety of functions on floating point data including load/store, arithmetic, and logical operations.See Section 3.4.1, “Floating-Point Registers (FPR0–FPR31).”

Table 1-5. Register sets in EIS (continued)

Register Set	Category	Number	Bits per Register	Description
VR <V>	Vector (Altivec)	32	128	Vector Registers. VRs are used for a variety of functions for processing SIMD vector elements. VRs can contain floating point data, integer data, and more. See the AltiVec PEM.
SPR	—	1024	32 or 64	Special Purpose Registers. SPRs are used for controlling a variety of functions. A few of the SPRs are available to user mode (such as Link Register, Count Register, Integer Exception Register), however the large majority of them are used by supervisor and hypervisor mode to control various aspects of the processor. SPRs are accessed with **mfspr** and **mtspr** instructions. See Section 3.2.2, “Special-Purpose Registers (SPRs).”
PMR <E.PM>	Embedded. Performance Monitor	1024	32	Performance Monitor Registers. PMRs are used for controlling the performance monitor facility which includes control functions and event counts. Changing the control functions is restricted to supervisor and hypervisor mode however some of the registers can be read in user mode. PMRs are accessed with **mfpmr** and **mtpmr** instructions. See Section 3.15.4, “Performance Monitor Counter Registers (PMCn/UPMCn).”
DCR <E.DC>	Embedded. Device Control	1024	32	Device Control Registers. DCRs are registers that exist architecturally outside the processor and are thus not part of the processor architecture. DCRs are implementation specific and defined in the core reference manual that implements them. DCRs are accessed with **mfdcr** and **mtdcr** instructions. See Section 3.18, “Device Control Registers (DCRs).”
individual registers	—	—	32 or 64	Individual registers are not a register set, but represent a few registers which are similar in nature to SPRs, but have specific instructions defined to access them. Such register include Machine State Register (MSR), Floating-Point Status and Control Register (FPSCR), Accumulator (ACC), and Vector Status and Control Register (VSCR).

1.2.6 Exceptions and Interrupt Handling

The following sections define the interrupt model, including an overview of exception handling as implemented in Freescale embedded devices, a brief description of the exception classes, and an overview of the registers involved.

1.2.6.1 Understanding Exceptions

In general, interrupt processing occurs as a result of an exception condition existing in the processor. An exception condition occurs when certain external signals are presented, when error conditions are detected, or program execution problems are encountered. More than one exception condition may be present during execution of a given instruction, and an exception priority mechanism exists to determine the highest priority exception condition that can cause an interrupt.

1.2.6.2 Interrupts Overview

When an exception is detected, and the exception condition has an interrupt enable associated with it or the exception condition does not require an associated interrupt enable, and it is the highest priority exception condition, the processor takes an interrupt. The interrupt causes the critical state of the processor to be saved into appropriate save/restore registers and redirects execution of the processor to the interrupt vector associated with the interrupt and changes the critical machine state so that it is appropriate for executing the interrupt handler.

1.2.6.2.1 Understanding the Four Levels of Interrupts

There are four levels of interrupts; the levels determine which set of save/restore registers the critical state is saved to when the interrupt occurs. The levels are somewhat hierarchical since an interrupt at a “higher” level will automatically set critical machine state so that asynchronous interrupts of a “lower” level are disabled. Synchronous exceptions are not disabled by critical machine state and can occur if problems occur doing instruction execution. System software is responsible for ensuring that no such conditions occur until it has safely saved any necessary state before executing instructions that might generate synchronous exceptions.

This table provides the four levels of interrupts from low to high.

Table 1-6. Four Levels of Interrupts

Interrupt Level	Registers Used	Used to handle...
Standard, non-critical interrupts	Save and restore registers (SRR0/SRR1)[1]	Standard interrupts are used to handle most synchronous program exception conditions. Standard interrupts are also used to handle the asynchronous external input exception, programmable asynchronous timer exceptions, and asynchronous doorbell exceptions. Standard interrupts which are asynchronous are enabled or disabled based on the state of the EE bit in the Machine State Register (MSR).
Critical interrupts	Critical save and restore registers (CSRR0/CSRR1)	Critical interrupts are used to handle the asynchronous critical input exception, the watchdog timer exceptions, and critical asynchronous doorbell exceptions. Critical interrupts that are asynchronous are enabled or disabled based on the state of the CE bit in the Machine State Register (MSR). Critical interrupts are also used to handle synchronous and asynchronous debug program exception conditions if category Embedded.Enhanced Debug is not implemented.
Debug interrupts	Debug save and restore registers (DSRR0/DSRR1)	Debug interrupts are used to handle synchronous and asynchronous debug program exception conditions when category Embedded.Enhanced Debug is implemented. Debug interrupts are enabled or disabled based on the state of the DE bit in the Machine State Register (MSR). <E.ED>
Machine check interrupts	Machine check save and restore registers (MCSRR0/MCSRR1)	Machine check interrupts are used to handle synchronous program exception conditions (called Error Reports) that are detected during execution of an instruction. Machine check interrupts are also used to handle the asynchronous machine check exception and the asynchronous non-maskable interrupt exception. Machine check interrupts resulting from asynchronous machine check exceptions are enabled or disabled based on the state of the ME bit in the Machine State Register (MSR).

Note:

[1] For guest operating systems operating under control of a hypervisor, separate guest save and restore registers are provided (GSRR0/GSRR1).

1.2.6.2.2 Interrupt Handling

Once the interrupt handler is executing, the handler may need to check one or more bits in the exception syndrome register (ESR) or in other status and control registers to diagnose the cause of the exception and take appropriate action.

Integrated devices also define error detection, enabling, and reporting bits that the handler would need to read to determine specific sources of asynchronous interrupts, which are typically reported as external input, critical input, or machine check interrupts. Often these fields are implemented in memory-mapped registers and are described in the documentation for the integrated device.

When an interrupt handler completes its execution, it normally returns to the context that it interrupted by way of a return from interrupt instruction (**rfi**, **rfgi** <E.HV>, **rfci**, **rfdi** <E.ED>, or **rfmci**), which returns execution to the interrupted address and returns the critical machine state to its previous pre-interrupt state.

The interrupts are described in Section Chapter 7, “Interrupts and Exceptions.”

1.2.7 Memory (Storage) Model

1.2.7.1 Memory Addresses

An address is required to access locations in storage (memory).

This table provides the four different types of memory addresses defined by the architecture.

Table 1-7. Four Types of Memory Addresses

Address Type	Definition	Used...
Effective address (EA)	EAs are generated by load, store, cache management, and other instructions. An EA is also generated by branch instructions and the sequential execution of instruction (that is, the next instruction to be executed or the address of an instruction). The vast majority of software deals only with EAs. EAs are transformed by the processor into other types of addresses through address translation in the memory management unit. The memory management unit is normally maintained by the operating system and/or the hypervisor. EAs are either 32 bits in 32-bit mode, or 64 bits in 64-bit mode.	By software to specify locations in the memory map.
Virtual Address (VA)	A VA is formed by taking an EA and adding address space information from the Machine State Register (MSR[IS,DS], MSR[GS] <E.HV>), the Processes ID Registers (PIDn), and the Logical Partition ID Register (LPIDR) <E.HV>. The address space information is maintained by the operating system and the hypervisor. In general, a unique address space is provided for each process or task performed by the operating system. The hypervisor also provides unique address spaces between logical partitions.	To find matching translations in the memory management unit.

Table 1-7. Four Types of Memory Addresses (continued)

Address Type	Definition	Used...
Logical Address (LA)	LAs apply only to processors that implement logical partitioning (category E.HV). The hypervisor or a Logical to Real Address Translation <E.HV.LRAT> translates the LA into the appropriate real address. A guest operating system is unaware of what its true real addresses are. LAs are always translated to RAs at the time when a TLB entry is written.	By the guest operating system as a real address. These addresses are normally specified when the guest operating system attempts to write translation lookaside buffer (TLB) entries.
Real Address (RA)	An RA is the actual physical address that is used when the processor performs transactions. The RA is normally formed when a VA is translated by the memory management unit. The system interconnect may also provide its own translation of RA to RA; however, these translations are outside the scope of the processor architecture.	To access a location in the caches, or to access a storage location managed outside the processor (for example DRAM, a peripheral device, and so on) by sending it out on the system interconnect.

1.2.7.2 Weakly Ordered Memory

To optimize performance, the architecture supports weakly ordered references to memory. On a single processor, references to memory always appear to occur in program order. However, when memory is shared between multiple processors or between a processor and a device, memory accesses may not occur in program order. Thus, software must manage the order and synchronization of memory accesses to ensure proper execution when memory is shared between multiple processors or devices. The cache and data memory control attributes, along with the Synchronize (**sync**) and Memory Barrier (**mbar)** instructions, provide the required controls.

1.2.7.3 Memory Management

The architecture supports demand-paged virtual memory as well other memory management schemes that depend on precise control of effective-to-physical address translation and flexible memory protection. The mapping mechanism consists of software-managed unified (translate both instruction and data references) TLBs that support variable-sized pages with per-page properties and permissions.

1.2.7.3.1 Optional TLB Properties

Some of the properties that can be configured for each TLB are as follows:

- User-mode page execute, read, and write access
- Supervisor-mode page execute, read, write access
- Hypervisor or guest address space <E.HV>
- Logical partition ID <E.HV>
- Process ID
- Virtualization Fault <E.HV>

- Write-through required (W)
- Caching inhibited (I)
- Memory coherency required (M)
- Guarded (G)
- Endianness (E) (big endian or little endian)
- VLE mode <VLE>
- User-definable (U0–U3), a 4-bit implementation-specific field

1.2.7.3.2 How the Architecture Benefits Memory Management

The architecture provides the ability to have distinct address spaces between user processes, user and supervisor, logical partitions, and hypervisor state. The wealth of address spaces and the complete separation of address spaces provide flexibility for system software as well as an extra level of protection between differing software domains. To facilitate efficient accesses between address spaces, External Process ID (PID) instructions are provided for hypervisors and operating systems to perform storage and memory operations using another address space to perform translation and protection.

The architecture provides no ability for software to run in a “real mode,” in which effective address are used as real addresses. The requirement that all addresses are translated simplifies implementation and provides a strong foundation for partitioning and protection since the state of the MMU guarantees what real address the processor may access.

In addition, processors that implement the Embedded.Hypervisor category are provided with methods of hiding and translating real addresses specified by the guest (called logical addresses) into true real addresses.

1.2.7.4 Caches

The architecture supports a variety of cache models including split instruction and data (Harvard) with data caches that are kept coherent by hardware and instruction caches that are kept coherent through software coherency. Different levels of caches can be implemented, and the instruction set includes the ability to reference those cache directly for operations such as touch loads and cache locking.

Caches have a set of configuration SPRs for system software to understand the geometry and other aspects of cache operations. Also control and status SPRs are provided for software to configure the caches and perform maintenance and control operations.

Instructions are provided for performing cache block (line) operations such as locking, flushing, invalidating, and zeroing a specific block.

1.3 Timers

Several timers and timer functions are provided by the architecture. Most of the timers are driven by the time base, a 64-bit counter driven by an implementation clock or frequency. The time base can be accessed by software through SPRs.

Timers that are driven by the time base clock are as follows:

- Decrementer (DEC register). The decrementer is a 32-bit counter that counts down. The decrementer can be programmed to cause an interrupt to occur when the decrementer counts down to 0. Typically, software programs the DEC register with the number of time base ticks to elapse before the timer expires and an interrupt occurs. The decrementer also has an auto reload feature which will reload the DEC register from the Decrementer Auto Reload (DECAR) register when the counter expires.
- Fixed interval timer
- Watchdog timer

The architecture also provides an alternate time base, another 64-bit counter which is also driven by an implementation dependent frequency. In general, the time base and the alternate time base will increment on a different frequency. It is customary, but not required by the architecture for the alternate time base to be based on the processor clock frequency and the time base based on a frequency supplied external to the processor core.

1.4 Debug

The architecture provides a set of debug capabilities that can be utilized by an operating system, hypervisor, or debug agent running with supervisor or hypervisor privileges. The debug facility allows control of program execution for operations such as single-step, instruction address breakpoint, data access breakpoint, branch taken, and more. The architecture also supports basic integration with an external debugger (a debugger that operates independent of the processor). EIS does not define external debugger controls; EIS acknowledges the existence of external debugger controls, and provides a method for software to know when the processor is controlled by an external debugger.

1.5 Performance Monitoring

Performance monitoring supports counting of events such as processor clocks, instruction cache misses, data cache misses, mispredicted branches, and others. The count of these events may be configured to trigger a performance monitor interrupt.

The register set associated with performance monitoring consists of counter registers, a global control register, and local control registers implemented as performance monitor registers (PMRs) which are similar to SPRs. PMRs are read/write from supervisor mode, and each register is reflected to a corresponding read-only register for user mode. The **mtpmr** and **mfpmr** instructions move data to and from these registers.

For more information on performance monitoring, see Chapter 10, “Performance Monitor <E.PM>.”

1.6 Major Additions to this Revision

This version of EREF contains several additions to support the efficient use of multicore devices. It includes the following major additions:

- Support for secure partitioning and virtualization through a hypervisor privilege level. Features of the hypervisor privilege level are considered category Embedded.Hypervisor.

- Full definition of EREF as a 64-bit architecture.
- Another level of interrupts for debug interrupts, with the associated save/restore register pair and **rfdi** (return from debug interrupt) instructions.
- Topology independent processor-to-processor doorbell messages (interrupts) allowing one processor to send interrupts to other processors in the same coherence domain. This is considered category Embedded.Processor Control.
- The ability for privileged software to perform load and store operations using a specified address space for the translation and permissions. This allows privileged software to use virtual addresses from their processes (or guests). This is considered category Embedded.External PID.
- The ability to perform load and store operations with addresses associated with a device and pass decorations (meta data) about the transaction to the device which can then perform device unique semantics based on the decoration. This is considered category Decorated Storage. Some integrated devices use this to efficiently perform atomic updates to memory.
- Support for integrated backside private or shared L2 caches.

Chapter 2
Computation Modes

This chapter describes the computation modes of the processor. EIS offers both 32-bit and 64-bit computation modes on 64-bit implementations, and a 32-bit computation mode on 32-bit implementations.

2.1 Computation Modes

Implementations of EIS may be either 64-bit implementations or 32-bit implementations. 64-bit implementations provide for 64-bit effective addresses and provide 64-bit registers, and instructions for manipulating 64-bit addresses and 64-bit integer data. 32-bit implementations provide for 32-bit effective addresses and provide 32-bit registers, and instructions for manipulating 32-bit addresses and 32-bit integer data.

The computation mode of the processor is controlled through the setting of the Computation Mode bit in the Machine State Register (MSR[CM]). MSR[CM] may be changed by any move to MSR (**mtmsr**) instruction, return from interrupt (**rfi**, **rfgi** <E.HV>, **rfci**, **rfdi** <E.ED>, **rfmci**), or as the result of the processor taking an interrupt.

2.2 64-bit Implementations <64>

64-bit implementations provide two execution modes, 64-bit mode and 32-bit mode. In both of these modes, instructions that set a 64-bit register affect all 64 bits. The computational mode controls how the effective address is interpreted, how Condition Register bits and XER bits are set, how the Link Register is set by branch instructions in which LK=1, and how the Count Register is tested by branch conditional instructions. In both modes, effective address computations use all 64 bits of the relevant registers (General Purpose Registers (GPRs), Link Register, Count Register, etc.) and produce a 64-bit result. However, in 32-bit mode the high-order 32 bits of the computed effective address are ignored for the purpose of addressing memory.

In general, all instructions are available in both modes, however, software written using instructions from category 64-bit, but executing in 32-bit mode will not work correctly on a 32-bit implementation. If binary compatibility with 32-bit implementations is desired, software should restrict itself from using any instructions that are part of category 64 bit. For example, a program which runs in 32-bit mode on a 64-bit implementation, which uses the load doubleword (**ld**) instruction, will not run correctly on a 32-bit implementation.

Registers that are defined to be less than 64 bits such as many Special Purpose Registers (SPRs), when accessed on a 64-bit implementation, produce a result as if the remaining non-defined bits of the register are unimplemented. For example, a **mfspr** instruction which accesses an SPR defined as 32-bit, produces 0 in bits 0–31 of the target GPR, and places the value of the 32-bit defined SPR into bits 32–63 of the target

GPR. Similarly, a **mtspr** instruction which accesses an SPR defined as 32-bit writes only bits 32–63 of the source GPR into the SPR.

In 64-bit implementations, GPRs are 64-bits each. Note that processors that implement SPE also contain 64-bit GPRs, however, that is independent of whether the processor is a 64-bit implementation.

2.3 32-bit Implementations

32-bit implementations provide only 32-bit mode. Except for category Floating Point and Category SPE (and their dependent categories), instructions that set a register affect only the lower 32 bits, and instructions that source data use only the lower 32 bits, unless otherwise specified. Instructions from category 64 bit are not available and the effective address computation, XER bits, the Link Register, and the Count Register are affected by only bits 32–63.

Registers that are defined to be more than 32 bits such as some Special Purpose Registers (SPRs), when accessed on a 32-bit implementation, produce a result as if the remaining non-defined bits of the register are unimplemented. For example, a **mfspr** instruction which accesses an SPR defined as 64-bit, produces 0 in bits 0–31 of the target GPR, and places the value of bits 32–63 of the defined SPR into bits 32–63 of the target GPR. Similarly, a **mtspr** instruction which accesses an SPR defined as 64-bit writes bits 32–63 of the source GPR into bits 32–63 of the SPR.

In 32-bit implementations, GPRs are 32 bits each unless the processor implements category SPE. When SPE is implemented, only the lower 32 bits of the GPRs are used for non-SPE (and dependent categories) instructions. For example, on a 32-bit implementation that implements category SPE, a **lwz** instruction that references a GPR only uses the lower 32 bits of source registers to form the effective address and only modifies the lower 32 bits of the target register, leaving bits 0–31 of the target register unchanged.

Chapter 3
Register Model

This chapter describes the register model and indicates the architecture level at which each register is defined. Registers are defined using 64-bit numbering with the exception that AltiVec (Vector registers) are defined using 128 bit numbering. Thus, a 32-bit register will be defined using bits 32:63.

3.1 Register Model Overview

This section provides a general description of the types of registers provided by EIS. Freescale Power ISA embedded implementations include the following types of software-accessible registers.

NOTE

> Some fields of some registers are implementation-specific and some registers are only available if certain categories are implemented. Category assignment for each of the registers is specified with the definition of the register later in this chapter.

3.1.1 Registers Accessed as Part of an Instruction Execution

3.1.1.1 Registers Used for Integer Operations

The following registers are used for integer operations:

- General-purpose registers (GPRs)
  The architecture defines a set of 32 GPRs used to hold source and destination operands for load, store, arithmetic, and computational instructions, and to read and write to other registers.
- Integer exception register (XER)
  XER fields are set based on the operation of an instruction considered as a whole, not on intermediate results. (For example, the Subtract from Carrying instruction (**subfc**), the result of which is specified as the sum of three values, sets bits in the XER based on the entire operation, not on an intermediate sum.)
- Accumulator (ACC)
  The ACC is accessed based on the operation of SPE instructions which initialize the accumulator or access the accumulator (such as multiply-accumulate type instructions).

GPRs, the XER, and the ACC are described in Section 3.3, “Registers for Integer Operations.”

3.1.1.2 Registers Used for Floating-Point Operations

The following registers (category Floating-point) are used for floating-point operations:

- Floating-point registers (FPRs)—32 registers used to hold source and destination operands for floating-point operations. Note that the embedded floating-point instructions (from categories SPE.FD, SPE.FS, SPE.FV) use GPRs rather than FPRs for floating-point operands and are not associated with FPRs.
- Floating-point status and control register (FPSCR)—Used with floating-point operations.

FPRs and the FSCR are described in Section 3.4, "Registers for Floating-Point Operations <FP>."

3.1.1.3 Registers Used for Vector Operations

The following registers (category Vector) are used for vector operations:

- Vector registers (VRs)—32 registers used to hold source and destination operands for floating-point operations.
- Vector status and control register (VSCR)—Used with vector operations.

VRs and the VSCR are described in the AltiVec PEM.

3.1.1.4 Condition Register (CR)

The condition register (CR) is used to record the results of comparisons and conditions, such as overflows and carries, that occur as a result of executing arithmetic instructions (including those implemented by the SPE embedded floating-point instructions). The CR is described in Section 3.5, "Registers for Branch Operations."

3.1.1.5 Machine State Register (MSR)

The machine state register (MSR) is used by the operating system to configure parameters such as user/supervisor mode, address space, and enabling of asynchronous interrupts. See Section 3.6.1, "Machine State Register (MSR)."

3.1.2 Special-Purpose Registers (SPRs)

Special-purpose registers (SPRs) are accessed explicitly using **mtspr** and **mfspr** instructions. Some SPRs are also accessed implicitly through the use of other instructions, the taking of interrupts, or other processor behavior. SPRs are listed in Table 3-2 in Section 3.2.2, "Special-Purpose Registers (SPRs)."

SPRs are described by function in the following sections:

- Section 3.5, "Registers for Branch Operations"
- Section 3.6, "Processor Control Registers"
- Section 3.7, "Timer Registers"
- Section 3.8, "Interrupt Registers"
- Section 3.9, "Software-Use SPRs (SPRGs, GSPRGs<E.HV>, and USPRGs)"
- Section 3.10, "L1 Cache Registers"

- Section 3.11, “L2 Cache Registers”
- Section 3.12, “MMU Registers”
- Section 3.13, “Debug Registers”
- Section 3.14, “Processor Management Registers”

3.1.3 Performance Monitor Registers

Performance monitor registers (PMRs) are like SPRs, but are accessed with move to (**mtpmr**) and move from (**mfpmr**) PMR instructions. PMRs are described in Section 3.15, “Performance Monitor Registers (PMRs) <E.PM>.”

3.1.4 Device Control Registers

The architecture defines a format for implementation-specific Device Control Registers (DCRs), which are similar to SPRs but with implementation-specific functionality. See Section 3.18, “Device Control Registers (DCRs).”

3.2 Register Model

3.2.1 Understanding How the Registers are Accessed

This table describes how registers are accessed.

Table 3-1. How Registers are Accessed

Register	Description	Reference
General-purpose registers (GPRs)	Specified by integer operations to hold source and destination operands. Used by integer and other instructions that use GPRs to form the effective address of a memory access.	See Section 3.3.1, “General-Purpose Registers (GPRs).”
Floating-point registers (FPRs) <FP>	Specified by floating-point operations to hold source and destination for floating-point instructions. Note that SPE embedded floating-point instructions use GPRs.	See Section 3.4.1, “Floating-Point Registers (FPR0–FPR31).”
AltiVec Vector Registers (VRs) <V>	Specified by AltiVec vector operations to hold source and destination for AltiVec instructions.	See AltiVec PEM
Vector status and control register (VSCR) <V>	Contains status bits set as the result of vector operations. Accessed explicitly by VSCR-specific instructions, **mtvscr**, **mfvscr**, described in the AltiVec PEM.	See AltiVec PEM
Special-purpose registers (SPRs)	Accessed by using the Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions. Some SPRs reflect status at configuration or as the result of specific events or conditions.	See Section 3.2.2, “Special-Purpose Registers (SPRs).”
Machine state register (MSR)	Accessed with the Move to Machine State Register (**mtmsr**) and Move from Machine State Register (**mfmsr**) instructions.	See Section 3.6.1, “Machine State Register (MSR).”

Table 3-1. How Registers are Accessed (continued)

Register	Description	Reference
Condition register (CR)	CR bits are grouped into eight 4-bit fields, CR0–CR7, which are set as follows. • Specified CR fields can be set by a move to the CR from a GPR (**mtcrf**). • A specified CR field can be set by a move to the CR from another CR field (**mcrf**), from the FPSCR (**mcrfs**), or from the XER (**mcrxr**). CR0 can be set as the implicit result of an integer instruction. • CR1 can be set as the implicit result of a floating-point instruction. • A specified CR field can be set as the result of an integer or floating-point compare instruction (including SPE and SPFP compare instructions).	See Section 3.5.1, “Condition Register (CR)”
The floating-point status and control register (FPSCR) <FP>	Contains status bits set as the result of floating-point operations. Accessed explicitly by FPSCR-specific instructions, **mcrfs, mtfsfi, mtfsf,** or **mtfsb0**, described in Section 3.4.2, “Floating-Point Status and Control Register (FPSCR).”	See Section 3.4.2, “Floating-Point Status and Control Register (FPSCR).”
Accumulator (ACC) <SP>	Used by accumulating forms of SPE instructions.	See SPE PEM
Device control registers (DCRs) <E.DC>	The architecture defines the existence of a DCR address space and instructions to access them, but does not define particular DCRs. The on-chip DCRs exist architecturally outside the processor core. The contents of DCR DCRN can be read into a GPR using **mfdcr r**D**,**DCRN. GPR contents can be written into DCR DCRN using **mtdcr** DCRN**,r**S.	See Section 3.18, “Device Control Registers (DCRs).”
Performance monitor registers (PMRs) <E.PM>	PMRs are accessed by using the Move to Performance Monitor Register (**mtpmr**) and Move from Performance Monitor Register (**mfpmr**) instructions. Some PMRs reflect the count of certain events depending on how the performance monitor is configured.	See Section 3.15, “Performance Monitor Registers (PMRs) <E.PM>.”

3.2.2 Special-Purpose Registers (SPRs)

SPRs are on-chip registers implemented on the processor core. They are used to control the use of the debug facilities, timers, interrupts, memory management unit, and other architected processor resources and are accessed with the **mtspr** and **mfspr** instructions. The definition of each individual SPR is described later in this chapter organized by function. This section describes the SPR number assignments that are defined by the architecture. It is important to note the following:

- Although many, if not most, SPRs are required by all processors, not all processors implement all of the SPRs. Refer to the processor documentation to determine which SPRs are implemented.
- An implementation of an SPR may differ from the architectural definition in any of the following ways.
  - An implementation may not implement all architecture-defined SPR fields.
  - An implementation may not implement all bits within an architecture-defined SPR field.
  - An implementation may not support all possible architecturally defined bit settings for a particular SPR field.
  - An implementation may implement additional fields not defined by the architecture.

— The architecture defines some SPR fields in a very general way that may be defined more specifically by the core and by the integrated device that implements the core.

Because an implementation of an SPR may vary from the architectural definition in the ways described above, it is important to consult the reference manuals for the processor and for the integrated device to determine exactly how an SPR is implemented.

Table 3-2 summarizes SPRs. Access is given by the lowest level of privilege required to access the SPR. The following access methods appear in the table:

- User—denotes access is available for both **mfspr** and **mtspr** regardless of privilege level
- User RO—denotes access is available for only **mfspr** regardless of privilege level
- Guest supervisor—denotes access is available for both **mfspr** and **mtspr** when operating in supervisor mode (MSR[PR] = 0), regardless of the state of the MSR[GS] <E.HV> bit (for example, it is available in hypervisor state as well)
- Guest supervisor RO—denotes access is available for only **mfspr** when operating in supervisor mode (MSR[PR] = 0), regardless of the state of the MSR[GS] <E.HV> bit (for example, it is available in hypervisor state as well)
- Hypervisor—denotes access is available for both **mfspr** and **mtspr** when operating in hypervisor mode (MSR[GS,PR] = 0b00)
- Hypervisor RO—denotes access is available for only **mfspr** when operating in hypervisor mode (MSR[GS,PR] = 0b00)
- Hypervisor WO—denotes access is available for only **mtspr** when operating in hypervisor mode (MSR[GS,PR] = 0b00)
- Hypervisor R/Clear—denotes access is available for both **mfspr** and **mtspr** when operating in hypervisor mode (MSR[GS,PR] = 0b00); however, an **mtspr** only clears bit positions in the SPR that correspond to the bits set in the source GPR

For processors that do not implement category Embedded.Hypervisor, privilege level is determined solely by the MSR[PR] bit. Thus any register defined to be guest supervisor privileged or hypervisor privileged is considered to be supervisor privileged.

Bit 5 in an SPR number indicates whether an SPR is accessible from user or supervisor software. An **mtspr** or **mfspr** instruction that specifies an SPR number with bit 5 set while in user mode will cause a privilege operation program exception, regardless of whether the SPR number is implemented.

In guest supervisor mode, an **mtspr** or **mfspr** instruction that specifies an SPR that is hypervisor privileged will cause an embedded hypervisor privilege exception. For example, attempting to read an SPR which has “Hypervisor RO” privilege while in guest supervisor state will cause an embedded hypervisor privilege and subsequent interrupt.

An **mtspr** or **mfspr** instruction that specifies an unsupported SPR number is considered an invalid instruction. In user mode, the processor takes an illegal operation program exception on all accesses to unsupported, unprivileged SPRs (or read accesses to SPRs that are write-only and write accesses to SPRs that are read-only). In supervisor or hypervisor mode, such accesses are boundedly undefined.

Table 3-2. Special-Purpose Registers (by SPR Number)

Defined SPR Number	SPR Abbreviation	Name	Length (in bits)	Access (shared)	Section/ Page
1	XER	Integer exception register.	64	User	3.3.2/3-14
8	LR	Link register	64	User	3.5.2/3-33
9	CTR	Count register	64	User	3.5.3/3-34
22	DEC	Decrementer	32	Hypervisor	3.7.4/3-55
26	SRR0	Save/restore register 0	64	Guest supervisor[1]	3.8.1/3-57
27	SRR1	Save/restore register 1	32	Guest supervisor[1]	3.8.2/3-58
48	PID (PID0)	Process ID register[2]	14	Guest supervisor	3.12.2/3-100
54	DECAR	Decrementer auto-reload	32	Hypervisor WO	3.7.5/3-56
58	CSRR0	Critical save/restore register 0	64	Hypervisor	3.8.1/3-57
59	CSRR1	Critical save/restore register 1	32	Hypervisor	3.8.2/3-58
61	DEAR	Data exception address register	64	Guest supervisor[1]	3.8.9/3-65
62	ESR	Exception syndrome register	32	Guest supervisor[1]	3.8.7/3-62
63	IVPR	Interrupt vector prefix register	64	Hypervisor	3.8.3/3-59
256	USPRG0 (VRSAVE)	User SPR general 0[3]	32	User	3.9/3-69
259	SPRG3	SPR general 3 (alias to same physical register as SPR 275)	64	User RO[1]	3.9/3-69
260	SPRG4	SPR general 4 (alias to same physical register as SPR 276)	64	User RO	3.9/3-69
261	SPRG5	SPR general 5 (alias to same physical register as SPR 277)	64	User RO	3.9/3-69
262	SPRG6	SPR general 6 (alias to same physical register as SPR 278)	64	User RO	3.9/3-69
263	SPRG7	SPR general 7 (alias to same physical register as SPR 279)	64	User RO	3.9/3-69
268	TB (TBL)	Time base (time base lower) - read port to Time Base register (See Table 3-2 for how SPR ports access Time Base.)	64	User RO	3.7.3/3-54
269	TBU	Time base upper - read port to high-order 32-bits of Time Base register (See Table 3-2 for how SPR ports access Time Base.)	32	User RO	3.7.3/3-54

Table 3-2. Special-Purpose Registers (by SPR Number) (continued)

Defined SPR Number	SPR Abbreviation	Name	Length (in bits)	Access (shared)	Section/ Page
272	SPRG0	SPR general 0	64	Guest supervisor[1]	3.9/3-69
273	SPRG1	SPR general 1	64	Guest supervisor[1]	3.9/3-69
274	SPRG2	SPR general 2	64	Guest supervisor[1]	3.9/3-69
275	SPRG3	SPR general 3(alias to same physical register as SPR 275)	64	Guest supervisor[1]	3.9/3-69
276	SPRG4	SPR general 4 (alias to same physical register as SPR 276)	64	Guest supervisor	3.9/3-69
277	SPRG5	SPR general 5 (alias to same physical register as SPR 277)	64	Guest supervisor	3.9/3-69
278	SPRG6	SPR general 6 (alias to same physical register as SPR 278)	64	Guest supervisor	3.9/3-69
279	SPRG7	SPR general 7 (alias to same physical register as SPR 279)	64	Guest supervisor	3.9/3-69
284	TBL	Time base lower - write port to low-order 32 bits of Time Base register. (See Table 3-2 for how SPR ports access Time Base.)	32	Hypervisor WO	3.7.3/3-54
285	TBU	Time base upper - write port to high-order 32 bits of Time Base register. (See Table 3-2 for how SPR ports access Time Base.)	32	Hypervisor WO	3.7.3/3-54
286	PIR	Processor ID register	32	Guest supervisor[1]	3.6.5/3-46
287	PVR	Processor version register	32	Guest supervisor RO	3.6.7/3-49
304	DBSR	Debug status register[2]	32	Hypervisor R/Clear	3.13.4/3-129
306	DBSRWR	Debug status register write[2]	32	Hypervisor	3.13.4/3-129
307	EPCR	Embedded processor control register[2]	32	Hypervisor	3.6.3/3-41
308	DBCR0	Debug control register 0[2]	32	Hypervisor	3.13.2.1/3-118
309	DBCR1	Debug control register 1[2]	32	Hypervisor	3.13.2.2/3-120
310	DBCR2	Debug control register 2[2]	32	Hypervisor	3.13.2.3/3-123
311	MSRP	MSR protect[2]	32	Hypervisor	3.6.2/3-39
312	IAC1	Instruction address compare 1, table3-2	64	Hypervisor	3.13.5/3-133
313	IAC2	Instruction address compare 2[2]	64	Hypervisor	3.13.5/3-133
314	IAC3	Instruction address compare 3[2]	64	Hypervisor	3.13.5/3-133
315	IAC4	Instruction address compare 4[2]	64	Hypervisor	3.13.5/3-133
316	DAC1	Data address compare 1[2]	64	Hypervisor	3.13.6/3-134

Table 3-2. Special-Purpose Registers (by SPR Number) (continued)

Defined SPR Number	SPR Abbreviation	Name	Length (in bits)	Access (shared)	Section/ Page
317	DAC2	Data address compare 2[2]	64	Hypervisor	3.13.6/3-134
318	DVC1	Data value compare 1[2]	64	Hypervisor	3.13.7/3-134
319	DVC2	Data value compare 2[2]	64	Hypervisor	3.13.7/3-134
336	TSR	Timer status register	32	Hypervisor R/Clear	3.7.2/3-53
338	LPIDR	Logical PID register[2]	13	Hypervisor	3.12.1/3-100
339	MAS5	MMU assist register 5[2] (alias to same physical register as upper bits of SPR 348)	32	Hypervisor	3.12.6.6/3-111
340	TCR	Timer control register	32	Hypervisor	3.7.1/3-51
341	MAS8	MMU assist register 8[2]	32	Hypervisor	3.12.6.9/3-113
348	MAS5_MAS6	MMU assist register 5 and MMU assist register 6[2] (alias to same physical register as SPR 339 concatenated with SPR 630)	64	Hypervisor	3.12.6.10/3-114
349	MAS8_MAS1	MMU assist register 8 and MMU assist register 1[2] (alias to same physical register as SPR 341 concatenated with SPR 625)	64	Hypervisor	3.12.6.10/3-114
368	GSPRG0	Guest SPR general 0	64	Guest supervisor	3.9/3-69
369	GSPRG1	Guest SPR general 1	64	Guest supervisor	3.9/3-69
370	GSPRG2	Guest SPR general 2	64	Guest supervisor	3.9/3-69
371	GSPRG3	Guest SPR general 3	64	Guest supervisor	3.9/3-69
372	MAS7_MAS3	MMU assist register 7 and MMU assist register 3[2] (alias to same physical register as SPR 944 concatenated with SPR 627)	64	Guest supervisor	3.12.6.10/3-114
373	MAS0_MAS1	MMU assist register 0 and MMU assist register 1[2] (alias to same physical register as SPR 624 concatenated with SPR 625)	64	Guest supervisor	3.12.6.10/3-114
378	GSRR0	Guest save/restore register 0	64	Guest supervisor	3.8.1/3-57
379	GSRR1	Guest save/restore register 1	32	Guest supervisor	3.8.2/3-58
380	GEPR	Guest external proxy register	32	Guest supervisor	3.8.14/3-69
381	GDEAR	Guest data exception address register	64	Guest supervisor	3.8.10/3-66
382	GPIR	Guest processor ID register	32	Guest supervisor[4]	3.6.6/3-48
383	GESR	Guest exception syndrome register	32	Guest supervisor	3.8.8/3-65
400	IVOR0	Critical input interrupt offset	32	Hypervisor	3.8.5/3-60
401	IVOR1	Machine check interrupt offset	32	Hypervisor	3.8.5/3-60
402	IVOR2	Data storage interrupt offset	32	Hypervisor	3.8.5/3-60
403	IVOR3	Instruction storage interrupt offset	32	Hypervisor	3.8.5/3-60

Table 3-2. Special-Purpose Registers (by SPR Number) (continued)

Defined SPR Number	SPR Abbreviation	Name	Length (in bits)	Access (shared)	Section/ Page
404	IVOR4	External input interrupt offset	32	Hypervisor	3.8.5/3-60
405	IVOR5	Alignment interrupt offset	32	Hypervisor	3.8.5/3-60
406	IVOR6	Program interrupt offset	32	Hypervisor	3.8.5/3-60
407	IVOR7	Floating-point unavailable interrupt offset.	32	Hypervisor	3.8.5/3-60
408	IVOR8	System call interrupt offset	32	Hypervisor	3.8.5/3-60
409	IVOR9	APU unavailable interrupt offset	32	Hypervisor	3.8.5/3-60
410	IVOR10	Decrementer interrupt offset	32	Hypervisor	3.8.5/3-60
411	IVOR11	Fixed-interval timer interrupt offset	32	Hypervisor	3.8.5/3-60
412	IVOR12	Watchdog timer interrupt offset	32	Hypervisor	3.8.5/3-60
413	IVOR13	Data TLB error interrupt offset	32	Hypervisor	3.8.5/3-60
414	IVOR14	Instruction TLB error interrupt offset	32	Hypervisor	3.8.5/3-60
415	IVOR15	Debug interrupt offset	32	Hypervisor	3.8.5/3-60
432	IVOR38	Guest processor doorbell interrupt offset	32	Hypervisor	3.8.5/3-60
433	IVOR39	Guest processor doorbell critical and machine check interrupt offset	32	Hypervisor	3.8.5/3-60
434	IVOR40	Hypervisor system call interrupt offset	32	Hypervisor	3.8.5/3-60
435	IVOR41	Hypervisor privilege interrupt offset	32	Hypervisor	3.8.5/3-60
440	GIVOR2	Guest data storage interrupt offset	32	Hypervisor[4]	3.8.6/3-61
441	GIVOR3	Guest instruction storage interrupt offset	32	Hypervisor[4]	3.8.6/3-61
442	GIVOR4	Guest external input interrupt offset	32	Hypervisor[4]	3.8.6/3-61
443	GIVOR8	Guest system call interrupt offset	32	Hypervisor[4]	3.8.6/3-61
444	GIVOR13	Guest data TLB error interrupt offset	32	Hypervisor[4]	3.8.6/3-61
445	GIVOR14	Guest Instruction TLB error interrupt offset	32	Hypervisor[4]	3.8.6/3-61
447	GIVPR	Guest interrupt vector prefix	64	Hypervisor[4]	3.8.4/3-60
512	SPEFSCR	SPE floating point status and control register	32	User	SPE PEM
515	L1CFG0	L1 cache configuration register 0	32	User RO	3.10.6/3-80
516	L1CFG1	L1 cache configuration register 1	32	User RO	3.10.7/3-81
517	NPIDR[5]	Nexus processor ID register	32	User	3.13.8.4/3-137
519	L2CFG0	L2 cache configuration register 0	32	User RO	3.11.1/3-84
526	ATBL (ATBL)	Alternate time base register lower (alias to same physical register as SPR 527)	64	User RO	3.7.6/3-56
527	ATBU	Alternate time base register upper (alias to same physical register as upper bits of SPR 526)	32	User RO	3.7.6/3-56
528	IVOR32	SPE/Embedded floating point/AltiVec unavailable interrupt offset	32	Hypervisor	3.8.5/3-60

Table 3-2. Special-Purpose Registers (by SPR Number) (continued)

Defined SPR Number	SPR Abbreviation	Name	Length (in bits)	Access (shared)	Section/ Page
529	IVOR33	SPE/Embedded floating point/AltiVec unavailable interrupt offset	32	Hypervisor	3.8.5/3-60
530	IVOR34	Embedded floating point data exception/AltiVec assist interrupt offset	32	Hypervisor	3.8.5/3-60
531	IVOR35	Performance monitor interrupt offset	32	Hypervisor	3.8.5/3-60
532	IVOR36	Processor doorbell interrupt offset	32	Hypervisor	3.8.5/3-60
533	IVOR37	Processor doorbell critical interrupt offset	32	Hypervisor	3.8.5/3-60
561	DBCR3	Debug control register 3[2]	32	Hypervisor	3.13.2.4/3-125
569	MCARU	Machine check address register upper (alias to same physical register as upper bits of SPR 573 on some e500 implementations)	32	Hypervisor RO	3.8.12/3-68
570	MCSRR0	Machine-check save/restore register 0	64	Hypervisor	3.8.1/3-57
571	MCSRR1	Machine-check save/restore register 1	32	Hypervisor	3.8.2/3-58
572	MCSR	Machine check syndrome register	32	Hypervisor	3.8.11/3-67
573	MCAR	Machine check address register (upper 32-bits alias to same physical register as SPR 569)	64	Hypervisor RO	3.8.12/3-68
574	DSRR0	Debug save/restore register 0	64	Hypervisor	3.8.1/3-57
575	DSRR1	Debug save/restore register 1	32	Hypervisor	3.8.2/3-58
576	DDAM	Debug data acquisition message	32	User	3.13.8.3/3-137
601	DVC1U	Alias to high order 32 bits of DVC1 register. (alias to same physical registers upper bits of SPR 318)	32	Hypervisor	3.13.7/3-134
602	DVC2U	Alias to high order 32 bits of DVC2 register. (alias to same physical registers upper bits of SPR 319)	32	Hypervisor	3.13.7/3-134
604	SPRG8	SPRG8	64	Hypervisor	3.9/3-69
605	SPRG9	SPRG9	64	Guest supervisor	3.9/3-69
606	L1CSR2	L1 cache control and status register 2[2]	32	Hypervisor	3.10.3/3-77
607	L1CSR3	L1 cache control and status register 3[2]	32	Hypervisor	3.10.4/3-78
624	MAS0	MMU assist register 0[2] (alias to same physical register as upper bits of SPR 373)	32	Guest supervisor	3.12.6.1/3-105
625	MAS1	MMU assist register 1[2] (alias to same physical register as lower bits of SPR 373)	32	Guest supervisor	3.12.6.2/3-107
626	MAS2	MMU assist register 2[2]	64	Guest supervisor	3.12.6.3/3-108
627	MAS3	MMU assist register 3[2] (alias to same physical register as lower bits of SPR 372)	32	Guest supervisor	3.12.6.4/3-109
628	MAS4	MMU assist register 4[2]	32	Guest supervisor	3.12.6.5/3-110

Table 3-2. Special-Purpose Registers (by SPR Number) (continued)

Defined SPR Number	SPR Abbreviation	Name	Length (in bits)	Access (shared)	Section/ Page
630	MAS6	MMU assist register 6[2] (alias to same physical register as upper bits of SPR 348)	32	Guest supervisor	3.12.6.7/3-112
633	PID1	Process ID Register 1 (phased out)[2]	14	Guest supervisor	3.12.2/3-100
634	PID2	Process ID Register 2 (phased out)[2]	14	Guest supervisor	3.12.2/3-100
688	TLB0CFG	TLB configuration register 0	32	Hypervisor RO	3.12.5/3-103
689	TLB1CFG	TLB configuration register 1	32	Hypervisor RO	3.12.5/3-103
690	TLB2CFG	TLB configuration register 2	32	Hypervisor RO	3.12.5/3-103
691	TLB3CFG	TLB configuration register 3	32	Hypervisor RO	3.12.5/3-103
696	CDCSR0	Core device control and status register[2]	32	Hypervisor	3.14.1/3-137
702	EPR	External proxy register	32	Guest supervisor RO[1]	3.8.13/3-68
720	L2ERRINTEN	L2 cache error interrupt enable	32	Hypervisor	3.11.4.6/3-94
721	L2ERRATTR	L2 cache error attribute	32	Hypervisor	3.11.4.8/3-96
722	L2ERRADDR	L2 cache error address	32	Hypervisor	3.11.4.9/3-97
723	L2ERREADDR	L2 cache error extended address	32	Hypervisor	3.11.4.9/3-97
724	L2ERRCTL	L2 cache error control	32	Hypervisor	3.11.4.7/3-95
725	L2ERRDIS	L2 cache error disable	32	Hypervisor	3.11.4.4/3-92
944	MAS7	MMU assist register 7[2] (same physical register as upper bits of SPR 372)	32	Guest supervisor	3.12.6.8/3-113
947	EPLC	External PID load context[2]	32	Guest supervisor[6]	3.12.7.1/3-115
948	EPSC	External PID store context[2]	32	Guest supervisor[6]	3.12.7.2/3-116
959	L1FINV1	L1 cache flush and invalidate register 1	32	Hypervisor	3.10.9/3-83
975	DEVENT	Debug event	32	User WO	3.13.8.2/3-136
985	L2ERRINJHI	L2 cache error injection mask high	32	Hypervisor	3.11.4.13/3-99
986	L2ERRINJLO	L2 cache error injection mask low	32	Hypervisor	3.11.4.13/3-99
987	L2ERRINJCTL	L2 cache error injection control	32	Hypervisor	3.11.4.12/3-98
988	L2CAPTDATAHI	L2 cache error capture data high	32	Hypervisor	3.11.4.10/3-97
989	L2CAPTDATALO	L2 cache error capture data low	32	Hypervisor	3.11.4.10/3-97
990	L2CAPTECC	L2 cache error capture ECC syndrome	32	Hypervisor	3.11.4.11/3-98
991	L2ERRDET	L2 cache error detect	32	Hypervisor	3.11.4.5/3-93
1008	HID0	Hardware implementation dependent register 0[2]	32	Hypervisor	3.6.4/3-43
1009	HID1	Hardware implementation dependent register 1[2]	32	Hypervisor	3.6.4/3-43
1010	L1CSR0	L1 cache control and status register 0[2]	32	Hypervisor	3.10.1/3-71

Table 3-2. Special-Purpose Registers (by SPR Number) (continued)

Defined SPR Number	SPR Abbreviation	Name	Length (in bits)	Access (shared)	Section/ Page
1011	L1CSR1	L1 cache control and status register 1[2]	32	Hypervisor	3.10.2/3-75
1012	MMUCSR0	MMU control and status register 0[2]	32	Hypervisor	3.12.3/3-101
1013	BUCSR (BUCSR0)	Branch unit control and status register[2]	32	Hypervisor	3.5.4/3-35
1015	MMUCFG	MMU configuration register	32	Hypervisor RO	3.12.4/3-102
1016	L1FINV0	L1 cache flush and invalidate register 0	32	Hypervisor	3.10.8/3-82
1017	L2CSR0	L2 cache control and status register 0[2]	32	Hypervisor	3.11.2/3-85
1018	L2CSR1	L2 cache control and status register 1[2]	32	Hypervisor	3.11.3/3-89
1020-1022	—	Reserved.	—	—	—
1023	SVR	System version register	32	Guest supervisor RO	3.6.8/3-50

[1] When these registers are accessed in guest supervisor state, the access are mapped to their analogous guest SPRs (for example, DEAR is mapped to GDEAR, etc.). See Section 3.2.2.1, "SPR Register Mapping <E.HV>".

[2] Writing to these registers requires synchronization, as described in Section 4.5.4.3, "Synchronization Requirements."

[3] VRSAVE and USPRG0 are aliases to the same SPR number and register. This register is a separate physical register from SPRG0.

[4] This register is only writable in hypervisor state, but can be read in guest supervisor state.

[5] NPIDR contents are transferred to the Nexus port whenever it is written.

[6] Certain fields in this register are only writable in hypervisor state.

3.2.2.1 SPR Register Mapping <E.HV>

To facilitate better performance for operating systems executing in the guest supervisor state, some SPR accesses are redirected to analogous guest-state SPRs. An SPR is said to be *mapped* if this redirection takes place when executing in guest supervisor state. These guest-state SPRs separate performance critical state of the hypervisor and the operating system executing in guest supervisor state. The mapping of these register accesses allows the same programming model to be used for an operating system running in the guest supervisor state or is running bare metal.

For example, when a **mtspr** SRR0,r5 instruction is executed in guest supervisor state, the access to SRR0 is mapped to GSRR0. This produces the same operation as executing **mtspr** GSRR0,r5.

This table shows SPR accesses that are mapped in guest supervisor state.

Table 3-3. Register Mapping in Guest Supervisor State

Register	mfspr Mapping	mtspr Mapping	Notes
SRR0	GSRR0	GSRR0	Access mapped during **mtspr**, **mfspr**.[1]
SRR1	GSRR1	GSRR1	Access mapped during **mtspr**, **mfspr**.[1]

Table 3-3. Register Mapping in Guest Supervisor State (continued)

Register	mfspr Mapping	mtspr Mapping	Notes
EPR	GEPR	*none*	Guest supervisor **mtspr** to EPR causes embedded hypervisor privilege exception.
ESR	GESR	GESR	
DEAR	GDEAR	GDEAR	
PIR	GPIR	*none*	Guest supervisor **mtspr** to PIR causes embedded hypervisor privilege exception.[2]
SPRG0	GSPRG0	GSPRG0	
SPRG1	GSPRG1	GSPRG1	
SPRG2	GSPRG2	GSPRG2	
SPRG3	GSPRG3	GSPRG3	
SPRG3 (259)	GSPRG3	*none*	User level access to SPRG3 (259) for **mfspr** is mapped when executing in guest state regardless of the setting of MSR[PR].

[1] Note that indirect accesses to SRR0 and SRR1 through execution of the **rfi** instruction, are handled in a similar manner through instruction mapping in guest supervisor state. For **rfi**, execution of **rfi** is mapped to rfgi in guest supervisor state which will in turn use GSRR0 and GSRR1.

[2] Note that PIR is only mapped when performing reads from guest state. Attempted writes to PIR from guest state will produce an embedded hypervisor privilege exceptions. This allows the hypervisor software to track changes to the guest's PIR. Also note that direct write access to GPIR (without mapping) is also not allowed in guest state.

NOTE: Software Considerations

- Operating system software should not directly access the analogous guest state SPRs but should instead perform **mfspr** and **mtspr** to the non-guest state SPRs. For example, the operating system should not code **mfspr** r5,GSRR0, but should instead code **mfspr** r5,SRR0 and allow the processor will perform the mapping to access GSRR0. This makes the programming model the same whether the operating system is running as a guest under a hypervisor or is running bare-metal.
- Hypervisor software will directly access the analogous guest state SPRs when it is providing emulation for a guest operating system.
- Analogous guest state SPRs only exist for a number of SPRs that are considered to be high performance or allow the hypervisor to safely interrupt the guest. Other SPRs are required to be accessed for an operating system to effectively perform its functions as if it were running bare metal. These other SPRs are either directly accessible by guest supervisor state, or are required to be emulated by the hypervisor and cause an embedded hypervisor privilege exception when accessed in guest state.
- Some SPRs are only useful for the purposes of performing explicit hypervisor functions (for example MAS8 or LPIDR). These registers

are not intended to be used by an operating system and should not be emulated by the hypervisor.

3.3 Registers for Integer Operations

The following sections describe registers defined for integer computational instructions.

3.3.1 General-Purpose Registers (GPRs)

All implementations provide 32 GPRs (GPR0–GPR31) for integer operations. Instruction formats provide 5-bit fields for specifying the GPRs to be used in the execution of the instruction.

The architecture defines 32-bit GPRs for 32-bit implementations; however, category SPE extends GPRs to 64 bits to accommodate vector operands and embedded double-precision floating point operands.

Category SPE treats the 64-bit operands as consisting of two, 32-bit elements. The SPE embedded scalar double-precision floating-point instructions treat the GPRs as single 64-bit double-precision operands that accommodate the IEEE Std 754™. See the *SPE PEM* for more information.

For 64-bit implementations, category 64 bit is supported, and GPRs are each 64 bits.

3.3.2 Integer Exception Register (XER)

XER bits are set based on the operation of an instruction considered as a whole, not on intermediate results. (For example, the Subtract from Carrying instruction (**subfc**), the result of which is specified as the sum of three values, sets bits in the XER based on the entire operation, not on an intermediate sum.)

This figure shows the XER.

SPR 1 User

	32	33	34	35 – 56	57 – 63
R W	SO	OV	CA	—	SL
Reset	All zeros				

Figure 3-1. Integer Exception Register (XER)

This table describes XER bit definitions.

Table 3-4. XER Field Descriptions

Bits	Name	Description
32	SO	Summary overflow. Set when an instruction (except **mtspr**) sets the overflow bit (OV). Once set, SO remains set until it is cleared by **mtspr[XER]** or **mcrxr**. SO is not altered by compare instructions or by other instructions (except **mtspr[XER]** and **mcrxr**) that cannot overflow. Executing **mtspr[XER]**, supplying the values 0 for SO and 1 for OV, causes SO to be cleared and OV to be set.
33	OV	Overflow. Indicates whether an overflow occurred during execution of an instruction. Add, subtract from, and negate instructions having OE=1 set OV if the carry out of bit M is not equal to the carry out of bit M+1, and clear it otherwise. Multiply low and divide instructions having OE=1, set OV if the result cannot be represented in 64 bits (**mulld**, **divd**, **divdu**) or in 32 bits (**mullw**, **divw**, **divwu**), and set it to 0 otherwise. The OV bit is not altered by compare instructions, nor by other instructions (except **mtspr** [XER], and **mcrxr**) that cannot overflow.
34	CA	Carry. Add carrying, subtract from carrying, add extended, and subtract from extended instructions set CA if there is a carry out of bit 32 in 32-bit mode or bit 0 in 64-bit mode and clear it otherwise. CA can be used to indicate unsigned overflow for add and subtract operations that set CA. Shift right algebraic word instructions set CA if any 1 bits are shifted out of a negative operand and clear CA otherwise. Compare instructions and instructions that cannot carry (except Shift Right Algebraic , **mtspr[XER]**, and **mcrxr**) do not affect CA.
35–56	—	Reserved, should be cleared.
57–63	SL	String length. Power ISA defines this for category Move Assist instructions which are not supported by EIS.

3.4 Registers for Floating-Point Operations <FP>

This section describes registers that support FPR-based floating-point operations. Embedded, GPR-based floating-point registers are described in the *SPE PEM*.

3.4.1 Floating-Point Registers (FPR0–FPR31)

Floating-point instruction formats provide 5-bit fields for specifying FPRs used in instruction execution.

Each FPR contains 64 bits that support the floating-point format. Instructions that interpret FPR contents as floating-point values use double-precision format for this interpretation.

The computational instructions and the move and select instructions operate on data in FPRs and, except for compare instructions, place the result into an FPR, and optionally place status information into the CR.

Load and store double instructions are provided that transfer 64 bits of data between memory and the FPRs with no conversion. Load single instructions are provided to transfer and convert floating-point values in floating-point single format from memory to the same value in floating-point double format in the FPRs. Store single instructions are provided to transfer and convert floating-point values in floating-point double format from the FPRs to the same value in floating-point single format in memory.

Instructions are provided that manipulate the FPSCR and the CR explicitly. Some of these instructions copy data between an FPR and the FPSCR.

The computational instructions and the select instruction accept values from the FPRs in double format. For single-precision arithmetic instructions, all input values must be representable in single format; if they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the CR (if Rc = 1), are undefined.

3.4.2 Floating-Point Status and Control Register (FPSCR)

The FPSCR controls how floating-point exceptions are handled and records status resulting from floating-point operations. FPSCR[32–55] are status bits; FPSCR[56–63] are control bits.

This figure shows the FPSCR.

User

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
R W	FX	FEX	VX	OX	UX	ZX	XX	VXSNAN	VXISI	VXIDI	VXZDZ	VXIMZ	VXVC	FR	FI	C
Reset	All zeros															

	48	51	52	53	54	55	56	57	58	59	60	61	62	63
R W	FPCC		—	VXSOFT	VXSQRT	VXCVI	VE	OE	UE	ZE	XE	NI	RN	
Reset	All zeros													

Figure 3-2. Floating-Point Status and Control Register (FPSCR)

The exception bits, FPSCR[35–45,53–55], are sticky; once set they remain set until they are cleared by an **mcrfs, mtfsfi, mtfsf,** or **mtfsb0** instruction. Exception summary bits FPSCR[FX,FEX,VX] are not considered to be exception bits, and only FX is sticky.

FEX and VX are simply the ORs of other FPSCR bits, and so are not listed among the FPSCR bits affected by the various instructions.

This table describes the FPSCR fields.

Table 3-5. FPSCR Field Descriptions

Bits	Name	Description
32	FX	Floating-point exception summary. Every floating-point instruction, except **mtfsfi** and **mtfsf**, implicitly sets FX if that instruction causes any of the floating-point exception bits in the FPSCR to change from 0 to 1. **mcrfs**, **mtfsfi, mtfsf, mtfsb0**, and **mtfsb1** can alter FPSCR[FX] explicitly. **Note:** (Programming) FPSCR[FX] is defined not to be altered implicitly by **mtfsfi** and **mtfsf** because permitting these instructions to alter FPSCR[FX] implicitly could cause a paradox. An example is an **mtfsfi** or **mtfsf** that supplies 0 for FPSCR[FX] and 1 for FPSCR[OX] and executes when FPSCR[OX] = 0. See also the programming notes with the definition of these two instructions.
33	FEX	Floating-point enabled exception summary. FEX is the OR of all the floating-point exception bits masked by their respective enable bits. **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, and **mtfsb1** cannot alter FPSCR[FEX] explicitly.
34	VX	Floating-point invalid operation exception summary. VX is the OR of all the invalid operation exception bits. **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, and **mtfsb1** cannot alter FPSCR[VX] explicitly.
35	OX	Floating-point overflow exception. See Section 4.6.1.7.3, “Overflow Exception.”
36	UX	Floating-point underflow exception. See Section 4.6.1.7.4, “Underflow Exception.”
37	ZX	Floating-Point zero divide exception. See Section 4.6.1.7.2, “Zero Divide Exception.”
38	XX	Floating-point inexact exception. See Section 4.6.1.7.5, “Inexact Exception.” FPSCR[XX] is a sticky version of FPSCR[FI] (see below). Thus the following rules completely describe how FPSCR[XX] is set by a given instruction: • If the instruction affects FPSCR[FI], the new FPSCR[XX] value is obtained by ORing the old value of FPSCR[XX] with the new value of FPSCR[FI]. • If the instruction does not affect FPSCR[FI], the value of FPSCR[XX] is unchanged.

Table 3-5. FPSCR Field Descriptions (continued)

Bits	Name	Description
39	VXSNAN	Floating-point invalid operation exception (SNAN). See Section 4.6.1.7.1, “Invalid Operation Exception.”
40	VXISI	floating-point invalid operation exception ($\infty - \infty$). See Section 4.6.1.7.1, “Invalid Operation Exception.”.
41	VXIDI	Floating-point invalid operation exception ($\infty \div \infty$).See Section 4.6.1.7.1, “Invalid Operation Exception.”.
42	VXZDZ	Floating-point invalid operation exception ($0 \div 0$) See Section 4.6.1.7.1, “Invalid Operation Exception.”.
43	VXIMZ	Floating-point invalid operation exception ($\infty \times 0$). See Section 4.6.1.7.1, “Invalid Operation Exception.”.
44	VXVC	Floating-point invalid operation exception (invalid compare). See Section 4.6.1.7.1, “Invalid Operation Exception.”.
45	FR	Floating-point fraction rounded. The last arithmetic or rounding and conversion instruction incremented the fraction during rounding. See Section 3.4.3.6, “Rounding.” This bit is not sticky.
46	FI	Floating-point fraction inexact. The last arithmetic or rounding and conversion instruction either produced an inexact result during rounding or caused a disabled overflow exception. See Section 3.4.3.6, “Rounding.” FI is not sticky. See the definition of FPSCR[XX], above, regarding the relationship between FI and XX.
47–51	FPRF	Floating-point result flags. Arithmetic, rounding, and convert from integer instructions set FPRF based on the result placed into the target register and on the target precision, except that if any portion of the result is undefined, the value placed into FPRF is undefined. Floating-point compare instructions set FPRF based on the relative values of the operands compared. For convert to integer instructions, the value placed into FPRF is undefined. See Table 3-6. **Note:** (Programming) A single-precision operation that produces a denormalized result sets FPRF to indicate a denormalized number. When possible, single-precision denormalized numbers are represented in normalized double format in the target register.
47	C	Floating-point result class descriptor. Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits, to indicate the class of the result as shown in Figure 3-6.
48–51	FPCC	Floating-point condition code. Floating-point Compare instructions set one of the FPCC bits and clear the other three FPCC bits. Arithmetic, rounding, and conversion instructions may set the FPCC bits with the C bit to indicate the class of the result. In this case, the three high-order FPCC bits retain their relational significance indicating that the value is less than, greater than, or equal to zero. 48 Floating-point less than or negative (FL or <) 49 Floating-point greater than or positive (FG or >) 50 Floating-point equal or zero (FE or =) 51 Floating-point unordered or NaN (FU or ?)
52	—	Reserved, should be cleared.
53	VXSOFT	Floating-point invalid operation exception (software request). Can be altered only by **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, or **mtfsb1**.
54	VXSQRT	Floating-point invalid operation exception (invalid square root). Note that VXSQRT is defined even for implementations that do not support either of the two optional instructions that set it, **fsqrt**[**.**] and **frsqrte**[**.**]. Defining it for all implementations gives software a standard interface for handling square root exceptions. If an implementation does not support **fsqrt**[**.**] or **frsqrte**[**.**], software can simulate the instruction and set VXSQRT to reflect the exception.
55	VXCVI	Floating-point invalid operation exception (invalid integer convert)
56	VE	Floating-point invalid operation exception enable
57	OE	Floating-point overflow exception enable
58	UE	Floating-point underflow exception enable
59	ZE	Floating-point zero divide exception enable
60	XE	Floating-point inexact exception enable

Table 3-5. FPSCR Field Descriptions (continued)

Bits	Name	Description
61	NI	Floating-point non-IEEE mode. If NI = 1, the remaining FPSCR bits may have meanings other than those given in this document and results of floating-point operations need not conform to IEEE 754. If the IEEE-754-conforming result of a floating-point operation would be a denormalized number, the result of that operation is 0 (with the same sign as the denormalized number) if FPSCR[NI] = 1 and other requirements specified in the user's manual for the implementation are met. The other effects of setting NI may differ among implementations. Setting NI is intended to permit results to be approximate and to cause performance to be more predictable and less data-dependent than when NI = 0. For example, in non-IEEE mode, an implementation returns 0 instead of a denormalized number and may return a large number instead of an infinity. In non-IEEE mode an implementation should provide a means for ensuring that all results are produced without software assistance (that is, without causing an enabled exception type program interrupt or a floating-point unimplemented instruction exception type program interrupt and without invoking an emulation assist). The means may be controlled by one or more other FPSCR bits (recall that the other FPSCR bits have implementation-dependent meanings if NI = 1).
62–63	RN	Floating-point rounding control (RN). 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward –infinity

This table describes floating-point result flags.

Table 3-6. Floating-Point Result Flags

Result Flags					Result Value Class
C	<	>	=	?	
1	0	0	0	1	Quiet NaN
0	1	0	0	1	–Infinity
0	1	0	0	0	–Normalized number
1	1	0	0	0	–Denormalized number
1	0	0	1	0	–Zero
0	0	0	1	0	+Zero
1	0	1	0	0	+Denormalized number
0	0	1	0	0	+Normalized number
0	0	1	0	1	+Infinity

3.4.3 Floating-Point Data

This section includes the following:

- Section 3.4.3.1, “Data Format”
- Section 3.4.3.2, “Value Representation”
- Section 3.4.3.3, “Sign of Result”
- Section 3.4.3.4, “Normalization and Denormalization”

- Section 3.4.3.5, “Data Handling and Precision”
- Section 3.4.3.6, “Rounding”

3.4.3.1 Data Format

This architecture defines the representation of a floating-point value in two different binary fixed-length formats. The format may be a 32-bit single format for a single-precision value or a 64-bit double format for a double-precision value. The single format may be used for data in storage. The double format may be used for data in storage and for data in FPRs.

The lengths of the exponent and the fraction fields differ between these two formats.

These figures show the structure of the single and double formats.

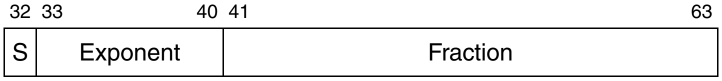


Figure 3-3. Floating-Point Single Format

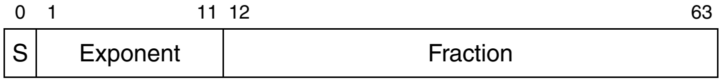


Figure 3-4. Floating-Point Double Format

Representation of numeric values in the floating-point formats consists of the following three fields:

- Sign bit (S)
- Exponent (EXP) + bias
- Fraction

The significand consists of a leading implied bit concatenated on the right with the fraction. This leading implied bit is 1 for normalized numbers and 0 for denormalized numbers and is located in the unit bit position (that is, the first bit to the left of the binary point). This table lists the parameters used to specify values representable within the two floating-point formats.

Table 3-7. IEEE-754 Floating-Point Fields

Component	Format	
	Single	Double
Exponent bias	+127	+1023
Maximum exponent	+127	+1023
Minimum exponent	-126	-1022

Table 3-7. IEEE-754 Floating-Point Fields

Component	Format	
	Single	Double
Widths (bits)		
Format	32	64
Sign	1	1
Exponent	8	11
Fraction	23	52
Significand	24	53

The architecture requires that FPRs support only floating-point double format.

3.4.3.2 Value Representation

3.4.3.2.1 Numeric and Non-Numeric Values

The architecture defines numeric and non-numeric values representable within each of the two supported formats. The numeric values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. Representable non-numeric values are the infinities and the Not a Numbers (NaNs). The infinities are adjoined to the real numbers, but are not numbers themselves, and the standard rules of arithmetic do not hold when they are used in an operation. They are related to the real numbers by order alone. It is possible however to define restricted operations among numbers and infinities as defined below.

This figure shows the relative location on the real number line for each of the defined entities.

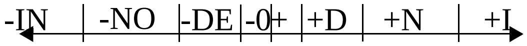


Figure 3-5. Approximation to Real Numbers

The NaNs are not related to the numeric values or infinities by order or value; they are encodings used to convey diagnostic information such as the representation of uninitialized variables.

3.4.3.2.2 Floating-Point Values

The following is a description of the different floating-point values defined in the architecture:

- Binary floating-point numbers.
  Machine representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.
- Normalized numbers (± NOR).
  These are values that have a biased exponent value in the range:
  - 1 to 254 in single format
  - 1 to 2046 in double format

These are values in which the implied unit bit is 1.

Normalized numbers are interpreted as follows, where *s* is the sign, *E* is the unbiased exponent, and *1.fraction* is the significand, which is composed of a leading unit bit (the implied bit) and a fraction part:

- NOR = $(-1)^s$ x 2^E x (1.fraction)

Ranges covered by the magnitude (M) of a normalized floating-point number are approximately as follows:

- Single format. $1.2\text{x}10^{-38} \leq M \leq 3.4\text{x}10^{38}$
- Double Format. $2.2\text{x}10^{-308} \leq M \leq 1.8\text{x}10^{308}$
- Zero values (± 0).
  These values have a biased exponent value of zero and a fraction value of zero. Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (that is, comparison regards +0 as equal to -0).
- Denormalized numbers (± DEN).
  These are values that have a biased exponent value of zero and a nonzero fraction value. They are nonzero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is 0. Denormalized numbers are interpreted as DEN = $(-1)^s$ x $2^{E_{min}}$ x (0.fraction), where E_{min} is the minimum representable exponent value (-126 for single-precision, -1022 for double-precision).
- Infinities (± ∞).
  These values have the maximum biased exponent value—255 in single format, 2047 in double format, and a zero fraction value.
  They are used to approximate values greater in magnitude than the maximum normalized value.Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and the real numbers can be related by ordering in the affine sense: $-\infty <$ every finite number $< +\infty$. Arithmetic on infinities is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 4.6.1.7.1, "Invalid Operation Exception." For comparison operations, +Infinity compares equal to +Infinity and -Infinity compares equal to -Infinity.
- Not a Numbers (NaNs).
  These values have the maximum biased exponent value and a nonzero fraction value. The sign bit is ignored (that is, NaNs are neither positive nor negative). If the high-order bit of the fraction field is 0 then the NaN is a Signaling NaN; otherwise it is a Quiet NaN.
- Signaling NaNs are used to signal exceptions when they appear as operands of computational instructions. Quiet NaNs are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when invalid operation exception is disabled (FPSCR[VE]=0). Quiet NaNs propagate through all floating-point operations except ordered comparison, floating round to single-precision, and conversion to integer. Quiet NaNs do not signal exceptions, except for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of floating-point operations, and used to convey diagnostic information to help identify results from invalid operations.
  When a QNaN is the result of a floating-point operation because one of the operands is a NaN or

because a QNaN was generated due to a disabled invalid operation exception, the following rule determines the NaN with the high-order fraction bit set that is to be stored as the result.

```
if (frA) is a NaN
   then frD ← (frA)
   else if (frB) is a NaN
     then if instruction is frsp
         then frD ← $(\mathbf{fr}B)_{0:34} \,||\, {}^{29}0$
         else frD ← (frB)
     else if (frC) is a NaN
         then frD ← (frC)
         else if generated QNaN
            then frD ← generated QNaN
```

If the operand specified by **fr**A is a NaN, that NaN is stored as the result. Otherwise, if the operand specified by **fr**B is a NaN (if the instruction specifies an **fr**B operand), that NaN is stored as the result, with the low-order 29 bits of the result cleared if the instruction is **frsp**. Otherwise, if the operand specified by **fr**C is a NaN (if the instruction specifies an **fr**C operand), that NaN is stored as the result. Otherwise, if a QNaN was generated due to a disabled invalid operation exception, then that QNaN is stored as the result. If a QNaN is to be generated as a result, then the QNaN generated has a sign bit of 0, an exponent field of all 1s, and a high-order fraction bit of 1 with all other fraction bits 0. Any instruction that generates a QNaN as the result of a disabled invalid operation exception generates this QNaN (that is, 0x7FF8_0000_0000_0000).

A double-precision NaN is considered to be representable in single format if and only if the low-order 29 bits of the double-precision NaN's fraction are zero.

3.4.3.3 Sign of Result

The following rules govern the sign of the result of an arithmetic, rounding, or conversion operation, when the operation does not yield an exception. They apply even if operands or results are zeros or infinities.

- The sign of the result of an add operation is the sign of the operand having the larger absolute value. If both operands have the same sign, the sign of the result of an add operation is the same as the sign of the operands. The sign of the result of the subtract operation $x-y$ is the same as the sign of the result of the add operation $x+(-y)$.

  When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except Round toward -Infinity, in which mode the sign is negative.
- The sign of the result of a multiply or divide operation is the Exclusive OR of the signs of the operands.
- The sign of the result of a square root or reciprocal square root estimate operation is always positive, except that the square root of -0 is -0 and the reciprocal square root of -0 is -Infinity.
- The sign of the result of a round to single-precision, or convert from integer, or round to integer operation is the sign of the operand being converted.

For the multiply-add instructions, these rules are applied first to the multiply and then to the add or subtract operation (one of the inputs to the add or subtract operation is the result of the multiply operation).

3.4.3.4 Normalization and Denormalization

The intermediate result of an arithmetic or **frsp** instruction may require normalization or denormalization as described below. Normalization and denormalization do not affect the sign of the result.

When an arithmetic or rounding instruction produces an intermediate result which carries out of the significand, or in which the significand is nonzero but has a leading zero bit, it is not a normalized number and must be normalized before it is stored. For the carry-out case, the significand is shifted right one bit, with a one shifted into the leading significand bit, and the exponent is incremented by one. For the leading-zero case, the significand is shifted left while decrementing its exponent by one for each bit shifted, until the leading significand bit becomes one. The guard and round bits (see Section E.1, “Execution Model for IEEE-754 Operations”) participate in the shift with zeros shifted into the round bit. The exponent is regarded as if its range were unlimited.

After normalization, or if normalization was not required, the intermediate result may have a nonzero significand and an exponent value that is less than the minimum value that can be represented in the format specified for the result. In this case, the intermediate result is said to be “Tiny” and the stored result is determined by the rules described in Section 4.6.1.7.4, “Underflow Exception.” These rules may require denormalization.

A number is denormalized by shifting its significand right while incrementing its exponent by 1 for each bit shifted, until the exponent is equal to the format’s minimum value. Loss of any significant bits in this process indicates loss of accuracy (see Section 4.6.1.7.4, “Underflow Exception”), which is signaled by an underflow exception is signaled.

3.4.3.5 Data Handling and Precision

Most floating-point operations, including all computational, move, and select instructions, use the floating-point double format to represent data in the FPRs. Single-precision and integer-valued operands may be manipulated using double-precision operations. Instructions are provided to coerce these values from a double format operand. Instructions are also provided for manipulations which do not require double-precision. In addition, instructions are provided to access a true single-precision representation in storage, and a fixed-point integer representation in GPRs.

3.4.3.5.1 Single-Precision Operands

For single format data, a format conversion from single to double is performed when loading from storage into an FPR and a format conversion from double to single is performed when storing from an FPR to storage. No floating-point exceptions are caused by these instructions. An instruction is provided to explicitly convert a double format operand in an FPR to single-precision.

This table shows the four types of instruction with which floating-point single-precision is enabled.

NOTE

> If the result of a load floating-point single, floating round to single-precision, or single-precision arithmetic instruction is stored in an FPR, the low-order 29 FRACTION bits are zero.

Table 3-8. Four Types of Instruction for Enabling Floating-Point Single-Precision

Instruction Type	Definition
Load floating-point single	This form of instruction accesses a single-precision operand in single format in storage, converts it to double format, and loads it into an FPR. No floating-point exceptions are caused by these instructions.
Floating round to single-precision	This form of instruction rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits, and places that operand into an FPR in double format. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of the floating round to single-precision instruction, this operation does not alter the value.
Single-precision arithmetic	This form of instruction takes operands from the FPRs in double format, performs the operation as if it produced an intermediate result having infinite precision and unbounded exponent range, and then coerces this intermediate result to fit in single format. Status bits, in the FPSCR and optionally in the CR, are set to reflect the single-precision result. The result is then converted to double format and placed into an FPR. The result lies in the range supported by the single format. **Note:** All input values must be representable in single format; if they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the CR (if Rc=1), are undefined.
Store floating-point single	This form of instruction converts a double-precision operand to single format and stores that operand into storage. No floating-point exceptions are caused by these instructions. (The value being stored is effectively assumed to be the result of an instruction of one of the preceding three types.)

NOTE: Software Considerations

The floating round to single-precision instruction is provided to allow value conversion from double-precision to single-precision with appropriate exception checking and rounding. This instruction should be used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions and by **fcfid**) to single-precision values prior to storing them into single format storage elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions are already single-precision values and can be stored directly into single format storage elements, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by a Floating Round to Single-Precision instruction.

NOTE: Software Considerations

- A single-precision value can be used in double-precision arithmetic operations. The reverse is true only if the double-precision value is representable in single format.
- Some implementations may perform single-precision arithmetic faster than double-precision arithmetic. Therefore, if double-precision accuracy is not required, single-precision data and instructions should be used.

3.4.3.5.2 Integer-Valued Operands

Instructions are provided to round floating-point operands to integer values in floating-point format. To facilitate the exchange of data between the floating-point and fixed-point processors, instructions are provided to convert between floating-point double format and fixed-point integer format in an FPR. Computation on integer-valued operands may be performed using arithmetic instructions of the required precision. (The results may not be integer values.) The two groups of instructions provided specifically to support integer-valued operands are described below.

- Floating round to integer.
  The floating round to integer instructions round a double-precision operand to an integer value in floating-point double format. These instructions may cause invalid operation (VXSNAN) exceptions. See Section 3.4.3.6, "Rounding," and Section E.1, "Execution Model for IEEE-754 Operations," for more information about rounding.
- Floating convert to/from integer.
  The floating convert to integer instructions convert a double-precision operand to a 32-bit or 64-bit signed fixed-point integer format. Variants are provided both to perform rounding based on the value of FPSCR[RN] and to round toward zero. These instructions may cause invalid operation (VXSNaN, VXCVI) and inexact exceptions. The floating convert from integer instruction converts a 64-bit signed fixed-point integer to a double-precision floating-point integer. Because of the limitations of the source format, only an Inexact exception may be generated.

3.4.3.6 Rounding

The material in this section applies to operations that have numeric operands (that is, operands that are not infinities or NaNs). Rounding the intermediate result of such an operation may cause an overflow exception, an underflow exception, or an inexact exception. The remainder of this section assumes that the operation causes no exceptions and that the result is numeric. See Section 3.4.3.2, "Value Representation," and Section 4.6.1.7, "Floating-Point Exceptions," for the cases not covered here.

3.4.3.6.1 Arithmetic, Rounding, and Conversion Instructions

The arithmetic and rounding and conversion instructions round their intermediate results. With the exception of the estimate instructions, these instructions produce an intermediate result that can be regarded as having infinite precision and unbounded exponent range. All but two groups of these instructions normalize or denormalize the intermediate result prior to rounding and then place the final result into the target FPR in double format. The floating round to integer and floating convert to integer instructions with biased exponents ranging from 1022 through 1074 are prepared for rounding by repetitively shifting the significand right one position and incrementing the biased exponent until it reaches a value of 1075. (Intermediate results with biased exponents 1075 or larger are already integers, and with biased exponents 1021 or less round to zero.) After rounding, the final result for floating round to integer is normalized and put in double format, and for floating convert to integer is converted to a signed fixed-point integer.

FPSCR[FR,FI] generally indicate the results of rounding. Each of the instructions which rounds its intermediate result sets these bits. If the fraction is incremented during rounding then FR is set; otherwise FR is cleared. If the result is inexact then FI is set, otherwise FI is set to zero. The round to integer instructions are exceptions to this rule, clearing FR and FI. The estimate instructions set FR and FI to undefined values. The remaining floating-point instructions do not alter FR and FI.

3.4.3.6.2 Rounding Modes

Four user-selectable rounding modes are provided through the floating-point rounding control field in the FPSCR. See Section 3.4.2, “Floating-Point Status and Control Register (FPSCR).” These are encoded in Table 3-9, which also describes the rules that correspond with each setting.

Let Z be the intermediate arithmetic result or the operand of a convert operation. If Z can be represented exactly in the target format, then the result in all rounding modes is Z as represented in the target format. If Z cannot be represented exactly in the target format, let Z1 and Z2 bound Z as the next larger and next smaller numbers representable in the target format. Then Z1 or Z2 can be used to approximate the result in the target format.

Table 3-9. Rounding Mode Rules

RN	Rounding Mode	Rule
00	Round to Nearest	Choose the value that is closer to Z (Z1 or Z2). In case of a tie, choose the one that is even (least significant bit 0).
01	Round toward Zero	Choose the smaller in magnitude (Z1 or Z2).
10	Round toward +Infinity	Choose Z1.
11	Round toward -Infinity	Choose Z2.

This figure shows the relation of Z, Z1, and Z2 in this case.

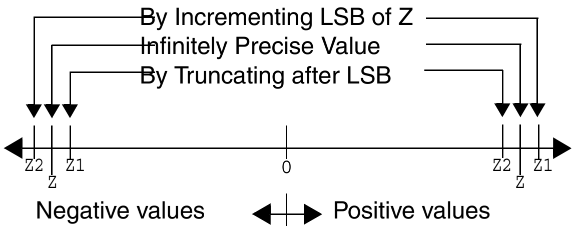


Figure 3-6. Selection of Z1 and Z2

For a detailed explanation of rounding, see Section E.1, “Execution Model for IEEE-754 Operations.”

3.5 Registers for Branch Operations

This section describes registers used by branch and CR operations.

3.5.1 Condition Register (CR)

The 32-bit CR reflects the result of certain operations and provides a mechanism for testing and branching.

This figure shows the 32-bit CR.

User

	32 35	36 39	40 43	44 47	48 51	52 55	56 59	60 63
R W	CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
Reset	All zeros							

Figure 3-7. Condition Register (CR)

CR bits are grouped into eight 4-bit fields, CR0–CR7, which are set as follows:

- Specified CR fields can be set by a move to the CR from a GPR (**mtcrf**, **mtocrf**).
- A specified CR field can be set by a move to the CR from another CR field (**mcrf**), from the FPSCR (**mcrfs**), or from the XER (**mcrxr**), or from the FPSCR (**mcrfs**).
- CR0 can be set as the implicit result of an integer instruction.
- CR1 can be set as the implicit result of a floating-point instruction.
- CR6 can be set as the implicit result of a floating-point instruction. <Vector>
- A specified CR field can be set as the result of either an integer compare instruction.
- A specified CR field can be set as the result of an embedded floating-point or SPE compare instruction. <SPE, SP.FD, SP.FS, SP.FV>

Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits (see Section 4.6.1.9.3, “Condition Register (CR) Logical Instructions”).

Note that instructions that access CR bits (for example, Branch Conditional (**bc**), CR logicals, and Move to Condition Register Field (**mtcrf**)) determine the bit position by adding 32 to the operand value. For example, in conditional branch instructions, the BI operand accesses bit BI + 32, as shown in this table.

Table 3-10. BI Operand Settings for CR Fields

CR*n* Bits	CR Bits	BI	Description
CR0[0]	32	00000	Negative (LT)—Set when the result is negative. For SPE compare and test instructions: Set if the high-order element of **r**A is equal to the high-order element of **r**B; cleared otherwise.
CR0[1]	33	00001	Positive (GT)—Set when the result is positive (and not zero). For SPE compare and test instructions: Set if the low-order element of **r**A is equal to the low-order element of **r**B; cleared otherwise.

Table 3-10. BI Operand Settings for CR Fields (continued)

CR*n* Bits	CR Bits	BI	Description
CR0[2]	34	00010	Zero (EQ)—Set when the result is zero. For SPE compare and test instructions: Set to the OR of the result of the compare of the high and low elements.
CR0[3]	35	00011	Summary overflow (SO). Copy of XER[SO] at the instruction’s completion. For SPE compare and test instructions: Set to the AND of the result of the compare of the high and low elements.
CR1[0]	36	00100	Copy of FPSCR[FX] at the instruction’s completion. Negative (LT) For SPE and SPFP compare and test instructions: Set if the high-order element of **r**A is equal to the high-order element of **r**B; cleared otherwise.
CR1[1]	37	00101	Copy of FPSCR[FEX] at the instruction’s completion. Positive (GT) For SPE and SPFP compare and test instructions: Set if the low-order element of **r**A is equal to the low-order element of **r**B; cleared otherwise.
CR1[2]	38	00110	Copy of FPSCR[VX] at the instruction’s completion. Zero (EQ) For SPE and SPFP compare and test instructions: Set to the OR of the result of the compare of the high and low elements.
CR1[3]	39	00111	Copy of FPSCR[OX] at the instruction’s completion. Summary overflow (SO) For SPE and SPFP compare and test instructions: Set to the AND of the result of the compare of the high and low elements.
CR*n*[0]	40 44 48 52 56 60	01000 01100 10000 10100 11000 11100	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A < **fr**B. For SPE and SPFP compare and test instructions: Set if the high-order element of **r**A is equal to the high-order element of **r**B; cleared otherwise.
CR*n*[1]	41 45 49 53 57 61	01001 01101 10001 10101 11001 11101	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A > **fr**B. For SPE and SPFP compare and test instructions: Set if the low-order element of **r**A is equal to the low-order element of **r**B; cleared otherwise.
CR*n*[2]	42 46 50 54 58 62	01010 01110 10010 10110 11010 11110	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B. For SPE and SPFP compare and test instructions: Set to the OR of the result of the compare of the high and low elements.
CR*n*[3]	43 47 51 55 59 63	01011 01111 10011 10111 11011 11111	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN. For SPE and SPFP vector compare and test instructions: Set to the AND of the result of the compare of the high and low elements.

3.5.1.1 CR Setting for Integer Instructions

For all integer instructions in which the Rc bit is defined and set (sometimes referred to as the *record* form), and for **addic.**, **andi.**, and **andis.**, CR0[32–34] are set by signed comparison of the result (low order 32 bits in 32-bit mode and 64-bits in 64-bit mode) to zero; CR[35] is copied from the final state of XER[SO].

```
if (64-bit mode)
   then M ← 0
   else M ← 32
```

```
if      (target_register)M:63 < 0 then c ← 0b100
else if (target_register)M:63 > 0 then c ← 0b010
else                                   c ← 0b001
CR0 ← c || XERSO
```

The value of any undefined portion of the result is undefined, and the value placed into the first three bits of CR0 is undefined. CR0 bits are interpreted as described in this table.

Table 3-11. CR0 Bit Descriptions

CR Bit	Name	Description
32	Negative (LT)	Bit 32 of the result is equal to one.
33	Positive (GT)	Bit 32 of the result is equal to zero, and at least one of bits 33–63 of the result is non-zero.
34	Zero (EQ)	Bits 32–63 of the result are equal to zero.
35	Summary overflow (SO)	This is a copy of the final state of XER[SO] at the completion of the instruction.

Note that CR0 may not reflect the true (infinitely precise) result if overflow occurs. See Section 4.6.1.1.1, "Integer Arithmetic Instructions."

3.5.1.2 CR Setting for Store Conditional Instructions

CR0 is also set by the integer store conditional instructions, **stbcx.** <ER>, **sthcx.** <ER>, **stwcx.**, **stdcx.** <64>. See instruction descriptions in Chapter 4, "Instruction Model," for detailed descriptions of how CR0 is set.

3.5.1.3 CR Setting for Floating-Point Instructions

For all floating-point instructions in which the Rc bit is defined and set, CR1 (CR[36–39]) is copied from FPSCR[32–35]. These bits are interpreted as shown in this table.

Table 3-12. CR Setting for Floating-Point Instructions

Bit	Name	Description
36	FX	Floating-point exception summary. Copy of final state of FPSCR[FX] at instruction completion.
37	FEX	Floating-point enabled exception summary. Copy of final state of FPSCR[FEX] at instruction completion.
38	VX	Floating-point invalid operation exception summary. Copy of final state of FPSCR[VX] at completion.
39	OX	Floating-point overflow exception. Copy of final state of FPSCR[OX] at instruction completion.

3.5.1.4 CR Setting for Compare Instructions

For compare instructions, a CR field specified by the BI field in the instruction is set to reflect the result of the comparison, as shown in the following table. A complete description of how the bits are set is given in Section 4.6.1.1.2, "Integer Compare Instructions."

Table 3-13. CR Setting for Compare Instructions

CR*n* Bit	Bit Expression	CR Bits		BI		Description
		AIM (BI Operand)	Power ISA	0–2	3–4	
CR*n*[0]	**4 * cr0 + lt** (or **lt**) **4 * cr1 + lt** **4 * cr2 + lt** **4 * cr3+ lt** **4 * cr4 + lt** **4 * cr5 + lt** **4 * cr6 + lt** **4 * cr7 + lt**	0 4 8 12 16 20 24 28	32 36 40 44 48 52 56 60	000 001 010 011 100 101 110 111	00	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions: frA < frB.
CR*n*[1]	**4 * cr0 + gt** (or **gt**) **4 * cr1 + gt** **4 * cr2 + gt** **4 * cr3+ gt** **4 * cr4 + gt** **4 * cr5 + gt** **4 * cr6 + gt** **4 * cr7 + gt**	1 5 9 13 17 21 25 29	33 37 41 45 49 53 57 61	000 001 010 011 100 101 110 111	01	Greater than or floating-point greater than (GT, FG). For integer compare instructions: rA > SIMM or rB (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: frA > frB.
CR*n*[2]	**4 * cr0 + eq** (or **eq**) **4 * cr1 + eq** **4 * cr2 + eq** **4 * cr3+ eq** **4 * cr4 + eq** **4 * cr5 + eq** **4 * cr6 + eq** **4 * cr7 + eq**	2 6 10 14 18 22 26 30	34 38 42 46 50 54 58 62	000 001 010 011 100 101 110 111	10	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: frA = frB.
CR*n*[3]	**4 * cr0 + so/un** (or **so/un**) **4 * cr1 + so/un** **4 * cr2 + so/un** **4 * cr3 + so/un** **4 * cr4 + so/un** **4 * cr5 + so/un** **4 * cr6 + so/un** **4 * cr7 + so/un**	3 7 11 15 19 23 27 31	35 39 43 47 51 55 59 63	000 001 010 011 100 101 110 111	11	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at instruction completion. For floating-point compare instructions, one or both of frA and frB is a NaN.

3.5.1.5 CR Bit Settings in VLE Mode <VLE>

The VLE extension implements the entire CR, but some comparison operations and all branch instructions are limited to using CR0–CR3. However, all CR field and logical operations are provided.

CR bits are grouped into eight 4-bit fields, CR0–CR7, which are set in one of the following ways.

- Specified CR fields can be set by a move to the CR from a GPR (**mtcrf**).
- A specified CR field can be set by a move to the CR from another CR field (**e_mcrf**).
- CR 0 field can be set as the implicit result of an integer instruction.
- A specified CR field can be set as the result of an integer compare instruction.
- CR 0 field can be set as the result of an integer bit test instruction.

Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits

For general information about the VLE, see the *VLE PEM*.

3.5.1.5.1 CR Settings for Integer Instructions in VLE <VLE>

For all integer word instructions in which the Rc bit is defined and set, and for **addic.**, the first three bits of CR 0 field (CR[32–34]) are set by signed comparison of bits 32–63 of the result to zero, and the fourth bit of CR 0 field (CR[35]) is copied from the final state of XER[SO].

```
if      (target_register)$_{32:63}$ < 0 then c ← 0b100
else if (target_register)$_{32:63}$ > 0 then c ← 0b010
else                                     c ← 0b001
CR0 ← c || XER$_{SO}$
```

If any portion of the result is undefined, the value placed into the first three bits of CR 0 field is undefined.

This table shows how the bits of CR 0 field are interpreted.

Table 3-14. CR0 Encodings

CR Bit	Description
32	Negative (LT). Bit 32 of the result is equal to 1.
33	Positive (GT). Bit 32 of the result is equal to 0 and at least one of bits 33–63 of the result is non-zero.
34	Zero (EQ). Bits 32–63 of the result are equal to 0.
35	Summary overflow (SO). This is a copy of the final state XER[SO] at the completion of the instruction.

3.5.1.5.2 CR Setting for Compare Instructions in VLE <VLE>

For compare instructions, a CR field specified by the **cr**D operand in the instruction for the **e_cmph**, **e_cmphl**, **e_cmpi**, and **e_cmpli** instructions, or CR0 for the **e_cmp16i**, **e_cmph16i**, **e_cmphl16i**, **e_cmpl16i**, **se_cmp**, **se_cmph**, **se_cmphl**, **se_cmpi**, and **se_cmpli** instructions is set to reflect the result of the comparison. The CR field bits are interpreted as shown in the following table. A complete description of how the bits are set is given in the *VLE PEM*.

Table 3-15. Condition Register Setting for Compare Instructions

CR Bit	Description
4×CRD + 32	Less than (LT). For signed-integer compare, GPR(**r**A or **r**X) < SCI8 or SI or GPR(**r**B or **r**Y). For unsigned-integer compare, GPR(**r**A or **r**X) $<_u$ SCI8 or UI or UI5 or GPR(**r**B or **r**Y). For floating-point compare instructions, (**fr**A) < (**fr**B).
4×CRD + 33	Greater than (GT). For signed-integer compare, GPR(**r**A or **r**X) > SCI8 or SI or UI5 or GPR(**r**B or **r**Y). For unsigned-integer compare, GPR(**r**A or **r**X) $>_u$ SCI8 or UI or UI5 or GPR(**r**B or **r**Y). For floating-point compare instructions, (**fr**A) > (**fr**B).
4×CRD + 34	Equal (EQ). For integer compare, GPR(**r**A or **r**X) = SCI8 or UI5 or SI or UI or GPR(**r**B or **r**Y). For floating-point compare instructions, (**fr**A) = (**fr**B).
4×CRD + 35	Summary overflow (SO). For integer compare, this is a copy of the final state of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of (**fr**A) and (**fr**B) is a NaN.

3.5.1.5.3 CR Setting for the VLE Bit Test Instruction <VLE>

The Bit Test Immediate instruction, **se_btsti**, also sets CR 0 field. See the instruction description and Section 3.6.6.4.5, “Integer Compare and Bit Test Instructions.”

For general information about the VLE, see the *VLE PEM*.

3.5.2 Link Register (LR)

The LR, shown in the following figure, can be used to provide the branch target address for a Branch Conditional to LR (**bclr***x*) instruction, and it holds the return address after branch and link instructions (LK=1).

NOTE

The LR is an SPR and can be directly addressed with **mfspr**/**mtspr** instructions.

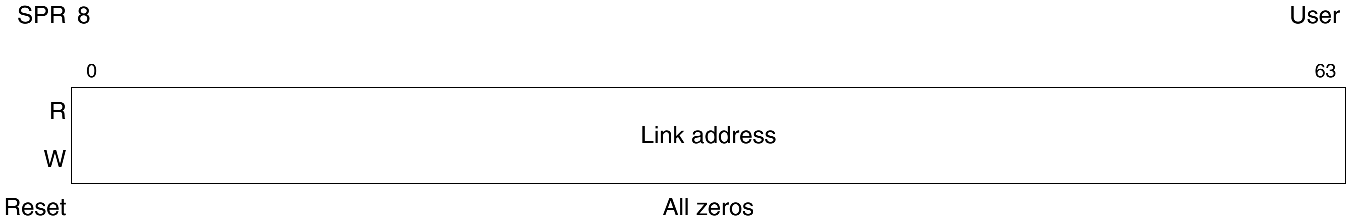


Figure 3-8. Link Register (LR)

The LR contents are read into a GPR using **mfspr**. The contents of a GPR can be written to the LR using **mtspr**. LR[62–63] are ignored by **bclr** instructions.

3.5.2.1 Link Register Usage in VLE Mode <VLE>

Category VLE defines a subset of all variants of conditional branches involving the LR, as shown in the following table. VLE instructions can reside on half-word boundaries so for processors that implement category VLE, LR[62] is used by **bclr***x* instructions..

Table 3-16. Branch to Link Register Instruction Comparison

Power ISA		VLE	
Instruction	**Syntax**	**Instruction**	**Syntax**
Branch Conditional to Link Register Branch Conditional to Link Register & Link	**bclr** BO,BI **bclrl** BO,BI	Branch (Absolute) to Link Register Branch (Absolute) to Link Register & Link	**se_blr** **se_blrl**
Branch Conditional & Link	**e_bcl** BO,BI,BD	Branch Conditional & Link	**e_bcl** BO32,BI32,BD15
		Branch (Absolute) & Link	**e_bl** BD24 **se_bl** BD8

3.5.3 Count Register (CTR)

CTR can be used to hold a loop count that can be decremented and tested during execution of branch instructions that contain an appropriately encoded BO field. If the CTR value is 0 before being decremented, it is –1 afterward. The entire CTR can be used to hold the branch target address for a Branch Conditional to CTR (**bcctr***x*) instruction. CTR[62–63] are ignored by **bcctr***x* instructions.

NOTE

The CTR is an SPR and can be directly addressed with **mfspr**/**mtspr** instructions.

This figure shows the Count register.

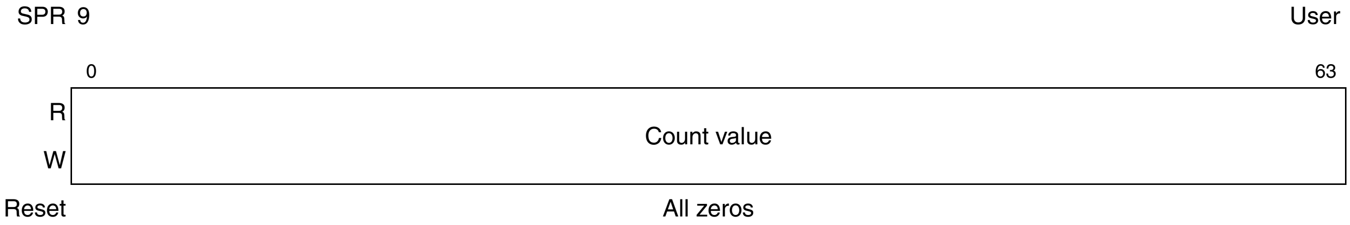


Figure 3-9. Count Register (CTR)

3.5.3.1 Count Register Usage in VLE Mode <VLE>

VLE defines a subset of the variants of conditional branches involving the CTR, as shown in the following table. VLE instructions can reside on half-word boundaries so for processors that implement category VLE CTR[62] is used by **bcctr***x* instructions.

Table 3-17. Branch to Count Register Instruction Comparison

Power ISA		VLE	
Instruction	**Syntax**	**Instruction**	**Syntax**
Branch Conditional to Count Register Branch Conditional to Count Register & Link	**bcctr** BO,BI **bcctrl** BO,BI	Branch (Absolute) to Count Register Branch (Absolute) to Count Register & Link	**se_bctr** **se_bctrl**

3.5.4 Branch Unit Control and Status Register (BUCSR)

The BUCSR, shown in the following figure, is an implementation-specific register used for general control and status of the branch prediction mechanisms supported by the core. This register is not implemented by all processors and generally contains more implementation specific fields than what are presented here.

Writing to BUCSR requires synchronization.

This register is hypervisor privileged.

SPR 1013 Hypervisor

	32–62	63
R W	—	BPEN
Reset	All zeros	

Figure 3-10. Branch Unit Control and Status Register (BUCSR)

This table describes the BUCSR fields.

Table 3-18. BUCSR Field Descriptions

Bits	Name	Description
32–62	—	Reserved for implementation use.
63	BPEN	Branch prediction enable 0 Branch prediction disabled 1 Branch prediction enabled

3.6 Processor Control Registers

This section addresses machine state, processor ID, and processor version registers.

3.6.1 Machine State Register (MSR)

The MSR, shown in Figure 3-11, defines the processor state; that is, enabling and disabling of interrupts and exceptions, address translation for instruction and data memory accesses, enabling and disabling some resources, and specifying the privilege mode.

3.6.1.1 Interrupt Handling and the MSR

MSR contents are saved, altered, and restored automatically by the interrupt handling mechanism. MSR contents are automatically copied into a save/restore register 1 when an interrupt is taken. The save/restore register (*x*SRR1), which gets a copy of the MSR, depends on which interrupt is taken and whether the interrupt is directed to the hypervisor state or the guest supervisor state. The MSR is also altered when an interrupt is taken. The new value of the MSR depends on which interrupt is taken and whether the interrupt is directed to the hypervisor state. See Section 7.7, “Interrupt Processing,” for complete details about interrupt MSR and save/restore register settings. When a return from interrupt instruction executes (**rfi**, **rfgi**, **rfci**, **rfdi**, or **rfmci**), MSR contents are restored from the corresponding *x*SRR1.

3.6.1.2 Using the mfmsr Instruction

MSR contents are read into a GPR using **mfmsr**. A GPR's contents can be written to MSR by using an **mtmsr**. The write MSR external enable instructions (**wrtee** and **wrteei**) are used to set or clear MSR[EE] without affecting other MSR bits.

3.6.1.3 Using the mtmsr Instruction

Changing CM, PR, GS, IS, or DS using the **mtmsr** instruction requires a context-synchronizing operation before the effects of the change are guaranteed to be visible. Prior to the context synchronization, these bits can change at any time and with any combination. Changes in CM, GS, or IS can cause an implicit branch since the GS and IS bits are used to compute the virtual address for instruction translation and CM affects the fetch address. Instructions may be fetched and executed from any context from any permutation of these bits. Software should guarantee that a translation exists for each of the permutations of these bits and that translation has the same characteristics including appropriate permissions and RPN fields. For this reason it is unwise and difficult to use **mtmsr** to change these bits and such changes should only be done through return from interrupt type instructions, which provide the context synchronization atomically with instruction execution.

3.6.1.4 Changing MSR Content in Guest Supervisor State

<Embedded.Hypervisor>:
Certain bits in the MSR cannot be changed when the processor is in the guest supervisor state (MSR[GS]=1, MSR[PR]=0). MSR[GS] cannot be changed from 1 to 0 unless it is changed as a result of taking an interrupt that is directed to the hypervisor state. MSR[WE] cannot be changed when MSR[GS]=1, MSR[PR]=0 unless it is changed as a result of taking an interrupt that is directed to the hypervisor state. Other bits in the MSR may be protected from changing in guest supervisor state when configured by the hypervisor. MSR[UCLE], MSR[DE], and MSR[PMM] can only be changed when MSR[GS]=1, MSR[PR]=0 if MSRP[UCLEP], MSRP[DEP], and MSRP[PMMP] respectively are set to 0.

Guest supervisor[1]

	32	33 34	35	36	37	38	39 44	45	46	47	48	49	50	51	52	53	54	55	56 57	58	59	60	61	62	63
Field R/W	CM	—	GS	—	UCLE	SPV	—	WE	CE	—	EE	PR	FP	ME	FE0	—	DE	FE1	—	IS	DS	—	PMM	RI	
Reset	All zeros																								

Figure 3-11. Machine State Register (MSR)

[1] Note: GS and WE cannot be changed when executing in guest supervisor state. UCLE, DE, and PMM are writable in guest supervisor state only if allowed by MSRP register settings. <E.HV> See below.

Table 3-19 describes MSR fields.

Table 3-19. MSR Field Descriptions

Bits	Name	Description
32	CM	Computation mode. <64-bit> 0 The processor runs in 32-bit mode. 1 The processor runs in 64-bit mode.
33–34	—	Reserved, should be cleared.[1]
35	GS	Guest state. <Embedded.Hypervisor> Indicates whether the core is running in guest state under the control of a hypervisor program. 0 The processor is not running in the guest state. 1 The processor is running in the guest state. MSR[GS,PR] are used to define privilege levels. MSR[GS] is part of the virtual address and is used to form distinct address spaces for hypervisor and guest software. MSR[GS] cannot be changed from 1 to 0, except when taking an interrupt that is directed to the hypervisor state.
36	—	Reserved, should be cleared.[1]
37	UCLE	User-mode cache lock enable. <Embedded.Cache Locking> 0 Any cache lock instruction executed in user-mode takes a cache-locking DSI exception and sets either ESR[DLK] or ESR[ILK]. This allows the operating system to manage and track the locking/unlocking of cache lines by user-mode tasks. 1 Cache-locking instructions can be executed in user-mode and they do not take a DSI for cache-locking. (They may still take a DSI for access violations though.) MSR[UCLE] cannot be modified when MSR[GS]=1 unless MSRP[UCLEP] is 0. <Embedded.Hypervisor>
38	SPV	SP/embedded floating-point/vector available. Enables use of 64-bit extended GPRs for SPE, single-precision vector, and double-precision floating-point instructions and enables use of vector registers for AltiVec. <SPE>: 0 The processor cannot execute any category SPE, SP.FD, or SP.FV instructions except for the **brinc** instruction. 1 The processor can execute all category SPE, SP.FD, or SP.FV instructions. **Note:** Embedded floating-point instructions require SPV to be set. An attempt to execute an embedded floating-point instruction when MSR[SPV] is 0 results in an SPE or embedded floating-point unavailable interrupt. <Vector>: 0 The processor cannot execute any category Vector instruction. 1 The processor can execute category Vector instructions.
39–44	—	Reserved, should be cleared. [1]
45	WE	Wait state enable. Allows the core to signal a request for power management, according to the states of HID0[DOZE,NAP,SLEEP]. 0 The processor is not in wait state and continues processing. No power management request is signaled to external logic. 1 The processor enters wait state by ceasing to execute instructions and entering low-power mode. Details of how wait state is entered and exited and how the processor behaves in the wait state are implementation-dependent. **Note:** MSR[WE] is being phased out of EIS. Newer processors do not implement MSR[WE] and power management states are controlled by writing registers in the integrated device

Table 3-19. MSR Field Descriptions (continued)

Bits	Name	Description
46	CE	Critical enable 0 Critical input, processor doorbell critical <E.PC>, guest processor doorbell critical <E.HV>, and watchdog timer interrupts are disabled 1 Critical input, processor doorbell critical <E.PC>, guest processor doorbell critical <E.HV>, and watchdog timer interrupts are enabled <Embedded,Hypervisor>: Interrupts that are maskable by MSR[CE], except for guest processor doorbell critical ,are enabled regardless of the state of MSR[CE] when MSR[GS]=1. Guest processor doorbell critical interrupts are only enabled when MSR[CE] = 1 and MSR[GS]=1.
47	—	Reserved, should be cleared.
48	EE	External enable 0 External input, decrementer, fixed-interval timer, processor doorbell <E.PC>, guest processor doorbell <E.HV>, and embedded performance monitor <E.PM> interrupts are disabled 1 External input, decrementer, fixed-interval timer, processor doorbell <E.PC>, guest processor doorbell <E.HV>, and embedded performance monitor <E.PM> interrupts are enabled <Embedded,Hypervisor>: Interrupts that are maskable by MSR[EE] are enabled differently based on whether the interrupt is directed to the hypervisor state or the guest supervisor state. If the interrupt is directed to the guest supervisor state (except for guest processor doorbell), the interrupt is enabled if MSR[EE]=1 or MSR[GS]=1. If the interrupt is directed to the guest supervisor state (or the interrupt is guest processor doorbell) , the interrupt is enabled if MSR[EE]=1 and MSR[GS]=1.
49	PR	User mode (problem state) 0 The processor is in supervisor mode. 1 The processor is in user mode. MSR[PR] also affects memory access control. <Embedded.Hypervisor>: Access to hypervisor privileged instructions and resources is allowed only when MSR[GS]=0 and MSR[PR]=0.
50	FP	Floating-point available. 0 The processor cannot execute floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions.
51	ME	Machine check enable. 0 Machine check interrupts are disabled. 1 Machine check interrupts are enabled. <Embedded.Hypervisor>: Interrupts that are maskable by MSR[ME], except for guest processor doorbell machine check, are enabled regardless of the state of MSR[ME] when MSR[GS]=1. Guest processor doorbell machine check interrupts are only enabled when MSR[ME] = 1 and MSR[GS]=1.
52	FE0	Floating-point exception mode 0. FE0 and FE1 are interpreted together, as follows: FE0 FE1 Mode 0 0 Ignore exceptions 0 1 Imprecise nonrecoverable 1 0 Imprecise recoverable 1 1 Precise **Note:** Freescale processors only implement precise exceptions. Setting FE0 and FE1 to any of the imprecise modes will still result in precise recoverable exceptions.

Table 3-19. MSR Field Descriptions (continued)

Bits	Name	Description
53	—	Reserved, should be cleared.
54	DE	Debug interrupt enable 0 Debug interrupts are disabled. 1 Debug interrupts are enabled if DBCR0[IDM] = 1. See the description of the DBSR[UDE] in Section 3.13.4, “Debug Status Register (DBSR/DBSRWR).” MSR[DE] cannot be modified when MSR[GS]=1 unless MSRP[DEP] is 0. <Embedded.Hypervisor>
55	FE1	Floating-point exception mode 1. FE0 and FE1 are interpreted together, as described in the FE0 description.
56	—	Reserved, should be cleared.
57	—	Reserved, should be cleared.
58	IS	Instruction address space 0 The processor directs all instruction fetches to address space 0 (TS = 0 in the relevant TLB entry). 1 The processor directs all instruction fetches to address space 1 (TS = 1 in the relevant TLB entry).
59	DS	Data address space 0 The processor directs data memory accesses to address space 0 (TS = 0 in the relevant TLB entry). 1 The processor directs data memory accesses to address space 1 (TS = 1 in the relevant TLB entry).
60	—	Reserved, should be cleared.
61	PMM	<Embedded.Performance Monitor> Performance monitor mark. System software can set PMM when a marked process is running to enable statistics gathering only during the execution of the marked process. PMM and MSR[PR] together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified in the PMLCax, the state for which monitoring is enabled, counting is enabled. See Section 10.4, “Performance Monitor Use Case,” for additional information. MSR[PMM] cannot be modified when MSR[GS]=1 unless MSRP[PMMP] is 0. <Embedded.Hypervisor>
62	RI	Recoverable Interrupt. When an interrupt occurs, this bit is unchanged by the interrupt mechanism when a new MSR is established; however, when a machine check, error report or NMI interrupt occurs, RI is cleared. **Note:** If used properly, RI determines whether an interrupt that is taken at the machine check interrupt vector can be safely returned from (for example, that architected state set by the interrupt mechanism has been safely stored by software). RI should be set by software when all MSR values are first established. When an interrupt occurs that is taken at the machine check interrupt vector, software should set RI when it has safely stored MCSRR0 and MCSRR1. The associated MCSRR1 bit should be checked to determine whether the interrupt occurred when another machine check interrupt was being processed and before state was successfully saved. If RI is set in MCSRR1, it is safe to return when processing is complete.
63	—	Reserved, should be cleared.

[1] An MSR bit that is reserved may be altered by return from interrupt instructions.

3.6.2 Machine State Register Protect Register (MSRP) <E.HV>

MSRP, shown in the following figure, controls whether certain bits in the MSR can be modified in guest supervisor state (MSR[PR] = 0 and MSR[GS] = 1). In addition, the MSRP affects the behavior of cache locking and performance monitor instructions in guest state.

3.6.2.1 Using the MSRP

The MSRP is used to prevent a guest operating system from modifying the UCLE, DE, or PMM bits in the MSR. The MSRP bits UCLEP, DEP, and PMMP control whether the guest can change the corresponding MSR bits UCLE, DE, and PMM respectively. When the MSRP bit associated with a corresponding MSR bit is 0, any operation which changes the MSR in guest supervisor state is allow to modify that MSR bit, whether from an instruction that modifies the MSR, or from an interrupt which is taken in the guest supervisor state. When the MSRP bit associated with a corresponding MSR bit is 1, no operation which changes the MSR in guest supervisor state is allow to modify that MSR bit (that is, it remains unchanged), whether from an instruction that modifies the MSR, or from an interrupt which is taken in the guest supervisor state.

A change to MSRP requires a context synchronizing operation to be performed before the effects of the change are guaranteed to be visible in the current context.

This register is hypervisor privileged.

3.6.2.2 Behavior of Cache Locking Instructions in Guest Supervisor State

<Embedded.Cache Locking>:
The behavior of cache locking instructions (**dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, and **icblc**) in guest supervisor state is dependent on the setting of MSRP[UCLEP]. When MSRP[UCLEP] = 0, cache locking instructions are permitted to execute normally in the guest supervisor state. When MSRP[UCLEP] = 1, cache locking instructions are not permitted to execute in the guest supervisor state and cause an embedded hypervisor privilege exception when executed. Execution of cache locking instructions in user state is not affected by the setting of MSRP[UCLEP].

3.6.2.3 Behavior of Performance Monitor Instructions in Guest Supervisor State

<Embedded.Performance Monitor>:
The behavior of performance monitor instructions (**mtpmr** and **mfpmr**) in guest state is dependent on the setting of MSRP[PMMP]. When MSRP[PMMP] = 0, performance monitor instructions are permitted to execute normally in the guest state. When MSRP[PMMP] = 1, performance monitor instructions are not permitted to execute in the guest state. Execution of a **mfpmr** instruction which specifies a user performance monitor register (PMR) produces a value of 0 in the destination GPR. In the guest supervisor state, execution of any **mfpmr** or **mtpmr** instruction which specifies a privileged PMR causes an embedded hypervisor privilege exception.

SPR 311 Hypervisor

	32–36	37	38–53	54	55–60	61	62–63
R W	—	UCLEP	—	DEP	—	PMMP	—
Reset	All zeros						

Figure 3-12. Machine State Register Protect (MSRP) Format

This table describes the MSRP fields.

Table 3-20. MSRP Field Descriptions

Bits	Name	Description
32–36	—	Reserved, should be cleared.
37	UCLEP	User cache lock enable protect 0 Changes to MSR[UCLE] are allowed if MSR[GS] = 1. 1 Changes to MSR[UCLE] are ignored if MSR]GS] = 1. If MSR[GS,PR] = 0b10 and MSRP[UCLEP] = 1, cache-locking instructions cause an embedded hypervisor privilege exception.
38–53	—	Reserved, should be cleared.
54	DEP	Debug enable protect 0 Changes to MSR[DE] are allowed if MSR[GS] = 1. 1 Changes to MSR[DE] are ignored if MSR[GS] = 1.
55–60	—	Reserved, should be cleared.
61	PMMP	Performance monitor mark protect 0 Changes to MSR[PMM] are allowed if MSR[GS] = 1. 1 Changes to MSR[PMM] are ignored if MSR[GS] = 1. **mfpmr** instructions for user PMRs return 0. mfpmr and mtpmr instructions executed when MSR[GS] = 1 which specify a privileged PMR cause an embedded hypervisor privilege exception.
62–63	—	Reserved, should be cleared.

NOTE: Software Considerations

> The state of the MSRP at reset allows guest supervisor state access to MSR[UCLE,DE,PMM] and the associated cache locking and performance monitor facilities.

3.6.3 Embedded Processor Control Register (EPCR)

EPCR, shown in the following figure, controls whether certain interrupts are directed to the hypervisor state or guest supervisor state and whether debug events are suppressed when in hypervisor state.

Changing the EPCR register with **mtspr** requires a context synchronizing operation to be performed before the effects of the change are guaranteed to be visible in the current context.

This register is implemented if category Embedded.Hypervisor or category 64-bit is supported.

This register is hypervisor privileged.

.

SPR 307 Hypervisor

	32	33	34	35	36	37	38	39	40	41	42 63
R W	EXTGS	DTLBGS	ITLBGS	DSIGS	ISIGS	DUVD	ICM	GICM	DGTMI	DMIUH	–
Reset	All zeros										

Figure 3-13. Embedded Processor Control Register (EPCR)

This table describes EPCR fields.

Table 3-21. EPCR Field Descriptions

Bits	Name	Description
32	EXTGS	External input interrupt directed to guest state. <E.HV> 0 External inputs interrupts are directed to the hypervisor state. External input interrupts remain pending until MSR[GS] = 1 or MSR[EE] = 1. 1 External inputs interrupts are directed to the guest state. External input interrupts remain pending until MSR[GS] = 1 and MSR[EE] = 1.
33	DTLBGS	Data TLB error interrupt directed to guest state. <E.HV> 0 Data TLB error interrupts are directed to the hypervisor state when MSR[GS] = 1. 1 Data TLB error interrupts are directed to the guest state when MSR[GS] = 1. When MSR[GS] = 0, interrupts are always directed to the hypervisor state.
34	ITLBGS	Instruction TLB error interrupt directed to guest state. <E.HV> 0 Instruction TLB error interrupts are directed to the hypervisor state when MSR[GS] = 1. 1 Instruction TLB error interrupts are directed to the guest state when MSR[GS] = 1. When MSR[GS] = 0, interrupts are always directed to the hypervisor state.
35	DSIGS	Data storage interrupt directed to guest state. <E.HV> 0 Data storage interrupts are directed to the hypervisor state when MSR[GS] = 1. 1 Data storage interrupts are directed to the guest state when MSR[GS] = 1. When MSR[GS] = 0, interrupts are always directed to the hypervisor state.
36	ISIGS	Instruction storage interrupt directed to guest state. <E.HV> 0 Instruction storage interrupts are directed to the hypervisor state when MSR[GS] = 1. 1 Instruction storage interrupts are directed to the guest state when MSR[GS] = 1. When MSR[GS] = 0, interrupts are always directed to the hypervisor state.
37	DUVD	Disable hypervisor debug. <E.HV> 0 Debug events can occur when MSR[GS] = 0. 1 Debug events, except for the unconditional debug event, are suppressed (do not occur) when MSR[GS] = 0. It is implementation dependent whether the unconditional debug event is suppressed.
38	ICM	Interrupt computation Mode. <64-bit> Indicates the computation mode when an interrupt occurs. If category E.HV is implemented, ICM controls the computation mode for interrupts that are directed to the hypervisor state. If category E.HV is not implemented, this bit controls the computation mode for all interrupts. At interrupt time, EPCR[ICM] is copied into MSR[CM]. 0 Interrupts will execute in 32-bit mode. 1 Interrupts will execute in 64-bit mode.
39	GICM	Guest interrupt computation mode.<E.HV><64-bit> Indicates the computation mode when an interrupt occurs that is directed to the guest supervisor state. At interrupt time, EPCR[GICM] is copied into MSR[CM] if the interrupt is directed to guest supervisor state. 0 Interrupts will execute in 32-bit mode. 1 Interrupts will execute in 64-bit mode.
40	DGTMI	Disable guest TLB management instructions.<E.HV> Controls whether guest supervisor state can execute any TLB management instructions. 0 **tlbilx** is allowed to execute normally when MSR[GS,PR] = 10. 1 Any TLB management instruction (**tlbilx**, **tlbwe**, **tlbre**, **tlbsx**, **tlbivax**, **tlbsync**) causes an embedded hypervisor privilege exception when MSR[GS,PR] = 10. Note: *Only the TLB management instruction **tlbilx** is allowed to execute normally from guest supervisor state. The other TLB management instructions always require hypervisor privilege to execute.*

Table 3-21. EPCR Field Descriptions (continued)

Bits	Name	Description
41	DMIUH	Disable MAS interrupt updates for hypervisor. <E.HV> Controls whether MAS registers are updated by hardware when a Data or Instruction TLB error interrupt or a data or instruction storage interrupt is taken in the hypervisor. 0 MAS registers are set by hardware as described in Section 6.5.8.3, “MAS Register Updates for Exceptions, tlbsx, and tlbre” when an ITLB, DTLB, ISI, or DSI interrupt is taken in the hypervisor. 1 MAS registers are left unchanged by hardware when an ITLB, DTLB, ISI, or DSI interrupt is taken in the hypervisor.
42–63	—	Reserved.

3.6.4 Hardware Implementation-Dependent Registers

Hardware implementation-dependent registers, HID0 and HID1, contain fields defined by the architecture or by the implementation. This section describes only architecture-defined fields.

NOTE: Software Considerations

Always consult the core register descriptions in the integrated device reference manual. An integrated device may not use all HID fields and may define some fields more specifically.

3.6.4.1 Hardware Implementation-Dependent Register 0 (HID0)

HID0, shown in the following figure, is used for configuration and control. Although the HID0 bits defined here are required for certain functionality, the actual bits implemented varies from device to device. Also processors typically implement additional HID0 fields not defined by the architecture. In some cases, a processor may implement an architected field more specifically than it is defined here; in turn, the field may have even more specific behavior defined for the specific embedded device. Consult the processor and SoC documentation.

Unless otherwise noted in processor documentation, writing to HID0 typically requires synchronization before the effects are guaranteed to be seen in the current context.

This register is hypervisor privileged.

SPR 1008 Hypervisor

	32	33 37	38 39	40 42	43	44	45	46	47
R W	EMCP	—	BPRED	PWRMGMT	DPM	EDPM	—	IPR	—
Reset	All zeros								

	48	49	50	51 54	55	56	57	58	59	60	61	62	63
R W	EIEC	TBEN	SEL_TBCLK	—	DAPUEN	EN_MAS7_ UPDATE	DCFA	EIEIO_EN	CIGLSO	—	NOPTST	—	NOPTI
Reset	All zeros												

Figure 3-14. Hardware Implementation-Dependent Register 0 (HID0)

This table describes the HID0 fields.

Table 3-22. HID0 Field Descriptions

Bits	Name	Description
32	EMCP	Enable machine check pin. Used to mask machine check exceptions delivered to the core from the machine check input signal. 0 Machine check exceptions from the machine check signal are disabled. 1 Machine check exceptions from the machine check signal are enabled. Asserting the machine check input signal will cause MCSR[MCP] to be set which will lead to a asynchronous machine check interrupt when asynchronous machine check interrupts are enabled. On some older cores, if MSR[ME] = 0, asserting the machine check input signal causes a checkstop. If MSR[ME] = 1, asserting the machine check input signal causes a machine check exception.
33–37	—	Implementation dependent.
38–39	BPRED	Branch prediction control (e200 cores). Controls instruction buffer look-ahead for branch acceleration. Note that for branches with AA = 1, the msb of the displacement field is still used to indicate forward/backward, even though the branch is absolute. 00 Branch acceleration is enabled. 01 Branch acceleration is disabled for backward branches. 10 Branch acceleration is disabled for forward branches. 11 Branch acceleration is disabled for forward and backward branches.
40–42	PWRMGMT	Power management control. The semantics of PWRMGMT are implementation dependent. Previous processors have used bit 40 for DOZE, bit 41 for NAP, and bit 42 for SLEEP although the actual semantics vary from processor to processor. Newer processors use SoC based controls.
43	DPM	Dynamic power management. Used to enable power-saving by shutting off functional resources not in use. Setting or clearing DPM should not affect performance. 0 Dynamic power management is disabled. 1 Dynamic power management is enabled.
44	EDPM	Enhanced dynamic power management. Used to enable additional power-saving by shutting off functional resources not in use. Setting EDPM may have adverse effects on performance. 0 Enhanced dynamic power management is disabled. 1 Enhanced dynamic power management is enabled.
45	—	Implementation dependent.

Table 3-22. HID0 Field Descriptions (continued)

Bits	Name	Description
46	ICR	Interrupt inputs clear reservation. Controls whether external input and critical input interrupts cause an established reservation to be cleared. 0 External and critical input interrupts do not affect reservation status. 1 External and critical input interrupts, when taken, clear an established reservation.
47	—	Implementation dependent.
48	EIEC	Enable internal error checking. Used to control whether internal processor errors cause a machine check exception. 0 Internal error reporting is disabled. Internally detected processor errors do not generate a machine check interrupt. 1 Internal error reporting is enabled. Internally detected processor errors generate a machine check interrupt.
49	TBEN	Time base enable. Used to control whether the time base increments. 0 The time base is not enabled and does not increment. 1 The time base is enabled and increments at a rate determined by HID0[SEL_TBCLK].
50	SEL_TBCLK	Select time base clock. Used to select the source of the time base clock. 0 The time base is updated based on a core implementation-specific rate. 1 The time base is updated based on an external signal to the core.
51–54	—	Implementation dependent.
55	DAPUEN	Enhanced debug enable. Controls whether the enhanced debug category is enabled. 0 Enhanced debug is disabled. Debug interrupts use CSRR0 and CSRR1 to save state and the **rfci** instruction to return from the debug interrupt. 1 Enhanced debug is enabled; debug interrupts use DSRR0 and DSRR1 to save state and the **rfdi** instruction to return from the debug interrupt.
56	EN_MAS7 _UPDATE	Enable hot-wire update of MAS7 register. Implementations that support this bit do not update MAS7 (upper bits of RPN (RPNU) field) when hardware writes MAS registers via a **tlbre**, **tlbsx**, or an interrupt unless this bit is set. This provides a compatibility path for processors that originally offered only 32 bits of physical addressing but have since extended past 32 bits. 0 Hardware updates of MAS7 are disabled. 1 Hardware updates of MAS7 are enabled.
57	DCFA	Data cache flush assist. 0 The primary data cache uses its normal line replacement policy. 1 The primary data cache uses a line replacement policy that is deterministic regardless of the state of lines in the cache.
58	EIEIO_EN	Eieio synchronization enable. Allows **mbar** instructions to provide the same synchronization semantics as the **eieio** instruction from PowerPC 1.xx architectures. 0 The synchronization provided by the **mbar** instruction is performed in the PowerISA manner. Additional forms of synchronization, if implemented, are determined by the value in the MO field. 1 The synchronization provided by the **mbar** instruction is equivalent to PowerPC 1.xx **eieio** synchronization. The MO field is ignored.
59	CIGLSO	Cache Inhibited Guarded Load Store Ordering 0 Loads and stores to storage that are marked as Caching Inhibited and Guarded (WIMGE = 0b01x1x) have no ordering implied except what is defined in the rest of the architecture. 1 Loads and stores to storage that are marked as Caching Inhibited and Guarded (WIMGE = 0b01x1x) are performed in order with respect to other Caching Inhibited and Guarded loads and stores.
60	—	Implementation dependent.

Table 3-22. HID0 Field Descriptions (continued)

Bits	Name	Description
61	NOPTST	No-op cache touch for store instructions. Controls whether data cache touch for store instructions perform no operation. Cache locking instructions are not affected by this control. 0 **dcbtst**, **dstst**, and **dststt** and other forms of cache touch for store instructions operate as defined by the architecture unless disabled by NOPDST or NOPTI. 1 **dcbtst**, **dstst**, and **dststt** and other forms of cache touch for store instructions are treated as no-ops for execution and debug events.
62	—	Implementation dependent.
63	NOPTI	No-op cache touch instructions. Data and instruction cache touch instructions perform no operations. Cache locking instructions are not affected by this control. 0 **dcbt**, **dcbtst**, **icbt** and other forms of cache touch instructions operate as defined by the architecture unless disabled by NOPDST or NOPTST. 1 **dcbt**, **dcbtst**, **icbt** and other cache touch instruction forms are treated as no-ops.

3.6.4.2 Hardware Implementation-Dependent Register 1 (HID1)

Although HID1, shown in the following figure, is defined as SPR 1009 and intended for bus configuration and control, its fields are implementation specific, and some cores do not implement the HID1 at all. Refer to the processor and SoC documentation.

Unless otherwise noted in processor documentation, writing to HID0 typically requires synchronization before the effects are guaranteed to be seen in the current context.

This register is hypervisor privileged.

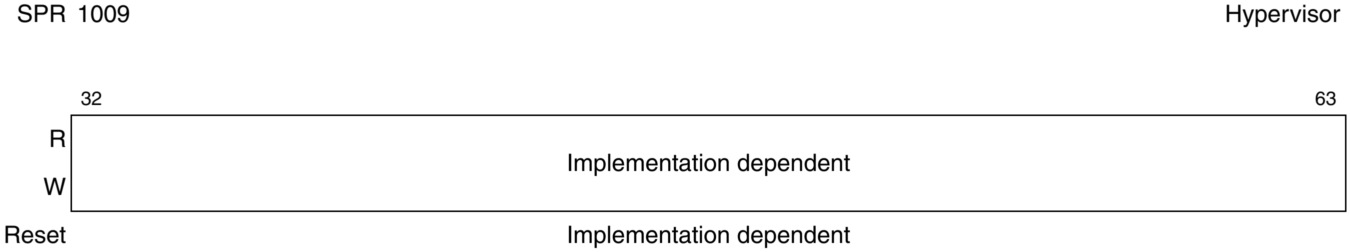


Figure 3-15. Hardware Implementation-Dependent Register 1 (HID1)

3.6.5 Processor ID Register (PIR)

The value in PIR, shown in the following figure, is used to distinguish processors in a system from one another.

If category Embedded.Processor Control is implemented, the PIR is writable so software can store information specific to its needs. Otherwise, whether the PIR is writable is implementation specific.

In a multiprocessor system, each PIR register should be initialized at power-on-reset by hardware to a unique value. The contents of the PIR register are used as a tag for matching requested doorbell interrupts sent to the processor. Note that SPR 286 was assigned to the system version register (SVR) on some earlier devices.

<Embedded.Hypervisor>:
If the processor is in the guest supervisor state, reading the PIR with **mfspr** is mapped to the Guest Processor ID Register (GPIR). See Section 3.2.2.1, "SPR Register Mapping <E.HV>."

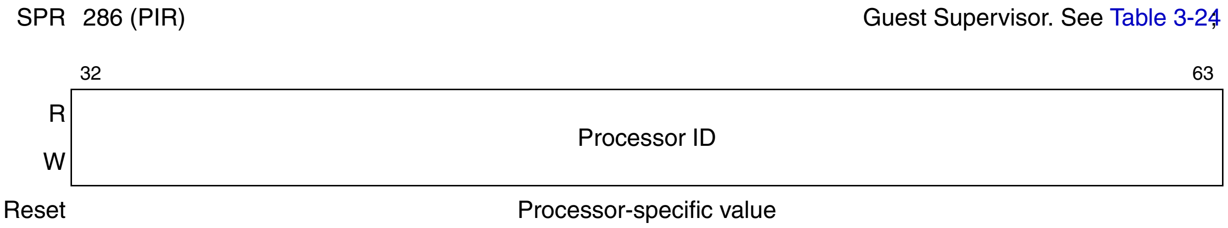


Figure 3-16. Processor ID Register (PIR)

<Embedded.Processor Control>:
The processor sets the initial value of PIR at reset, after which it is writable by software. The initial value of the PIR is a processor unique value within the coherence domain.

This table describes the PIR reset value.

Table 3-23. PIR Reset Value <E.PC>

Bits	Name	Reset Value/Description
32–49	—	Reserved, should be cleared.
50–51	CHIP_ID	Represents the unique chip number in a given system from 0–3. Configured by an external customer-driven resource such as signals and consists of two signals to the processor core.
52–53	CLUSTER_ID	Represents the unique SoC cluster number in a given SoC from 0–3. The CLUSTER_ID comes from the SoC platform and consists of two signal pins to the processor core.
54–58	CORE_CLUSTER_ID	Represents the unique core cluster number in a given SoC cluster from 0–31. The CLUSTER_CORE_ID comes from the SoC platform and consists of 5 signal pins to the processor core.
59–60	CORE_ID	Represents the unique core number in a given core cluster from 0–3. The CORE_ID comes from the core cluster and consists of 2 signal pins to the processor core.
61–63	—	Reserved, should be cleared.

<Embedded.Hypervisor>:
Writes to PIR in guest supervisor state cause an embedded hypervisor privilege exception, as shown in the following table.

NOTE: Software Considerations

Guest supervisor state is not allowed to write PIR (GPIR) so that the hypervisor can easily track the changes a guest OS makes to GPIR. The hypervisor should perform the write on behalf of the guest.

Table 3-24. PIR Access Privilege <E.HV>

Instruction	MSR[GS]	MSR[PR]	Result
mtspr	0	0	Execute
	0	1	Privilege (PPR) exception
	1	0	Embedded hypervisor privilege
	1	1	Privilege (PPR) exception
mfspr	0	0	Execute
	0	1	Privilege (PPR) exception
	1	0	Execute **mfspr GPIR** (mapped to GPIR)
	1	1	Privilege (PPR) exception

3.6.6 Guest Processor ID Register (GPIR) <E.HV>

The value in GPIR, shown in the following figure, is used to distinguish processors in a logical partition from one another.

The GPIR is writable so software in a logical partition can store information specific to its needs.

In a multiprocessor system, each GPIR register in a logical partition should be initialized at partition creation time by the hypervisor to a unique value in the logical partition. The contents of the GPIR register are used as a tag for matching requested guest processor doorbell interrupts sent to the processor.

If the processor is in the guest supervisor state, reading PIR with **mfspr** is mapped to the Guest Processor ID Register (GPIR). See Section 3.2.2.1, “SPR Register Mapping <E.HV>.”

The GPIR is not writable by guest supervisor software so that the hypervisor can track changes to it.

SPR 382 Guest supervisor. See Table 3-25

	32	63
R	Guest processor ID	
W		
Reset	All zeros	

Figure 3-17. Guest Processor ID Register (GPIR)

This table summarizes GPIR access permissions.

Table 3-25. GPIR Access Privilege

Instruction	MSR[GS]	MSR[PR]	Result
mtspr	0	0	Execute
	0	1	Privilege (PPR) exception
	1	0	Embedded hypervisor privilege
	1	1	Privilege (PPR) exception
mfspr	0	0	Execute
	0	1	Privilege (PPR) exception
	1	0	Execute
	1	1	Privilege (PPR) exception

NOTE: Software Considerations

Operating system software should not directly access GPIR, but should instead perform **mfspr** and **mtspr** to PIR. This makes the programming model the same whether the operating system is running as a guest under a hypervisor or is running bare-metal.

3.6.7 Processor Version Register (PVR)

The PVR, shown in the following figure, contains a value identifying the version and revision level of the processor. The PVR distinguishes between processor cores that differ in attributes that may affect software.

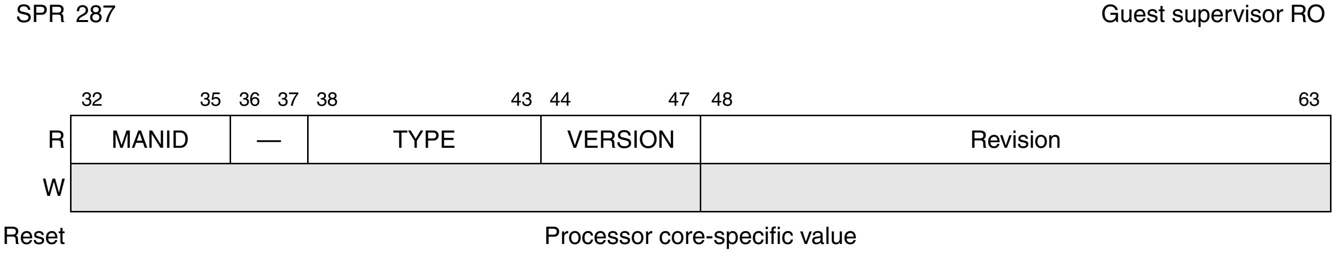


Figure 3-18. Processor Version Register (PVR)

This table describes PVR fields.

Table 3-26. PVR Field Descriptions

Bits	Name	Description
32–35	MANID	Manufacturer’s ID. Freescale = 0x8. IBM was initially assigned a value of 0x0, but has used other values. Using other values for this field may be possible in the future with agreement from IBM.
36–37	—	Reserved, should read as 0.

Table 3-26. PVR Field Descriptions (continued)

Bits	Name	Description
38–43	TYPE	Core ID, type of core. Specifies a different type of core. Usually associated with major micro architectural changes or major programming model changes.
44–47	VERSION	Core ID, version of TYPE. Specifies different versions of the same type of core.
48–63	Revision	Distinguishes implementations of the version. Different revision numbers indicate minor differences between processors having the same version number, but should reflect the same programming model as all values of Revision for MANID, PART and TYPE.

The PVR values allow software to differentiate between multiple processor core types. The EIS uses field descriptions and values consistent with past PowerPC processors.

In an integrated device the PVR applies to the specific core, not the IC itself, which is identified by the device's system version register (SVR).

3.6.8 System Version Register (SVR)

The SVR, shown in the following figure, contains a read-only SoC-dependent value. The integrated device may also implement a memory-mapped version of the SVR; consult the documentation for the implementation.

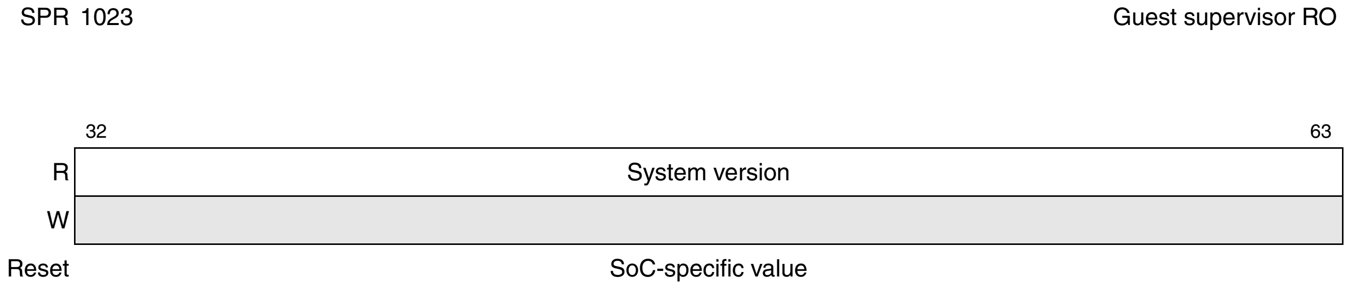


Figure 3-19. System Version Register (SVR)

3.7 Timer Registers

The time base (TB), decrementer (DEC), fixed-interval timer (FIT), and watchdog timer provide timing functions for the system. The relationship of these timer facilities to each other is shown in Figure 8-2 and is described in Chapter 8, "Timer Facilities."

- The TB is a long-period counter driven at an implementation-dependent frequency.
- The decrementer, updated at the same rate as the TB, provides a way to signal an exception after a specified period unless one of the following occurs:
  - — DEC is altered by software in the interim.
  - — The TB update frequency changes.

The DEC is typically used as a general-purpose software timer.

- The time base for the TB and DEC is implementation dependent, but is generally controlled by the time base enable (TBEN) and for some processors, the select time base clock (SEL_TBCLK) HID0 fields.
- Software can select one TB bit to signal a fixed-interval interrupt whenever the bit transitions from 0 to 1. It is typically used to trigger periodic system maintenance functions.
- The watchdog timer, also a selected TB bit, provides a way to signal a critical exception when the selected bit transitions from 0 to 1. It is typically used for system error recovery. If software does not respond in time to the initial interrupt by clearing the associated status bits in the TSR before the next expiration of the watchdog timer interval, a watchdog timer-generated processor reset may result, if so enabled.

All timer facilities must be initialized during start-up.

3.7.1 Timer Control Register (TCR)

TCR, shown in the following figure, provides control for on-chip timer resources. See Section 7.8.10, “Decrementer Interrupt,” Section 7.8.11, “Fixed-Interval Timer Interrupt,” and Section 7.8.12, “Watchdog Timer Interrupt.”

This register is hypervisor privileged.

SPR 340 Hypervisor

	32 33	34 35	36	37	38 39	40	41	42	43 46	47 50	51 63
R W	WP	WRC	WIE	DIE	FP	FIE	ARE	—	WPEXT	FPEXT	—
Reset	Processor specific value										

Figure 3-20. Timer Control Register (TCR)

This table describes the TCR fields.

Table 3-27. TCR Field Descriptions

Bits	Name	Description
32–33	WP	Watchdog timer period. When concatenated with WPEXT, specifies one of 64-bit locations of the time base used to signal a watchdog timer exception on a transition from 0 to 1. WPEXT || WP = 0000_00 selects TB[0] (the msb of the TB) WPEXT || WP = 1111_11 selects TB[63] (the lsb of the TB) If an implementation does not implement WPEXT, WP selects the bit.
34–35	WRC	Watchdog timer reset control. Cleared by processor reset and set only by software. To prevent errant code from disabling the watchdog reset function, once software sets the WRC field, it remains set until a reset occurs. 00 No watchdog timer reset can occur. TCR[WRC] resets to 00. WRC may be set by software, but cannot be cleared by software (except by a software-induced reset) 01–11 Force processor to be reset on second time-out of watchdog timer. The exact function of any of these settings is implementation-dependent. Earlier processors may implement this field somewhat differently: consult the core documentation to determine the exact behavior. Core-level behavior for WRC is provided in the core reference manual; SoC-specific information is provided in the documentation for the integrated device.
36	WIE	Watchdog timer interrupt enable 0 Watchdog timer interrupts disabled 1 Watchdog timer interrupts enabled
37	DIE	Decrementer interrupt enable 0 Decrementer interrupts disabled 1 Decrementer interrupts enabled
38–39	FP	Fixed-interval timer period. When concatenated with FPEXT, FP specifies one of 64 bit locations of the time base used to signal a fixed-interval timer exception on a transition from 0 to 1. FPEXT || FP = 0000_00 selects TB[0] (the msb of the TB) FPEXT || FP = 1111_11 selects TB[63] (the lsb of the TB) If an implementation does not implement FPEXT, FP selects the bit.
40	FIE	Fixed interval interrupt enable 0 Fixed interval interrupts disabled 1 Fixed interval interrupts enabled
41	ARE	Auto-reload enable. Controls whether the value in DECAR is reloaded into the DEC when the DEC value is decremented from a value of 0000_0001h. 0 Auto-reload disabled. A decrementer exception is presented (that is TSR[DIS] is set) when the decrementer is decremented from a value of 0x0000_0001. The next value placed in the decrementer is the value 0x0000_0000. The decrementer then stops decrementing. If MSR[EE]=1 (or MSR[GS]=1<E.HV>), TCR[DIE]=1, and TSR[DIS]=1, a decrementer interrupt is taken. Software must reset TSR[DIS]. 1 Auto-reload enabled. A decrementer exception is presented (TSR[DIS] is set) when the decrementer reaches a value of 0x0000_0001. The contents of the DECAR is placed in the decrementer. The decrementer resumes decrementing. If MSR[EE]=1 (or MSR[GS]=1<E.HV>), TCR[DIE]=1, and TSR[DIS]=1, a decrementer interrupt is taken. Software must reset TSR[DIS].
42	—	Reserved, should be cleared.
43–46	WPEXT	Watchdog timer period extension (see the description for WP). If an implementation does not implement WPEXT, WP selects the bit.

Table 3-27. TCR Field Descriptions (continued)

Bits	Name	Description
47–50	FPEXT	Fixed-interval timer period extension (see the description for FP). If an implementation does not implement FPEXT, FP selects the bit.
51–63	—	Reserved, should be cleared.

3.7.2 Timer Status Register (TSR)

The TSR, shown in the following figure, contains status on timer events and the most recent watchdog timer-initiated processor reset. All TSR bits are write-1-to-clear.

This register is hypervisor privileged.

NOTE

> Register fields designated as write-1-to-clear are cleared only by writing ones to them. Writing zeros to them has no effect.

SPR 336 Hypervisor R/Clear

	32	33	34 35	36	37	38 … 63
R	ENW	WIS	WRS	DIS	FIS	—
W	w1c	w1c	w1c	w1c	w1c	
Reset	All zeros					

Figure 3-21. Timer Status Register (TSR)

This table describes TSR fields.

Table 3-28. TSR Field Descriptions

Bits	Name	Description
32	ENW	Enable next watchdog time. If a watchdog time-out occurs while WIS = 0 and the next watchdog time-out is enabled (ENW = 1), a watchdog timer exception is generated and logged by setting WIS. See Section 7.8.12, “Watchdog Timer Interrupt,” for details. 0 Action on next watchdog timer time-out is to set TSR[ENW]. 1 Action on next watchdog timer time-out is governed by TSR[WIS].
33	WIS	Watchdog timer interrupt status. See the ENW description for more information about how WIS is used. 0 A watchdog timer event has not occurred. 1 A watchdog timer event occurred. If (MSR[CE] ǀ MSR[GS]) & TCR[WIE], a watchdog timer interrupt is taken.
34–35	WRS	Watchdog timer reset status. Cleared at reset; set to TCR[WRC] when a reset is caused by the watchdog timer. 00 No watchdog timer reset occurred. xx All other values are implementation-dependent.
36	DIS	Decrementer interrupt status. 0 A decrementer event has not occurred. 1 A decrementer event occurred. When (MSR[EE] ǀ MSR[GS]) & TCR[DIE], a decrementer interrupt is taken.

Table 3-28. TSR Field Descriptions (continued)

Bits	Name	Description
37	FIS	Fixed-interval timer interrupt status. 0 A fixed-interval timer event did not occur. 1 A fixed-interval timer event occurred. When MSR[EE] | MSR[GS]) & TCR[FIE], a fixed-interval timer interrupt is taken.
38–63	—	Reserved, should be cleared.

3.7.3 Time Base (TB)

The time base (TB) is a 64-bit register, seen in the following figure, composed of several ports using different SPR numbers. In general, the time base upper (TBU) refers to the high-order 32 bits and the time base lower (TBL) refers to the low-order 32 bits. See Table 3-2 for information about how the SPR numbers relate to which bits of the TB. TB provides timing functions for the system. TB is a volatile resource and must be initialized during start-up. That is, the TB is not required to be initialized by hardware to 0 during reset.

Writing the Time Base is hypervisor privileged.

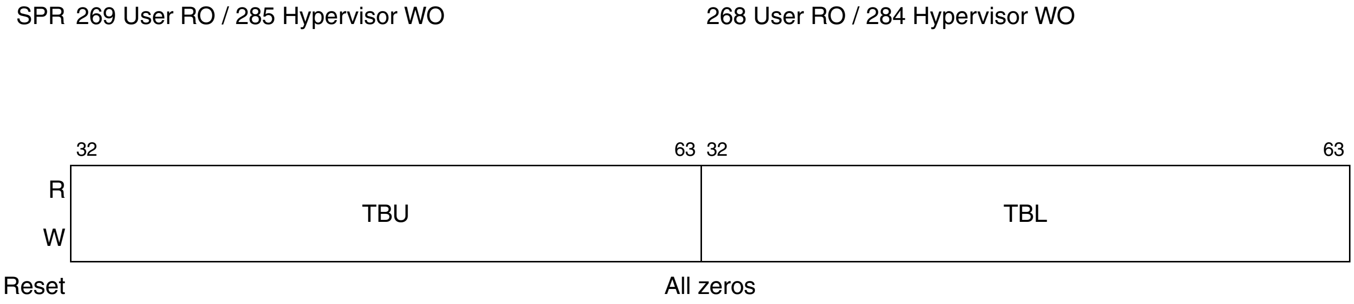

SPR 269 User RO / 285 Hypervisor WO
268 User RO / 284 Hypervisor WO
32
63
32
63
R
W
TBU
TBL
Reset
All zeros

Figure 3-22. Time Base Upper/Lower Registers (TBU/TBL)

TB is interpreted as a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the least-significant bit. The frequency at which the integer is updated is implementation-dependent.

In 64-bit mode, reading the TBL register will read all 64 bits of the time base. <64-bit>

In 64-bit mode, writing the TBL register will only write the lower 32 bits of the time base. <64-bit>

TBL increments until its value becomes 0xFFFF_FFFF (2^{32} – 1). At the next increment, its value becomes 0x0000_0000 and TBU is incremented. This process continues until the TBU value becomes 0xFFFF_FFFF and value TBL value becomes 0xFFFF_FFFF (TB is interpreted as 0xFFFF_FFFF_FFFF_FFFF (2^{64} – 1)). At the next increment, the TBU value becomes 0x0000_0000 and the TBL value becomes 0x0000_0000. There is no interrupt (or any other indication) when this occurs.

The period depends on the driving frequency. For example, if TB is driven by 1 GHz divided by 32, the TB period is as follows:

$$T_{TB} = 2^{64} \times \frac{32}{1\ \text{GHz}} = 5.90 \times 10^{11}\ \text{seconds} \qquad \text{(approximately 18,700 years)}$$

TB is implemented such that the following requirements are satisfied:

- Loading a GPR from the TB has no effect on the accuracy of the TB.
- Storing a GPR to the TB replaces the TB value with the GPR value.

The architecture does not specify a relationship between the frequency at which the TB is updated and other frequencies, such as the CPU clock or bus clock in an integrated device. The TB update frequency is not required to be constant. One of the following is required to ensure that system software can keep time of day and operate interval timers:

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the TB changes and a way to determine the current update frequency.
- The update frequency of the TB is under the control of system software.

NOTE: Software Considerations

- If the TB is initialized on power-on to some reasonable value and the update frequency of the TB is constant, the TB can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.
- Even if the update frequency is not constant, values read from the TB are monotonically increasing (except when the TB wraps from $2^{64} – 1$ to 0). If a trace entry is recorded each time the update frequency changes, the sequence of TB values can be post-processed to become actual time values.
- Successive readings of the TB may return identical values. See Section 8.3.3, “Computing Time of Day from the Time Base,” for ways to compute time of day in POSIX format from the TB.

3.7.4 Decrementer Register (DEC)

The DEC register, shown in the following figure, is a decrementing counter that is updated at the same rate as the TB. It provides a way to signal a decrementer interrupt after a specified period unless one of the following occurs:

- DEC is altered by software in the interim.
- The TB update frequency changes.

DEC is typically used as a general-purpose software timer. The decrementer auto-reload register can be used to automatically reload a programmed value into DEC, as described in Section 3.7.5, “Decrementer Auto-Reload Register (DECAR).”

This register is hypervisor privileged.

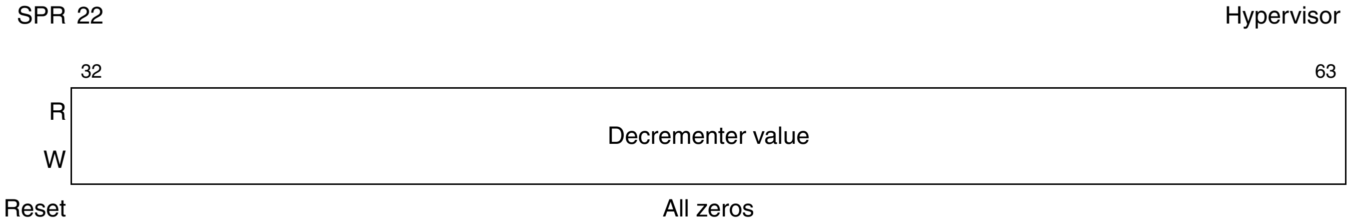


Figure 3-23. Decrementer Register (DEC)

3.7.5 Decrementer Auto-Reload Register (DECAR)

DECAR is shown in the following figure. If the auto-reload function is enabled (TCR[ARE] = 1), the auto-reload value in DECAR is written to DEC when DEC decrements from 0x0000_0001 to 0x0000_0000. Note that writing DEC with zeros by using **mtspr** does not automatically generate a decrementer exception.

This register is hypervisor privileged.

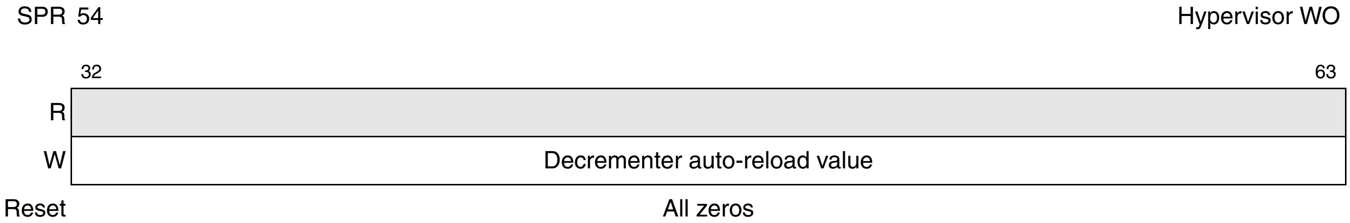


Figure 3-24. Decrementer Auto-Reload Register (DECAR)

Although DECAR is write-only, some implementations have allowed read access to DECAR. Software should not depend on being able to read DECAR.

3.7.6 Alternate Time Base Registers (ATB)

Alternate Time Base Upper (ATBU) and Alternate Time Base Lower (ATBL) together form the alternate time base counter (ATB), shown in the following figure. The ATB contains a 64-bit unsigned integer that is incremented periodically at an implementation-defined frequency. Like the TB implementation, ATBL is an aliased name for ATB and provides read-only access to the 64-bit alternate time base counter.

The alternate time base increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1). At the next increment, its value becomes 0x0000_0000_0000_0000. There is no explicit indication (such as an interrupt) that this has occurred.

In 64-bit mode, reading the ATBL register will read all 64 bits of the alternate time base. <64-bit>

The effect of entering a power-savings mode or of processor frequency changes on counting in the alternate time base is implementation-dependent.

SPR 527 User RO SPR 526 User RO

	0 … 31	32 … 63
R	ATBU	ATBL
W		
Reset	All zeros	All zeros

Figure 3-25. Alternate Time Base Register Upper/Lower (ATBU/ATBL)

3.8 Interrupt Registers

This section describes registers used for interrupt handling.

3.8.1 Save/Restore Registers 0 (SRR0, GSRR0, CSRR0, DSRR0, MCSRR0)

Each interrupt class has a save/restore register 0, shown in Figure 3-26, that holds the address of the instruction where the interrupted process should resume. The instruction is interrupt-specific, although for instruction-caused exceptions, it is typically the address of the instruction that caused the interrupt.

When the appropriate return from interrupt instruction (**rfi**, **rfci**, **rfdi**, **rfgi**, or **rfmci**) executes, instruction execution continues at the address in the corresponding *x*SRR0.

For 64-bit implementations, the contents of *x*SRR0 when a machine check interrupt is taken reflects the computation mode specified by MSR[CM] and the computation mode entered for execution of the machine check interrupt (specified by EPCR[ICM] or EPCR[GICM] <E.HV>). The contents of *x*SRR0 upon machine check interrupt can be described as follows (assuming Addr is the address to be put into *x*SRR0):

```
if E.HV implemented and interrupt directed to guest supervisor state
    then newCM ← EPCR$_{GICM}$
    else newCM ← EPCR$_{ICM}$           // directed to hypervisor state
if (MSR$_{CM}$ = 0) & (newCM = 0)       // 32-bit to 32-bit mode
    then xSRR0 ← $^{32}$undefined || Addr$_{32:63}$
if (MSR$_{CM}$ = 0) & (newCM = 1)       // 32-bit to 64-bit mode
    then xSRR0 ← $^{32}$0 || Addr$_{32:63}$
if (MSR$_{CM}$ = 1) & (newCM = 1)       // 64-bit to 64-bit mode
    then xSRR0 ← Addr$_{32:63}$
if (MSR$_{CM}$ = 1) & (newCM = 0)       // 64-bit to 32-bit mode
    then xSRR0 ← undefined
```

<Embedded.Hypervisor>:
If the processor is in the guest supervisor state, reading SRR0 with **mfspr** and writing SRR0 with **mtspr** is mapped to Guest Save/Restore Register 1 (GSRR0). See Section 3.2.2.1, “SPR Register Mapping <E.HV>.”

CSRR0 is hypervisor privileged. <E.HV>

DSRR0 is hypervisor privileged. <E.HV>

MCSRR0 is hypervisor privileged. <E.HV>

SPR 26 (SRR0)[1]	Guest supervisor
SPR 378 (GSRR0) <E.HV>	Guest supervisor
SPR 57 (CSRR0)	Hypervisor
SPR 574 (DSRR0)	Hypervisor
SPR 570 (MCSRR0)	Hypervisor

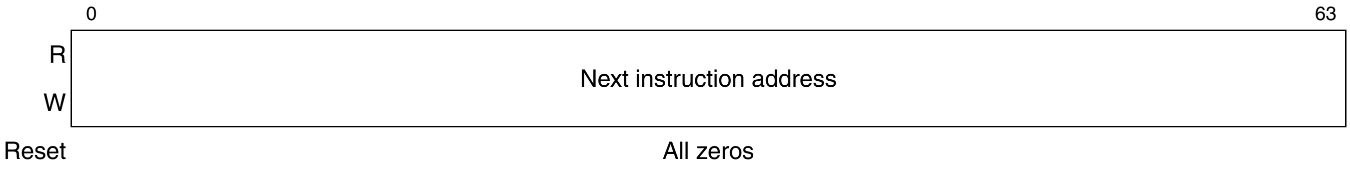


Figure 3-26. Save/Restore Register 0 (*x*SRR0)

[1] Guest supervisor state **mfspr** and **mtspr** accesses are mapped to GSRR0.<E.HV>

NOTE: Software Considerations

> Operating system software should not directly access GSRR0, but should instead perform **mfspr** and **mtspr** to SRR0. This makes the programming model the same whether the operating system is running as a guest under a hypervisor or is running bare-metal.

3.8.2 Save/Restore Register 1 (SRR1, GSRR1, CSRR1, DSRR1, MCSRR1)

A save/restore 1 register (Figure 3-27) is provided to save and restore machine state for each interrupt class. When an interrupt is taken, MSR contents are placed in *x*SRR1. When the appropriate return from interrupt instruction (**rfi**, **rfci**, **rfdi**, **rfgi**, or **rfmci**) executes, the corresponding *x*SRR1 contents are placed into MSR. *x*SRR1 bits that correspond to reserved MSR bits are also reserved.

<Embedded.Hypervisor>:
If an **rfi** or **rfgi** instruction is executed when in guest supervisor state, some bits from GSRR1 are not copied to protected bits of MSR. See Section 7.7, “Interrupt Processing.”

<Embedded.Hypervisor>:
If the processor is in the guest supervisor state, reading SRR1 with **mfspr** and writing SRR1 with **mtspr** is mapped to Guest Save/Restore Register 1 (GSRR1). See Section 3.2.2.1, “SPR Register Mapping <E.HV>.”

CSRR1 is hypervisor privileged. <E.HV>

DSRR1 is hypervisor privileged. <E.HV>

MCSRR1 is hypervisor privileged. <E.HV>

NOTE: Software Considerations

> Reserved MSR bits may be modified by a return from interrupt.

SPR 27 (SRR1)[1]	Guest supervisor
SPR 379 (GSRR1) <E.HV>	Guest supervisor
SPR 59 (CSRR1)	Hypervisor
SPR 575 (DSRR1)	Hypervisor
SPR 571 (MCSRR1)	Hypervisor

	32 … 63
R W	MSR state information
Reset	All zeros

Figure 3-27. Save/Restore Register 1(*x*SRR1)

[1] Guest supervisor state **mfspr** and **mtspr** accesses are mapped to GSRR1.<E.HV>

NOTE: Software Considerations

> Operating system software should not directly access GSRR1, but should instead perform **mfspr** and **mtspr** to SRR1. This makes the programming model the same whether the operating system is running as a guest under a hypervisor or is running bare-metal.

3.8.3 Interrupt Vector Prefix Register (IVPR)

IVPR is used with IVORs to determine the vector address of where execution is to begin when an interrupt occurs. IVPR[0–47] provides the high-order 48 bits of the address of the exception processing routines. The 16-bit vector offsets defined by IVOR registers for each interrupt are concatenated to the right of IVPR[0–47] to form the address of the exception processing routine. Shown in the following figure, IVPR[48–63] are reserved.

<Embedded.Hypervisor>:
IVPR and IVORs are used only to form the interrupt vector address for interrupts that are directed to the hypervisor state. GIVPR and GIVORs are used for interrupts directed to the guest supervisor state.

This register is hypervisor privileged.

SPR 63	Hypervisor

	0 … 47	48 … 63
R W	Interrupt vector prefix	—
Reset	All zeros	

Figure 3-28. Interrupt Vector Prefix Register (IVPR)

3.8.4 Guest Interrupt Vector Prefix Register (GIVPR) <E.HV>

GIVPR is used with GIVORs to determine the vector address of where execution is to begin when an interrupt occurs that is directed to the guest supervisor state. GIVPR[0–47] provides the high-order 48 bits of the address of the exception processing routines. The 16-bit vector offsets are concatenated to the right of GIVPR[0–47] to form the address of the exception processing routine. Shown in the following figure, GIVPR[48–63] are reserved.

GIVPR and GIVORs are used only for interrupts directed to the guest supervisor state. IVPR and IVORs are used to form the interrupt vector address for interrupts that are directed to the hypervisor state.

This register is hypervisor privileged for **mtspr** and guest supervisor privileged for **mfspr**.

.

SPR 447 Hypervisor[1]

	0 … 47	48 … 63
R W	Interrupt vector prefix	—
Reset	All zeros	

Figure 3-29. Guest Interrupt Vector Prefix Register (GIVPR) Format

[1] Guest supervisor state is allowed to read GIVPR using **mfspr**.

NOTE: Software Considerations

> Hypervisor software should emulate guest supervisor writes to IVPR and place the guest’s interrupt prefix into GIVPR. Operating system software should not directly access GIVPR, but should instead perform **mfspr** and **mtspr** to IVPR. This makes the programming model the same whether the operating system is running as a guest under a hypervisor or is running bare-metal.

3.8.5 Interrupt Vector Offset Registers (IVORs)

IVORs, shown in the following figure, hold the interrupt-specific quad-word index that, when concatenated with the base address provided by the IVPR, forms the interrupt vector.

SPR See Table 3-29. Hypervisor

	32 … 47	48 … 59	60 … 63
R W	—	Interrupt vector offset	—
Reset	All zeros		

Figure 3-30. Interrupt Vector Offset Registers (IVOR)

These registers are hypervisor privileged.

This table shows the IVOR assignments.

Table 3-29. IVOR Assignments

IVOR Number	SPR	Interrupt Type
IVOR0	400	Critical input interrupt
IVOR1	401	Machine check interrupt
IVOR2	402	Data storage interrupt
IVOR3	403	Instruction storage interrupt
IVOR4	404	External input interrupt
IVOR5	405	Alignment interrupt
IVOR6	406	Program interrupt
IVOR7	407	Floating-point unavailable interrupt
IVOR8	408	System call interrupt
IVOR9	409	Auxiliary processor unavailable interrupt
IVOR10	410	Decrementer interrupt
IVOR11	411	Fixed-interval timer interrupt
IVOR12	412	Watchdog timer interrupt
IVOR13	413	Data TLB error interrupt
IVOR14	414	Instruction TLB error interrupt
IVOR15	415	Debug interrupt
IVOR16–IVOR31	—	Reserved for future architectural use
IVOR32	528	SPE <E.SP>/embedded floating-point <SP.FD,SP.FV>/vector unavailable interrupt <V>
IVOR33	529	Embedded floating-point data exception <SP.FD,SP.FS,SP.FV>/AltiVec assist interrupt <V>
IVOR34	530	Embedded floating-point round exception <SP.FD,SP.FS,SP.FV>
IVOR35	531	Embedded performance monitor <E.PM>
IVOR36	532	Processor doorbell interrupt <E.PC>
IVOR37	533	Processor doorbell critical interrupt <E.PC>
IVOR38	534	Guest Processor doorbell interrupt <E.HV>
IVOR39	535	Guest Processor doorbell critical interrupt <E.HV>
IVOR40	536	Embedded hypervisor system call interrupt <E.HV>
IVOR41	537	Embedded hypervisor privilege interrupt <E.HV>

3.8.6 Guest Interrupt Vector Offset Registers (GIVORs) <E.HV>

GIVORs, shown in the following figure, hold the interrupt-specific quad-word index that, when concatenated with the base address provided by the GIVPR, forms the interrupt vector for interrupts directed to the guest state.

SPR See Table 3-30. Hypervisor[1]

	32–47	48–59	60–63
R W	—	Interrupt vector offset	—
Reset	All zeros		

Figure 3-31. Guest Interrupt Vector Offset Registers G(IVOR)

[1] Guest supervisor state is allowed to read GIVORx using **mfspr**.

These registers are hypervisor privileged for **mtspr** and guest supervisor privileged for **mfspr**.

This table shows the GIVOR assignments.

Table 3-30. GIVOR Assignments

IVOR Number	SPR	Interrupt Type
GIVOR2	440	Guest data storage interrupt
GIVOR3	441	Guest instruction storage interrupt
GIVOR4	442	Guest external input interrupt
GIVOR8	443	Guest system call interrupt
GIVOR13	444	Guest data TLB error interrupt
GIVOR14	445	Guest instruction TLB error interrupt

NOTE: Software Considerations

- Hypervisor software should emulate guest supervisor writes to IVORs and place the guest’s interrupt offsets into GIVORs. Operating system software should not directly access GIVOR, but should instead perform **mfspr** and **mtspr** to IVORs. This makes the programming model the same whether the operating system is running as a guest under a hypervisor or is running bare-metal.
- Some processors that support category E.HV do not allow the performance monitor interrupt to be directed to the guest and thus do not implement GIVOR35.

3.8.7 Exception Syndrome Register (ESR)

ESR, shown in the following figure, provides a syndrome to differentiate among exceptions that can generate the same interrupt type. When such an interrupt is generated, bits corresponding to the exception that caused the interrupt are set and all other ESR bits are cleared. Other interrupt types do not affect ESR contents. The ESR does not need to be cleared by software.

<Embedded.Hypervisor>:
If the processor is in the guest supervisor state, reading ESR with **mfspr** and writing ESR with **mtspr** is mapped to Guest Exception Syndrome Register (GESR). See Section 3.2.2.1, “SPR Register Mapping <E.HV>.”

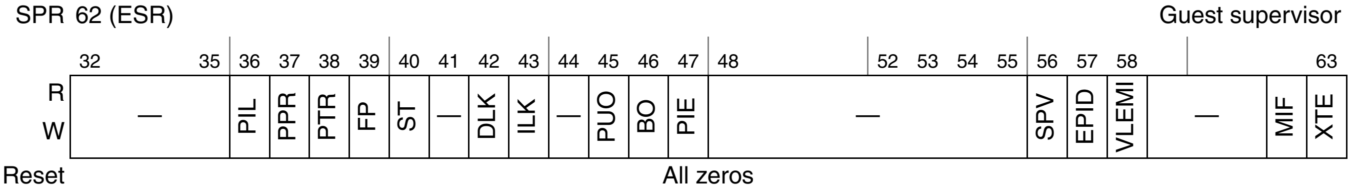


Figure 3-32. Exception Syndrome Register (ESR)

This table describes ESR bit definitions.

Table 3-31. Exception Syndrome Register (ESR) Definition

Bits	Name	Syndrome	Interrupt Types
32–35	—	Reserved, should be cleared.	—
36	PIL	Illegal instruction exception	Program
37	PPR	Privileged instruction exception	Program
38	PTR	Trap exception	Program
39	FP	Floating-point operations	Alignment, data storage, data TLB, program
40	ST	Store operation	Alignment, data storage, data TLB error
41	—	Reserved, should be cleared.	—
42	DLK	Data cache locking attempt. Set when a DSI occurs because **dcbtls**, **dcbtstls**, or **dcblc** executed in user mode when MSR[UCLE] = 0. 0 Default 1 DSI occurred on an attempt to lock line in data cache when MSR[UCLE] = 0.	Data storage
43	ILK	Instruction cache locking attempt. Set when a DSI occurs because an **icbtls** or **icblc** was executed in user mode while MSR[UCLE] = 0. 0 Default 1 DSI occurred on an attempt to lock instruction cache line and MSR[UCLE] = 0.	Data storage
44	—	Reserved, should be cleared. Was defined by Book E as AP.	—
45	PUO	Unimplemented operation exception	Program
46	BO	Byte-ordering exception	Data storage, instruction storage
47	PIE	Imprecise exception	Program
48–55	—	Reserved, should be cleared	—

Table 3-31. Exception Syndrome Register (ESR) Definition (continued)

Bits	Name	Syndrome	Interrupt Types
56	SPV[1]	0 Default 1 Any exception caused by an SPE/embedded floating-point/AltiVec instruction occurred.	Alignment, data storage, data TLB, embedded floating-point data, embedded floating-point round, SPE/embedded floating-point/AltiVec unavailable, AltiVec assist
57	EPID <E.PD>	External PID load and store. Translation was performed using context from EPLC or EPSC. Set when an alignment, DSI or a DTLB error occurs during a load or store by external PID instruction. 0 Default 1 Alignment, DSI, or DTLB error occurred during a load or store by external PID instruction.	Alignment, data storage, data TLB
58	VLEMI <VLE>	VLEMI indicates that an interrupt was caused by a VLE instruction. VLEMI is set on an exception associated with execution or attempted execution of a VLE instruction. 0 The instruction page associated with the instruction causing the exception does not have the VLE attribute set or the category VLE is not implemented. 1 The instruction page associated with the instruction causing the exception has the VLE attribute set and category VLE is implemented.	Alignment, data storage, data TLB, SPE/embedded floating-point/AltiVec unavailable, embedded floating-point data, embedded floating-point round, instruction storage, program, system call
59–61	—	Reserved, should be cleared	—
62	MIF <VLE>	Indicates that an interrupt was caused by a misaligned instruction fetch (NIA_{62} != 0) and the VLE attribute is cleared for the page or the second half of a 32-bit VLE instruction caused an instruction TLB error. 0 Default. 1 NIA_{62} != 0 and the instruction page associated with NIA does not have the VLE attribute set or the second half of a 32-bit VLE instruction caused an instruction TLB error.	Instruction TLB error, Instruction storage
63	XTE	External transaction error. An external transaction reported an error that the core handled precisely. *x*SRR0 contains the address of the instruction that initiated the transaction. XTE is not supported by all Freescale devices and may not be supported on future devices. 0 Default. No external transaction error was precisely detected. 1 An external transaction reported an error that was precisely detected.	Instruction storage, Data storage

[1] Formerly SPE (for signal processing engine) or AV (for AltiVec)

NOTE: Software Considerations

ESR information is incomplete, so system software may need to identify the type of instruction that caused the interrupt and examine the TLB entry and the ESR to fully identify the exception or exceptions. For example, a data storage interrupt may be caused by both a protection violation exception and a byte-ordering exception. System software would have to look beyond ESR[BO], such as the state of MSR[PR] in SRR1 and the TLB entry page protection bits to determine whether a protection violation also occurred.

3.8.8 Guest Exception Syndrome Register (GESR) <E.HV>

GESR, shown in Figure 3-33, provides a syndrome to differentiate among exceptions that can generate the same interrupt type. When such an interrupt is generated and directed to the guest supervisor state, bits corresponding to the exception that caused the interrupt are set and all other GESR bits are cleared. Other interrupt types do not affect GESR contents. The GESR does not need to be cleared by guest supervisor software. The bit definitions for GESR are the same as ESR.

If the processor is in the guest supervisor state, reading ESR with **mfspr** and writing ESR with **mtspr** is mapped to Guest Exception Syndrome Register (GESR). See Section 3.2.2.1, "SPR Register Mapping <E.HV>."

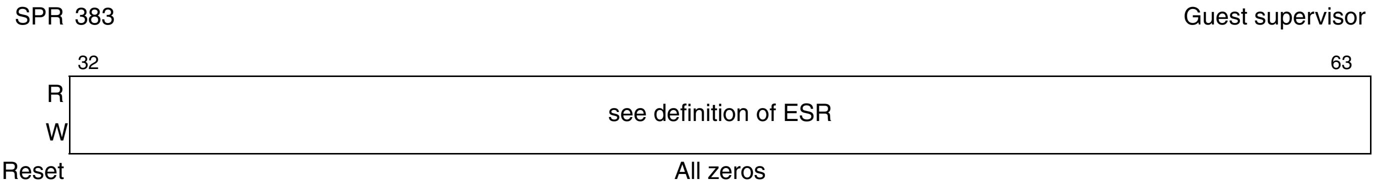


Figure 3-33. Guest Exception Syndrome Register (ESR)

NOTE: Software Considerations

Operating system software should not directly access GESR, but should instead perform **mfspr** and **mtspr** to ESR. This makes the programming model the same whether the operating system is running as a guest under a hypervisor or is running bare-metal.

NOTE: Architecture Considerations

GESR contents are updated by hardware when an interrupt is directed to the guest state. Hypervisor software should also update GESR directly when reflecting or synthesizing an interrupt to the guest.

3.8.9 Data Exception Address Register (DEAR)

DEAR, shown in Figure 3-34, is loaded with the effective address of a data access (caused by a load, store, or cache management instruction) that results in an alignment, data TLB miss, or DSI exception. The effective address loaded is the effective address of a byte that is in the range of the bytes being accessed and whithin the page whose access caused the exception. The setting of DEAR is mode dependent and are described as follows (assuming Addr is the address to be put into DEAR):

```
if ($MSR_{CM}$ = 0) & ($EPCR_{ICM}$ = 0) then
    DEAR ← $^{32}$undefined || $Addr_{32:63}$
if ($MSR_{CM}$ = 0) & ($EPCR_{ICM}$ = 1) then
    DEAR ← $^{32}$0 || $Addr_{32:63}$
if ($MSR_{CM}$ = 1) & ($EPCR_{ICM}$ = 1) then
    DEAR ← $Addr_{0:63}$
if ($MSR_{CM}$ = 1) & ($EPCR_{ICM}$ = 0) then
    DEAR ← undefined
```

<Embedded.Hypervisor>:
If the processor is in the guest supervisor state, reading DEAR with **mfspr** and writing DEAR with **mtspr**

is mapped to Guest Data Exception Address Register (GDEAR). See Section 3.2.2.1, “SPR Register Mapping <E.HV>.”

.

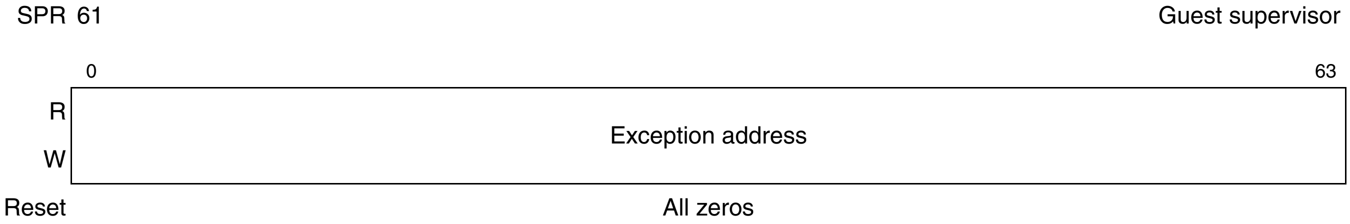


Figure 3-34. Data Exception Address Register (DEAR)

3.8.10 Guest Data Exception Address Register (GDEAR) <E.HV>

GDEAR, shown in Figure 3-34, is loaded with the effective address of a data access (caused by a load, store, or cache management instruction) that results in a data TLB miss or DSI exception that is directed to the guest supervisor state. The effective address loaded is the effective address of a byte that is in the range of the bytes being accessed and whithin the page whose access caused the exception. The setting of GDEAR is mode dependent and are described as follows (assuming Addr is the address to be put into GDEAR):

```
if ($MSR_{CM}$ = 0) & ($EPCR_{GICM}$ = 0) then
    GDEAR ← $^{32}$undefined || $Addr_{32:63}$
if ($MSR_{CM}$ = 0) & ($EPCR_{GICM}$ = 1) then
    GDEAR ← $^{32}$0 || $Addr_{32:63}$
if ($MSR_{CM}$ = 1) & ($EPCR_{GICM}$ = 1) then
    GDEAR ← $Addr_{0:63}$
if ($MSR_{CM}$ = 1) & ($EPCR_{GICM}$ = 0) then
    GDEAR ← undefined
```

If the processor is in the guest supervisor state, reading DEAR with **mfspr** and writing DEAR with **mtspr** is mapped to Guest Data Exception Address Register (GDEAR). See Section 3.2.2.1, “SPR Register Mapping <E.HV>.”

.

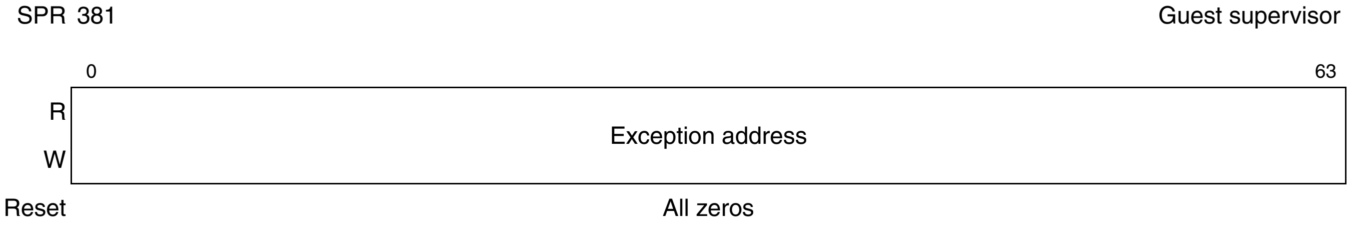


Figure 3-35. Data Exception Address Register (DEAR)

NOTE: Software Considerations

Operating system software should not directly access GDEAR, but should instead perform **mfspr** and **mtspr** to DEAR. This makes the programming model the same whether the operating system is running as a guest under a hypervisor or is running bare-metal.

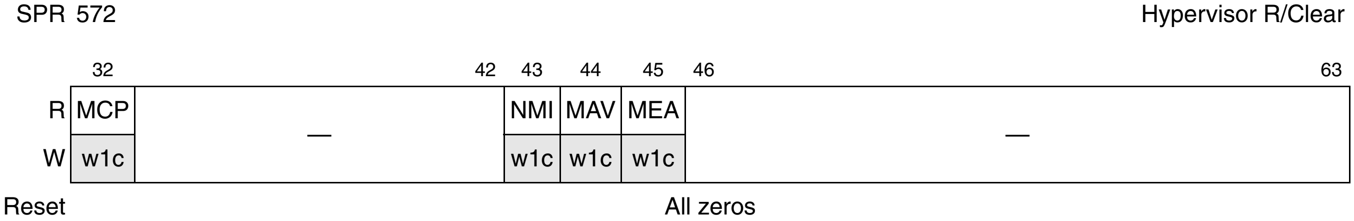
3.8.11 Machine Check Syndrome Register (MCSR)

The MCSR records the cause of machine check interrupts, error report interrupts, and NMI. In general, bits correlating to specific hardware errors are implementation dependent. Not all devices implement the fields shown in Figure 3-36 and typically, each processor implements additional MCSR fields; consult the Core reference Manual

Some processors treat bits corresponding to asynchronous machine check conditions as level sensitive, and when any of these bits are set, if the machine check interrupt is enabled, a machine check interrupt occurs. See Section 7.8.2, “Machine Check, NMI, and Error Report Interrupts.”

This register is hypervisor privileged.

MCSR cannot be written with a specific value by software. Individual bits may be cleared by writing a bit mask with bits set to 1 in the corresponding positions it wishes to clear.



SPR 572 Hypervisor R/Clear

	32	42	43	44	45	46 63
R	MCP	—	NMI	MAV	MEA	—
W	w1c		w1c	w1c	w1c	
Reset	All zeros					

Figure 3-36. Machine Check Syndrome Register 1 (MCSR)

This table describes the MCSR fields.

Table 3-32. MCSR Field Descriptions

Bits	Name	Description
32	MCP	Machine check input signal to core. Processor cores with a machine check input pin (signal) respond to a signal input by producing an asynchronous machine check. The existence of such a signal and how such a signal is generated is implementation dependent and may be tied to a an external pin on the IC package. This is bit is an asynchronous machine check condition.
33–42	—	Implementation-dependent.
43	NMI	Nonmaskable Interrupt. Set if a nonmaskable interrupt (NMI) has been sent to the virtual processor.
44	MAV	MCAR address valid. The address contained in MCAR was updated by the processor and corresponds to the first detected error condition that contained an associated address. Any subsequent machine check errors that have associated addresses are not placed in MCAR unless MAV is 0 when the error is logged. 0 The address in MCAR is not valid. 1 The address in MCAR is valid. Note: Software should read MCAR before clearing MAV. MAV should be cleared before setting MSR[ME].
45	MEA	MCAR effective address. Denotes the type of address in MCAR. MEA has meaning only if MCSR[MAV] = 1. 0 The address in MCAR is a physical address. 1 The address in MCAR is an effective address (untranslated).
46–63	—	Implementation-dependent.

NOTE: Software Considerations

> The machine check interrupt handler should always write what is read back to the MCSR after the error information has been logged. Writing contents that were read from the MCSR back to the MCSR clears only those status bits that were previously read. Failure to clear all MCSR bits causes an asynchronous machine check interrupt when MSR[ME] is set.

3.8.12 Machine Check Address Register (MCAR/MCARU)

When the core takes a machine check interrupt, it updates MCAR, shown in Figure 3-37, to indicate the address of the data associated with the machine check. Errors that update MCAR contents are implementation-dependent. An error which has an address associated with it will update MCAR/MCARU with the address if MCSR[MAV] = 0, and will set MCSR[MAV] to prevent other errors with addresses from overwriting MCAR/MCARU until software has read the error address and cleared MCSR[MAV]. MCSR[MEA] is set to 1 when MCAR/MCARU is set to denote if the address is an effective address, or set to 0 to denote that the address is a physical address.

These registers are hypervisor privileged.

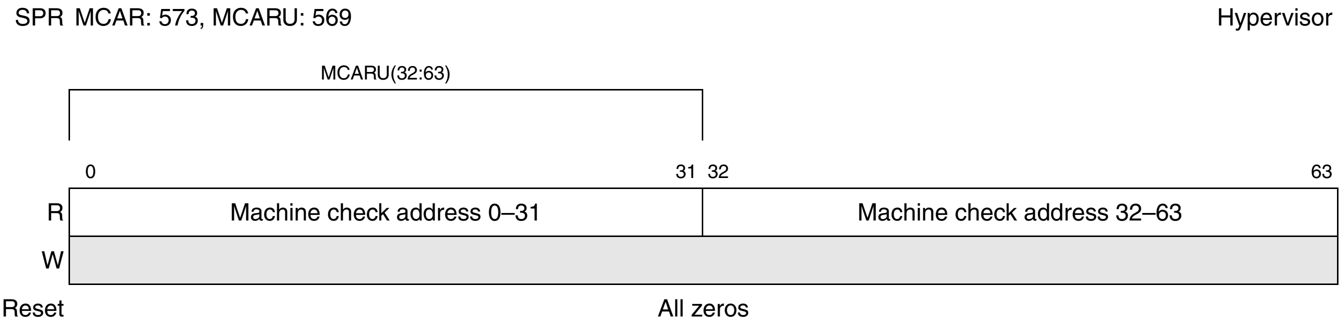

SPR MCAR: 573, MCARU: 569
Hypervisor
MCARU(32:63)
0
31
32
63
R
Machine check address 0–31
Machine check address 32–63
W
Reset
All zeros

Figure 3-37. Machine Check Address Register (MCAR/MCARU)

For 32-bit implementations that support physical addresses greater than 32 bits, MCARU provides an alias to the upper address bits that reside in MCAR[0–31].

Note that because on an interrupt, MCSR[MAV] indicates whether the address in MCAR is valid, software should read MCAR before clearing MAV. See Section 3.8.11, “Machine Check Syndrome Register (MCSR).”

3.8.13 External Proxy Register (EPR) <EXP>

The external proxy register (EPR), shown in Figure 3-38, conveys the peripheral-specific interrupt vector associated with the external input interrupt triggered by the programmable interrupt controller (PIC) in the integrated device when an external input interrupt is taken. See the Integrated Device Reference Manual for more details. The EPR is only considered valid from the time the external input interrupt is taken until appropriate enabling conditions are present for another external input interrupt to occur. The external proxy facility is described in Section 7.8.5.1, “External Proxy <EXP>.”

This register is hypervisor privileged.

<Embedded.Hypervisor>:
If the processor is in the guest supervisor state, reading EPR with **mfspr** is mapped to Guest External Proxy Register (GEPR). See Section 3.2.2.1, “SPR Register Mapping <E.HV>.”

SPR 702 (EPR) Guest supervisor RO

	32 … 47	48 … 63
R W	—	VECTOR
Reset	All zeros	

Figure 3-38. External Proxy Register (EPR)

NOTE: Software Considerations

Writing to EPR in guest supervisor state with **mtspr** is not mapped and produces an embedded hypervisor privilege exception.

3.8.14 Guest External Proxy Register (GEPR) <EXP,E.HV>

The guest external proxy register (GEPR), shown in Figure 3-38, conveys the peripheral-specific interrupt vector associated with the external input interrupt triggered by the programmable interrupt controller (PIC) in the integrated device when an external input interrupt is taken in the guest state. See the Integrated Device Reference Manual for more details. The GEPR is only considered valid from the time the external input interrupt is taken until appropriate enabling conditions are present for another external input interrupt to occur. The external proxy facility is described in Section 7.8.5.1, “External Proxy <EXP>.”

<Embedded.Hypervisor>:
If the processor is in the guest supervisor state, reading EPR with **mfspr** and writing EPR with **mtspr** is mapped to Guest External Proxy Register (GEPR). See Section 3.2.2.1, “SPR Register Mapping <E.HV>.”

SPR 380 Guest supervisor

	32 … 47	48 … 63
R W	—	VECTOR
Reset	All zeros	

Figure 3-39. Guest External Proxy Register (GEPR)

NOTE: Software Considerations

Operating system software should not directly access GEPR, but should instead perform **mfspr** and **mtspr** to EPR. This makes the programming model the same whether the operating system is running as a guest under a hypervisor or is running bare-metal.

3.9 Software-Use SPRs (SPRGs, GSPRGs<E.HV>, and USPRGs)

Software-use SPRs (SPRGs, GSPRGs <E.HV>, and USPRG0, shown in Figure 3-40) have no defined functionality, but may be used for any purpose software wishes. All processors implement SPRG[0–3].

Other software-use SPRGs may not be implemented, or processors may implement additional SPRs, and may provide different access support than those indicated here. Refer to the processor documentation.

Note that most SPRGs have multiple ports to the same physical register. For example SPRG4 is a single physical register, but has 2 ports (SPR 276 and SPR 260). The ports allow access at different privilege levels.

<Embedded.Hypervisor>:
If the processor is in the guest supervisor state, reading SPRG[0–3] with **mfspr** and writing SPRG[0–3] with **mtspr** is mapped to GSPRG[0–3]. If the processor is in the guest user state, reading SPRG3 (SPR 259) with **mfspr** is mapped to GSPRG3. See Section 3.2.2.1, “SPR Register Mapping <E.HV>.”

SPR	SPRG0	272	Guest supervisor
	SPRG1	273	Guest supervisor
	SPRG2	274	Guest supervisor
	SPRG3	275	Guest supervisor
		259	User RO
	SPRG4	276	Guest supervisor
		260	User RO
	SPRG5	277	Guest supervisor
		261	User RO
	SPRG6	278	Guest supervisor
		262	User RO
	SPRG7	279	Guest supervisor
		263	User RO
	SPRG8	604	Hypervisor
	SPRG9	605	Guest supervisor
	USPRG0	256	User
	GSPRG0	368	Guest supervisor
	GSPRG1	369	Guest supervisor
	GSPRG2	370	Guest supervisor
	GSPRG3	371	Guest supervisor

	0 … 63
R	–
W	
Reset	undefined

Figure 3-40. Software-Use SPRs (SPRGs, GSPRGs, and USPRG0)

USPRG0 is a separate physical register from SPRG0 and is an alias to VRSAVE.

NOTE: Software Considerations

Operating system software should not directly access GSPRG[0–3], but should instead perform **mfspr** and **mtspr** to SPRG[0–3]. This makes the programming model the same whether the operating system is running as a guest under a hypervisor or is running bare-metal.

3.10 L1 Cache Registers

L1 cache registers provide control, configuration, and status information for the primary (L1) caches.

3.10.1 L1 Cache Control and Status Register 0 (L1CSR0)

L1CSR0, shown in Figure 3-41, is used for general control and status of the L1 data cache. Not all processors implement all bits. Some processors may implement additional implementation specific bits in the unused fields.

If a processor implements a unified (instruction and data) L1 cache, L1CSR0 applies to the unified cache and L1CSR1 is not implemented.

With respect to errors, the term “error detection” means an error that was detected, but the error is not directly correctable (for example, a parity error is not directly correctable because the correct value cannot be constructed). Errors that are “detected”, may be correctable by other means in some cases by performing invalidations if the state can be recovered from elsewhere in the system. The term “correct” or “correction” means an error that has been corrected by hardware by restoring the corrected value when the error is seen.

This register is hypervisor privileged.

Writing this register requires synchronization. See Section 4.5.4.3, “Synchronization Requirements.”

SPR 1010 Hypervisor

	32 35	36 39	40	41	42	43	44 45	46	47
R W	WID	WDD	AWID	AWDD	WAM	—	CEA	—	CECE
Reset	All zeros								

	48	49	50 51	52	53	54	55	56	57 58	59	60	61	62	63
R W	CEI	—	CEDT	CSLC	CUL	CLO	CLFC	CLOA	CEIT	—	DCBZ32	CABT	CFI	CE
Reset	All zeros													

Figure 3-41. L1 Cache Control and Status Register 0 (L1CSR0)

This table describes L1CSR0 fields.

Table 3-33. L1CSR0 Field Descriptions

Bits	Name	Description
32–35	WID	Cache way partitioning. <CWP> Way instruction disable. (bit 32 = way 0, bit 33 = way 1, … bit 35 = way 3). 0 The corresponding way is available for replacement by instruction miss line refills. 1 The corresponding way is not available for replacement by instruction miss line refills.
36–39	WDD	Cache way partitioning. <CWP> Way data disable (bit 36 = way 0, bit 37 = way 1, … bit 39 = way 3). 0 The corresponding way is available for replacement by data miss line refills. 1 The corresponding way is not available for replacement by data miss line refills

Table 3-33. L1CSR0 Field Descriptions (continued)

Bits	Name	Description
40	AWID	Cache way partitioning. <CWP> Additional ways instruction disable. 0 Additional ways beyond 0–3 are available for replacement by instruction miss line fills. 1 Additional ways beyond 0–3 are not available for replacement by instruction miss line fills.
41	AWDD	Cache way partitioning. <CWP> Additional ways data disable. 0 Additional ways beyond 0–3 are available for replacement by data miss line fills. 1 Additional ways beyond 0–3 are not available for replacement by data miss line fills.
42	WAM	Cache way partitioning. <CWP> Way access mode. 0 All ways are available for access. 1 Only ways partitioned for the specific type of access are used for a fetch or read operation.
43	—	Reserved for implementation dependent use.
44–45	CEA	Data cache error action 00 Error detection causes a machine check interrupt (and possibly error report interrupts). It is implementation dependent whether the error detection causes the data cache location containing the error to be invalidated. Error correction occurs if supported by the implementation and does not result in a machine check interrupt or any data cache invalidations. 01 Error detection causes invalidation of the data cache location containing the error. It is implementation dependent whether a machine check interrupt (and possibly error report interrupts) occurs. Error correction occurs if supported by the implementation and does not result in a machine check interrupt or any data cache invalidations. 10 Error detection causes a machine check interrupt (and possibly error report interrupts) and invalidation of the instruction cache location containing the error. Error correction occurs if supported by the implementation and does not result in a machine check interrupt or any data cache invalidations. 11 Reserved The setting of CEA has no effect if L1CSR0[CECE] = 0. Reading CEA is not guaranteed to reflect the last written value in some implementations, however, it will return either the last written value or 0. **Note:** Some implementations may invalidate the cache line containing the error or the entire data cache when a error is detected. **Note:** If an implementation supports error correction, errors are always corrected when possible. Implementations may not support all defined modes of error action. Setting CEA to a value not supported by the implementation may cause no error detection to occur or may produce undefined results.
46	—	Reserved for implementation dependent use.
47	CECE CPE DCPE	(Data) Cache error checking enable. 0 Error detection of the cache disabled 1 Error detection of the cache enabled **Note:** If an implementation supports error correction, errors are always corrected when possible regardless of the setting of CECE.

Table 3-33. L1CSR0 Field Descriptions (continued)

Bits	Name	Description
48	CEI CPI DCPI	(Data) Cache error injection enable. 0 Error error injection disabled 1 Error injection enabled. Note that cache error checking must also be enabled (L1CSR0[CECE] = 1) when CEI is set. If L1CSR0[CECE] is not set the results are undefined and erratic behavior may occur. It is recommended that an attempt to set this bit when L1CSR0[CECE] = 0 will cause the bit not to be set (that is, L1CSR0[CEI] = L1CSR0[CECE] & L1CSR0[CEI]). Software Note: *When injection is enabled, accesses to the cache cause error conditions to be injected when data is written in the cache. Different implementations may offer varying amounts of error injection. For example, some may inject errors into the particular location where data is written, or some may also inject errors into tags and other data. Such methods are implementation specific, however, when injection is enabled, and a location in the cache is established, later accessing that location will cause an error to be present and detected (or corrected) if enabled unless that location is flushed from the cache prior to access by another mechanism.*
49	—	Reserved, should be cleared.
50–51	CEDT	Data cache error detection/correction type 00 Implementation specific error detection/correction 01 EDC error detection 10 reserved 11 reserved Software Note: *Implementations will generally use the implementation specific value 00 to represent the default error detection and correction that is available.* Software Note: *Implementations may not support all defined modes of error detection or may provide implementation specific values. Setting CEDT to a value not supported by the implementation may cause no error detection to occur, or may produce undefined results.*
52	CSLC DCSLC	(Data) Cache snoop lock clear. <E.CL> Sticky bit set by hardware if a cache line lock was cleared by a snoop operation which caused an invalidation. Note that the lock for that line is cleared whenever the line is invalidated. This bit can be cleared only by software. 0 The cache has not encountered a snoop that invalidated a locked line. 1 The cache has encountered a snoop that invalidated a locked line.
53	CUL DCUL	(Data) Cache unable to lock. <E.CL> Sticky bit set by hardware. This bit can be cleared only by software. 0 Indicates a lock set instruction was effective in the cache 1 Indicates a lock set instruction was not effective in the cache
54	CLO DCLO	(Data) Cache lock overflow. <E.CL> Sticky bit set by hardware. This bit can be cleared only by software. 0 Indicates a lock overflow condition was not encountered in the cache 1 Indicates a lock overflow condition was encountered in the cache
55	CLFC DCLFC	(Data) Cache lock bits flash clear. <E.CL> Clearing occurs regardless of the enable (L1CSR0[CE]) value. 0 Default. 1 Hardware initiates a cache lock bits flash clear operation. Cleared when the operation is complete. **Note:** During a flash clear operation, writing a 1 causes undefined results; writing a 0 has no effect

Table 3-33. L1CSR0 Field Descriptions (continued)

Bits	Name	Description
56	CLOA DCLOA	(Data) Cache lock overflow allocate. <E.CL> Set by software to allow a lock request to replace a locked line when a lock overflow situation exists. Implementation of this bit is optional. 0 Indicates a lock overflow condition does not replace an existing locked line with the requested line 1 Indicates a lock overflow condition replaces an existing locked line with the requested line
57–58	CEIT	Cache error injection type. Controls the type of error injection to be performed. 00 Inject single bit data error 01 Inject single bit tag error 10 Inject double bit data error 11 Inject double bit tag error Not all processors will support this field. Consult the core reference manual. **Note:** How error injection is actually performed (for example, how the single or double bit error is created) is implementation specific.
60	DCBZ32	Data cache operation length. <DEO> Implementation of this field is optional. If not implemented, it should always read as 0. 0 **dcba** and **dcbz** (**dcbzep** <E.PD>) instruction number of bytes operated on is all bytes in cache line 1 **dcba** and **dcbz** (**dcbzep** <E.PD>) number of bytes operated on is 32
61	CABT DCABT	(Data) Cache operation aborted. 0 No cache operation completed improperly 1 Cache operation did not complete properly
62	CFI DCFI	(Data) Cache flash invalidate. Invalidation occurs regardless of the enable (L1CSR0[CE]) value. 0 No cache invalidate. 1 Cache flash invalidate operation. A cache invalidation operation is initiated by hardware. Once complete, this bit is cleared. **Note:** During an invalidation operation, writing a 1 causes undefined results; writing a 0 has no effect.
63	CE DCE	(Data) Cache enable. 0 The cache is not enabled. (not accessed or updated) 1 Enables cache operation. **Note:** CE (DCE) should not be set when the cache is disabled until after the cache has been properly initialized by flash invalidating the cache and the flash invalidate operation is complete. This applies both to the first time the cache is enabled as well as sequences that want to re-enable the cache after software has disabled it.

NOTE: Software Considerations

For processors which implement ECC for the L1 data cache, ECC is always enabled regardless of the setting of CECE. Corrected errors only cause asynchronous machine check exceptions when the count of errors exceed the threshold from L1CSR4. When an ECC error occurs, the hardware performs the following actions:

- perform the correction of the corrupted memory
- arrange for any in progress actions in the cache which may have encountered bad data to be re-executed after the data error is corrected.

This will generally require the instruction to be flushed and re-executed and snoops to be re-performed prior to a snoop response.

- place the calculated syndrome and stored syndrome into L1CSR4
- increment the count of corrected ECC errors in L1CSR4
- if the count of corrected ECC errors = the ECC corrected count threshold, and the threshold is a non-zero value, set MCSR[DCELIM] which will cause an asynchronous machine check

Software in the machine check handler should read the MCSR to determine the reason for the machine check. Upon noticing that MCSR[DCELIM] is set, software should read L1CSR4 to get information about the last error. Software should write L1CSR4 to reset the count of ECC corrections prior to clearing MCSR[DCELIM].

3.10.2 L1 Cache Control and Status Register 1 (L1CSR1)

L1CSR1, shown in the following figure, is used for general control and status of the L1 instruction cache.

This register is hypervisor privileged.

Writing this register requires synchronization. See Section 4.5.4.3, “Synchronization Requirements.”

SPR 1011 Hypervisor

	32 43	44 45	46	47	48	49	50 51	52	53	54	55	56	57 60	61	62	63
R W	—	ICEA	—	ICECE	ICEI	—	ICEDT	ICSLC	ICUL	ICLO	ICLFC	ICLOA	—	ICABT	ICFI	ICE
Reset	All zeros															

Figure 3-42. L1 Cache Control and Status Register 1 (L1CSR1)

This table describes L1CSR1 fields.

Table 3-34. L1CSR1 Field Descriptions

Bits	Name	Description
32–43	—	Reserved, should be cleared.
44–45	ICEA	Instruction cache error action 00 Error detection causes an Error Report (or Machine Check). It is implementation dependent whether the error detection causes the instruction cache location containing the error to be invalidated. 01 Error detection causes invalidation of the instruction cache location containing the error. It is implementation dependent whether an Error Report (or Machine Check) occurs. 10 Error detection causes an error report (or machine check) and invalidation of the instruction cache location containing the error. 11 Reserved The setting of ICEA has no effect if L1CSR1[ICECE] = 0. Reading ICEA is not guaranteed to reflect the last written value in some implementations, however, it will return either the last written value or 0. **Note:** Some implementations may invalidate the cache line containing the error or the entire instruction cache when a error is detected. **Note:** Implementations may not support all defined modes of error action. Setting ICEA to a value not supported by the implementation may cause no error detection to occur or may produce undefined results.
46	—	Reserved for implementation dependent use.
47	ICECE	Instruction Cache Error Checking enable. 0 Error checking of the cache disabled 1 Error checking of the cache enabled
48	ICEI	Instruction cache error injection enable. 0 Error injection disabled 1 Error injection enabled. Note that cache error checking must also be enabled (L1CSR1[ICECE] = 1) when ICEI is set. If L1CSR1[ICECE] is not set the results are undefined and erratic behavior may occur. It is recommended that an attempt to set this bit when L1CSR1[ICECE] = 0 will cause the bit not to be set (that is, L1CSR1[ICEI] = L1CSR1[ICECE] & L1CSR1[ICEI]). Software Note: *When injection is enabled, accesses to the cache cause error conditions to be injected when an instruction is established in the cache. Different implementations may offer varying amounts of error injection. For example, some may inject errors into the particular location where an instruction is established, or some may also inject errors into tags and other data. Such methods are implementation specific, however, when injection is enabled, and a location in the cache is established, later accessing that location will cause an error to be present and detected if detection is enabled unless that location is flushed from the cache prior to access by another mechanism.*
49	—	Reserved, should be cleared.
50–51	ICEDT	Instruction Cache Error Detection Type 00 Parity error detection 01 EDC error detection 10 reserved 11 reserved Software Note: *Implementations may not support all defined modes of error detection. Setting ICEDT to a value not supported by the implementation may cause no error detection to occur, or may produce undefined results.*

Table 3-34. L1CSR1 Field Descriptions (continued)

Bits	Name	Description
52	ICSLC	Instruction cache snoop lock clear. <E.CL> Sticky bit set by hardware if a cache line lock was cleared by a snoop operation that caused an invalidation. Note that the lock for that line is cleared whenever the line is invalidated. This bit can be cleared only by software. 0 The cache has not encountered a snoop that invalidated a locked line. 1 The cache has encountered a snoop that invalidated a locked line.
53	ICUL	Instruction cache unable to lock. <E.CL> Sticky bit set by hardware and cleared only by software. 0 Indicates a lock set instruction was effective in the cache 1 Indicates a lock set instruction was not effective in the cache
54	ICLO	Cache line locking. Instruction cache lock overflow. <E.CL> Sticky bit set by hardware and cleared only by software. 0 Indicates a lock overflow condition was not encountered in the cache 1 Indicates a lock overflow condition was encountered in the cache
55	ICLFC	Instruction cache lock bits flash clear. <E.CL> Clearing occurs regardless of the enable (L1CSR1[ICE]) value. 0 Default. 1 Hardware initiates a cache lock bits flash clear operation. This bit is cleared when the operation is complete. **Note:** During a flash clear operation, writing a 1 causes undefined results; writing a 0 has no effect.
56	ICLOA	Instruction cache lock overflow no allocate. <E.CL> Set by software to prevent a lock request from replacing a locked line when a lock overflow situation exists. Implementation of this bit is optional. 0 A lock overflow condition replaces an existing locked line with the requested line 1 A lock overflow condition does not replace an existing locked line with a requested line
57–60	—	Reserved, should be cleared.
61	ICABT	Instruction cache operation aborted. 0 No cache operation completed improperly 1 Cache operation did not complete properly
62	ICFI	Instruction cache flash invalidate. Invalidation occurs regardless of the enable (L1CSR1[ICE]) value. 0 No cache invalidate. 1 Hardware initiated a cache invalidation. Once complete, ICFI is cleared. **Note:** During invalidation, writing a 1 causes undefined results; writing a 0 has no effect.
63	ICE	Instruction cache enable. 0 The cache is not enabled. (not accessed or updated) 1 Enables cache operation. **Note:** ICE should not be set when the cache is disabled until after the cache has been properly initialized by flash invalidating the cache and the flash invalidate operation is complete. This applies both to the first time the cache is enabled as well as sequences that want to re-enable the cache after software has disabled it.

3.10.3 L1 Cache Control and Status Register 2 (L1CSR2)

L1CSR2, shown in the following figure, provides additional control and status for the L1 data cache. If a processor implements a unified cache, L1CSR2 applies to the unified cache. Some processors do not implement this register or certain fields in this register.

This register is hypervisor privileged.

Writing this register requires synchronization. See Section 4.5.4.3, “Synchronization Requirements.”

SPR 606 Hypervisor

	32	33	34 53	54 63
R W	—	DCWS	—	DCSTASHID
Reset	All zeros			

Figure 3-43. L1 Cache Control and Status Register 2 (L1CSR2)

This table describes the L1CSR2 fields.

Table 3-35. L1CSR2 Field Descriptions

Bits	Name	Description
32	—	Reserved, should be cleared.
33	DCWS	Data cache write shadow. <WS> Set by software to place the primary data cache into write-shadow mode. If write-shadow mode is enabled, data written to the L1 data cache is also written to backing store so subsequent failures in the L1 data cache can be recovered from by invalidating the data cache. Not all processors implement this field. 0 The primary data cache is not in write-shadow mode. 1 The primary data cache is in write-shadow mode. **Note:** To prevent loss of modified data, software should flush and invalidate the primary data cache before setting DCWS. Some processors may force the invalidation of the primary cache data cache when DCWS is set.
34–53	—	Reserved, should be cleared.
54–63	DCSTASHID	Data cache stash ID. <CS> Contains the cache target identifier to be used for external stash operations directed to this processor’s data cache. Not all processors implement this field, nor do all processors that implement this field implement all 10 bits. Processors that do implement this field implement all 10 bits or implement a subset of the low-order bits. Cache target identifiers are implementation specific, but must always be greater than 7. 0 The L1 cache does not accept external stash operations. Other values indicate the cache target identifier. **Note:** Cache stashing is an operation provided by the integrated device to which the processor’s caches are possible targets. Cache stashing preloads the cache without direct reference from software running on the processor.

3.10.4 L1 Cache Control and Status Register 3 (L1CSR3)

L1CSR3, shown in the following figure, provides additional control and status for the primary (L1) instruction cache of the processor. The most significant bits of this register may be used to provide implementation specific functions. Not all processors implement this register.

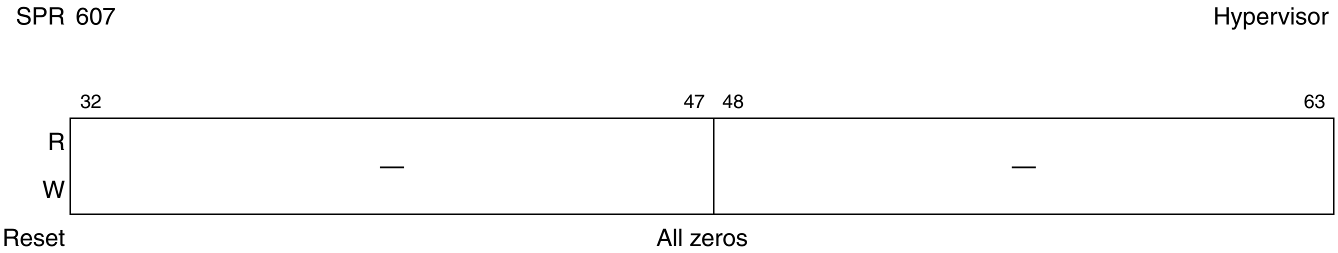


Figure 3-44. L1 Cache Control and Status Register 3 (L1CSR3)

This table describes the L1CSR3 fields.

Table 3-36. L1CSR3 Field Descriptions

Bits	Name	Description
32–47	—	Implementation dependent.
48–63	—	Reserved, should be cleared.

3.10.5 L1 Cache Control and Status Register 4 (L1CSR4)

L1CSR4, shown in Figure 3-45, provides additional error control and status for the primary (L1) data cache of the processor. Not all processors implement this register. L1CSR4 is used to record hardware corrected errors and provide a method for software notification when recorded corrected errors meet a specified threshold.

Each time an ECC error is corrected by hardware in the L1 data cache, L1CSR4[L1ECC_CNT] (count) is incremented, and the calculated syndrome and the stored syndrome which miscompared are stored in L1CSR4[L1ECC_CSYN] and L1CSR4[L1ECC_SSYN] (syndromes). If L1CSR4[L1ECC_THRESH] (threshold) is not zero and threshold is equal to count, an asynchronous machine check is generated. The bit(s) set in MCSR associated with the machine check are implementation dependent.

In the machine check interrupt handler, software should read the contents of L1CSR4 and write a new value setting the threshold to the desired value and clear the count and syndromes. Failure to clear the count may cause the asynchronous machine check to be continually asserted if the count and the threshold are equal and not zero.

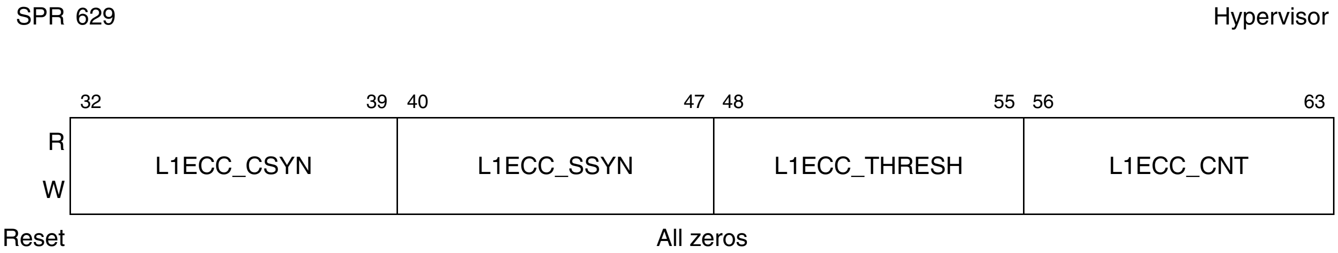


Figure 3-45. L1 Cache Control and Status Register 4 (L1CSR4)

This table describes the L1CSR4 fields.

Table 3-37. L1CSR4 Field Descriptions

Bits	Name	Description
32–39	L1ECC_CSYN	Calculated syndrome from corrected ECC error in the L1 data cache for most recent corrected error.
40–47	L1ECC_SSYN	Stored syndrome from corrected ECC error in the L1 data cache for most recent corrected error.
48–55	L1ECC_THRESH	The threshold at which to report corrected ECC errors in the data cache. When L1ECC_THRESH = 0, corrected ECC errors in the L1 data cache are not reported. When L1ECC_THRESH != 0, corrected ECC errors in the L1 data cache are reported when the corrected ECC error count (L1ECC_CNT) is equal to L1ECC_THRESH. An error is signaled in MCSR, which causes an asynchronous machine check interrupt.
56–63	L1ECC_CNT	The count of corrected ECC errors in the data cache. When L1ECC_THRESH = 0, corrected ECC errors in the L1 data cache are not reported. When L1ECC_THRESH != 0, corrected ECC errors in the L1 data cache are reported when the corrected ECC error count (L1ECC_CNT) is equal to L1ECC_THRESH. An error is signaled in MCSR, which causes an asynchronous machine check interrupt.

3.10.6 L1 Cache Configuration Register 0 (L1CFG0)

L1CFG0, shown in Figure 3-46, provides configuration information for the L1 data cache. If a processor implements a unified cache, L1CFG0 applies to the unified cache.

SPR 515 User RO

	32 33	34	35	36	37 38	39 40	41 42	43	44	45 52	53 63
R	CARCH	CWPA	CFAHA	CFISWA	—	CBSIZE	CREPL	CLA	CPA	CNWAY	CSIZE
W											
Reset	Implementation-dependent value										

Figure 3-46. L1 Cache Configuration Register 0 (L1CFG0)

This table describes L1CFG0 fields.

Table 3-38. L1CFG0 Field Descriptions

Bits	Name	Description
32–33	CARCH	Cache architecture 00 Harvard 01 Unified 10 Instruction only (L1CFG1 contains configuration information) 11 Reserved
34	CWPA	Cache way partitioning available. 0 Unavailable 1 Available
35	CFAHA	Cache flush all by hardware available 0 Unavailable 1 Available
36	CFISWA	Direct cache flush available. (Cache flush by set and way available.) 0 Unavailable 1 Available
37–38	—	Reserved, should be cleared.
39–40	CBSIZE	Cache line size 00 32 bytes 01 64 bytes 10 128 bytes 11 Reserved
41–42	CREPL	Cache replacement policy 00 True LRU 01 Pseudo LRU 10 Psuedo round robin 11 FIFO
43	CLA	Cache line locking available 0 Unavailable 1 Available
44	CPA	Cache parity available 0 Unavailable 1 Available
45–52	CNWAY	Cache number of ways minus 1.
53–63	CSIZE	Cache size in Kbytes.

3.10.7 L1 Cache Configuration Register 1 (L1CFG1)

L1CFG1, shown in the following figure, provides configuration information for the L1 instruction cache. If a processor implements a unified cache, L1CFG1 is not implemented.

SPR 516 User RO

	32 38	39 40	41 42	43	44	45 52	53 63
R	—	ICBSIZE	ICREPL	ICLA	ICPA	ICNWAY	ICSIZE
W							
Reset	Implementation-dependent value						

Figure 3-47. L1 Cache Configuration Register 1 (L1CFG1)

This table describes the L1CFG1 fields.

Table 3-39. L1CFG1 Field Descriptions

Bits	Name	Description
32–38	—	Reserved, should be cleared.
39–40	ICBSIZE	Instruction cache block size 00 32 bytes 01 64 bytes 10 128 bytes 11 Reserved
41–42	ICREPL	Cache replacement policy 00 True LRU 01 Pseudo LRU 10 Pseudo round robin 11 Reserved
43	ICLA	Cache line locking available 0 Unavailable 1 Available
44	ICPA	Cache parity available 0 Unavailable 1 Available
45–52	ICNWAY	Cache number of ways minus 1.
53–63	ICSIZE	Cache size in Kbytes.

3.10.8 L1 Flush and Invalidate Control Register 0 (L1FINV0) <DCF>

L1FINV0, shown in the following figure, allows the programmer to flush and/or invalidate the cache by specifying the cache set and cache way. See Section 6.3.2, "Direct Cache Flushing <DCF>," Tag matching is not required.

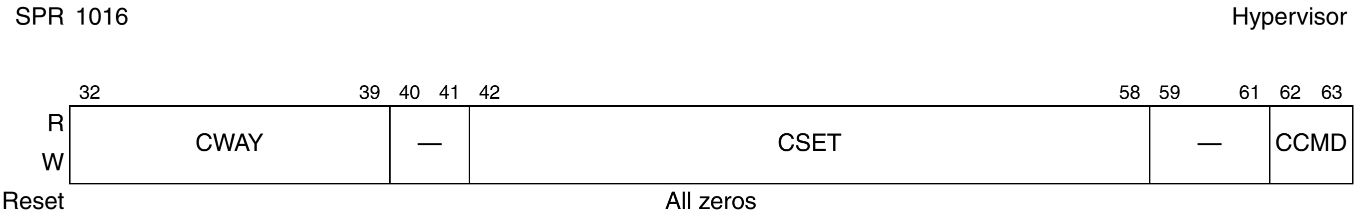


Figure 3-48. L1 Flush and Invalidate Control Register 0 (L1FINV0)

This table describes the L1FINV0 fields.

Table 3-40. L1FINV0 Fields—L1 Direct Cache Flush

Bits	Name	Descriptions
0–31	—	Reserved, should be cleared.
32–39	CWAY	Cache way. Specifies the cache way to be selected.
40–41	—	Reserved, should be cleared.
42–58	CSET	Cache set. Specifies the cache set to be selected.
59–61	—	Reserved, should be cleared.
62–63	CCMD	Cache flush command. 00 The line specified by CWAY and CSET is invalidated from the primary data cache without flushing. This is synonymous with a **dcbi** that references the specified line that has WIMGE = 0bxx0xx. The invalidation is local and occurs only in the primary data cache. 01 The line specified by CWAY and CSET is flushed if it is modified and valid. It is implementation dependent whether it remains in the cache, or is invalidated. For an implementation, the action performed on the line should be synonymous with a **dcbst** that references the same line that has WIMGE = 0bxx0xx. The invalidation is local and occurs only in the primary data cache. 01 If the line specified by CWAY and CSET is modified and valid, it is flushed and invalidated. For an implementation, the action performed on the line should be synonymous with a **dcbf** that references the same line that has WIMGE = 0bxx0xx. The invalidation is local and occurs only in the primary data cache. 11 Reserved for implementation use.

3.10.9 L1 Flush and Invalidate Control Register 1 (L1FINV1) <DCF>

L1FINV1, shown in Figure 3-48, allows the programmer to flush and/or invalidate the primary instruction cache by specifying the cache set and cache way. See Section 6.3.2, “Direct Cache Flushing <DCF>,” Tag matching is not required.

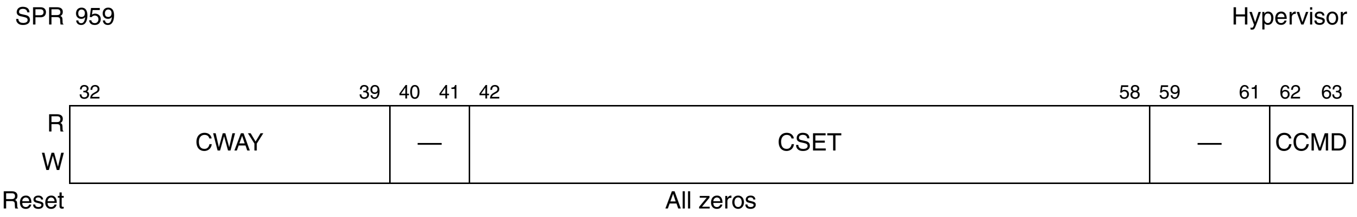


Figure 3-49. L1 Flush and Invalidate Control Register 0 (L1FINV0)

The L1FINV0 fields are described in this table.

Table 3-41. L1FINV0 Fields—L1 Direct Cache Flush

Bits	Name	Descriptions
0–31	—	Reserved, should be cleared.
32–39	CWAY	Cache way. Specifies the cache way to be selected.
40–41	—	Reserved, should be cleared.
42–58	CSET	Cache set. Specifies the cache set to be selected.
59–61	—	Reserved, should be cleared.
62–63	CCMD	Cache flush command. 00 The line specified by CWAY and CSET is invalidated. The invalidation is not equivalent to an **icbi** to the line as the invalidation is local. 01 Reserved. 10 Reserved. 11 Reserved for implementation use.

3.11 L2 Cache Registers

L2 cache registers provide control, configuration, and status for an L2 cache implemented on the core. If the core does not implement an L2 cache, these registers are generally not implemented. Some cores will implement these registers, which will always read as zero, which indicates that no L2 cache is present.

Some integrated devices implement L2 cache registers as SPRs and some implement them as MMRs. In general, MMRs are used to access the L2 cache registers when the L2 cache is shared among multiple processor cores. See the core reference manual.

3.11.1 L2 Configuration Register (L2CFG0)

L2CFG0, shown in Figure 3-50, allows software to identify the organization and capabilities of the secondary cache. The secondary cache is considered to be a private integrated backside L2 or a shared backside L2. Frontside caches are not part of EIS.

SPR 519 User RO
MMR block offset: 0x008

	32	33–34	35–36	37	38–40	41–42	43	44	45–47
R	—	L2CTEHA	L2CDEHA	L2CIDPA	L2CBSIZE	L2CREPL	L2CLA	—	L2CNWAY
W									
Reset	All zeros								

	48–49	50–63
R	L2CNWAY	L2CSIZE
W		
Reset	All zeros	

Figure 3-50. L2 Cache Configuration Register (L2CFG0)

This table describes the L2CFG0 fields.

Table 3-42. L2CFG0 Field Descriptions

Bits	Name	Description
32	—	Reserved, should be cleared.
33–34	L2CTEHA	L2 cache tags error handling available. 00 None 01 Parity detection 10 1-bit ECC correction, 2-bit ECC detection 11 Reserved
35–36	L2CDEHA	L2 cache data error handling available. 00 None 01 Parity detection 10 Single bit ECC correction, 2 bit ECC detection 11 Reserved
37	L2CIDPA	Cache instruction and data partitioning available. 0 Unavailable 1 Available
38–40	L2CBSIZE	Cache line size. 000 32 bytes 001 64 bytes 010 128 bytes 011 256 bytes 1xx Reserved
41–42	L2CREPL	Cache default replacement policy. This is the default line replacement policy at power-on-reset. If an implementation allows software to change the replacement policy it is not reflected here. 00 True LRU 01 Pseudo LRU 10 Round robin 11 Reserved
43	L2CLA	Cache line locking available. 0 Unavailable 1 Available **Note:** Setting CT = 2 in a cache line locking (unlocking) instruction addresses the L2 cache.
44	—	Reserved, should be cleared.
45–49	L2CNWAY	Cache number of ways minus 1.
50–63	L2CSIZE	Cache size in 64 Kbytes. A value of 1 for L2CSIZE denotes a 64-Kbyte cache. A value of 2 denotes a 128-Kbyte cache, etc. A value of 0 in this field denotes that no L2 cache is present. **Note:** Some processor cores that do not contain L2 caches still contain the L2 SPRs defined by EIS but when read, always return 0. This is because some other processor cores in the same family do provide an L2 cache, and allowing these registers to be read prevents system software from attempting to read the L2CFG0 register and causing an illegal instruction exception.

3.11.2 L2 Cache Control and Status Register (L2CSR0)

L2CSR0, shown in Figure 3-51, provides general control and status for the L2 cache of the processor.

This register is hypervisor privileged.

Writing to this register requires synchronization.

SPR 1017 Hypervisor
MMR block offset: 0x000

	32	33	34	35–37	38–39	40–41	42	43	44–46	47
R W	L2E	L2PE	—	L2WP	L2CM	—	L2FI	L2IO	—	L2DO
Reset	All zeros									

	48–49	50–51	52	53	54–55	56	57	58	59–63
R W	—	L2REP	L2FL	L2LFC	—	L2LOA	—	L2LO	—
Reset	All zeros								

Figure 3-51. L2 Cache Control and Status Register (L2CSR0)

This table describes L2CSR0 fields.

Table 3-43. L2CSR0 Field Descriptions

Bits	Name	Description
32	L2E	L2 cache enable. Used to enable the L2 array. Enables or disables all storage of the L2 array regardless of how it is partitioned. 0 The L2 storage array is disabled and is not accessed for reads, writes, or any coherency protocols. 1 The L2 storage array is enabled. **Note:** The L2 cache array should be invalidated by setting L2CSR0[L2FI] before it is first enabled. Software Note: *The L2 cache array and locks should be invalidated by setting L2CSR0[L2FI] = 1, L2CSR0[L2LFC] = 1, and L2CSR0[L2LFCID] = 0b11 before it is first enabled. Failure to do so may result in false errors being detected and random lines in the locked state and unavailable for allocation.* Software Note: *If the L2 cache is enabled and software wishes to disable it by writing a 0 to L2E, software should first flush the L2 cache to ensure that any modified data resident in the L2 cache is pushed to memory. If the L2 cache is not flushed, coherency will be lost and any line in the cache may provide stale data when the L2 cache is later re-enabled.*
33	L2PE	L2 cache parity/ECC error checking enable. Used to enable error checking in the L2 array and tags. Enables or disables all error checking of the L2. 0 The L2 has error checking disabled. 1 The L2 has error checking enabled. L2PE should not bet set until after the L2 cache has been properly initialized after reset by performing flash invalidation (using L2FI). Doing so can cause erroneous detection of errors because the state of the error detection bits are random out of reset. L2PE should only be set when L2E is set in the same operation and the value of L2E was 0 prior to this operation. . When L2PE = 0, caches that employ Error Correction capabilities may silently provide correction for errors. If such errors are corrected, they are not reported if L2PE = 0. When errors are being injected into the L2 and L2PE = 0, it is undefined whether errors are detected or not. Software should only perform error injection with L2PE = 1. Software Note: *This is a master control for error checking. Other controls exist to select how errors are checked for the L2 array and the L2 tags.*
34	—	Reserved, should be cleared.
35–37	L2WP	L2 instruction/data way partitioning This field is used to specify instruction and data way partitioning. The method and contents of this field are implementation specific.
38–41	—	Reserved, should be cleared.
42	L2FI	L2 cache flash invalidate. Invalidation occurs regardless of the enable (L2CSR[L2E]) value. 0 No cache invalidate. 1 Cache flash invalidate operation. A cache invalidation operation is initiated by hardware. Once complete, this bit is cleared. All lines in the L2 array that are designated as L2 cache will be invalidated. Any lines that were previously locked with a cache line locking operation will lose their locks if the L2 cache does not implement persistent locks. **Note:** Writing a 1 during any invalidation operation causes undefined results. Writing a 0 during an invalidation operation is ignored.

Table 3-43. L2CSR0 Field Descriptions (continued)

Bits	Name	Description
43	L2IO	L2 cache instruction only. L2 cache lines are allocated only for instruction cache transactions. Setting this prevents lines from being allocated in the L2 cache from data accesses. 0 The L2 cache allocates lines for data accesses that miss in the L2 cache. 1 The L2 cache does not allocate lines for data accesses that miss in the L2 cache. Data accesses that hit in the L2 cache are unaffected by the setting of L2IO. Data accesses that do not hit and are not allocated due to the setting of L2IO are serviced by other parts of the memory hierarchy. After writing this bit, software should continue to read back the value until the desired set value is reflected. Software should not write any other L2 cache register until the operation has completed . **Note:** Setting both L2DO and L2IO effectively prevents any new lines from being allocated in the L2 cache, effectively locking the entire L2 cache.
44–46	—	Reserved, should be cleared.
47	L2DO	L2 cache data only. L2 cache lines are allocated only for data cache transactions. Setting this prevents lines from being allocated in the L2 cache from instruction fetches. 0 The L2 cache allocates lines for instruction fetches that miss in the L2 cache. 1 The L2 cache does not allocate lines for instruction fetches that miss in the L2 cache. Instruction fetches that hit in the L2 cache are unaffected by the setting of L2DO. Instruction fetches that do not hit and are not allocated due to the setting of L2DO are serviced by other parts of the memory hierarchy. After writing this bit, software should continue to read back the value until the desired set value is reflected. Software should not write any other L2 cache register until the operation has completed. **Note:** Setting both L2DO and L2IO prevents the allocation of any new lines in the L2 cache, effectively locking the entire L2 cache.
48–49	—	Reserved, should be cleared.
50-51	L2REP	L2 line replacement algorithm. 00 Implementation default replacement algorithm. (see the core reference manual) 01 Implementation dependent. 10 Implementation dependent. 11 Implementation dependent. Locks set by cache locking instructions are honored, regardless of the replacement algorithm.
52	L2FL	L2 cache flush. Used to initiate an L2 flush, causing all modified lines to be written out to main memory. 0 An L2 cache flush is not being performed. 1 Hardware initiates an L2 cache flush operation. This bit is cleared when the operation is complete. All lines in the L2 array that are modified are written out to main memory. L2FL should not be set unless the L2 cache is currently enabled (for example, when L2E is not already set). **Note:** Writing a 1 while a flush operation is in progress causes undefined results. Writing a 0 during a flush operation is ignored. Writing a 1 to L2FL produces undefined results if the L2 cache is not enabled (for example, if L2CSR[L2E] =0).

Table 3-43. L2CSR0 Field Descriptions (continued)

Bits	Name	Description
53	L2LFC	L2 cache lock flash clear. This action clears locks on lines in the L2 cache. Depending on the setting of L2CSR[L2LFCID], all locks are cleared for data, instruction, or both. The contents of the cache are not affected. 0 Default. 1 Hardware initiates an L2 cache lock flash clear operation. This bit is cleared when the operation is complete. **Note:** Writing a 1 while a flash clear operation is in progress causes undefined results. Writing a 0 during a flash clear operation is ignored. Writing a 1 to L2LFC produces undefined results if the L2 cache is not enabled (L2CSR[L2E] =0). Software Note: On boot, the processor should set this bit to clear any lock state bits which may be randomly set out of reset.
54–55	L2FCID	L2 cache lock flash clear instruction or data. The setting of this bit determines which lines have locks cleared if L2CSR[L2LFC] is set. 00 No locks are cleared. 01 Clear locks on lines that contain data. 10 Clear locks on lines that contain instructions. 11 Clear locks on all lines. **Note:** Writing to L2FCID while a lock flash clear operation is in progress (L2CSR[L2LFC] = 1) causes undefined results.
56	L2LOA	L2 cache lock overflow allocate. Set by software to allow a lock request to replace a locked line when a lock overflow situation exists. 0 Indicates a lock overflow condition does not replace an existing locked line with the requested line 1 Indicates a lock overflow condition does replace an existing locked line with the requested line
57	—	Reserved, should be cleared.
58	L2LO	L2 cache lock overflow. Sticky bit set by hardware if a cache line lock overflow condition was detected by the L2 cache. Overflow conditions can occur as the result of executing a touch and lock set instruction that targets the L2 cache and resulting lock causes another locked line to be evicted. L2LO remains set until cleared by software. 0 The L2 cache has not encountered a lock overflow condition. 1 The L2 cache has encountered a lock overflow condition.
59–63	—	Reserved, should be cleared.

3.11.3 L2 Cache Control and Status Register 1 (L2CSR1)

L2CSR1, shown in Figure 3-52, provides general control and status for the L2 cache of the processor.

This register is hypervisor privileged.

Writing to this register requires synchronization.

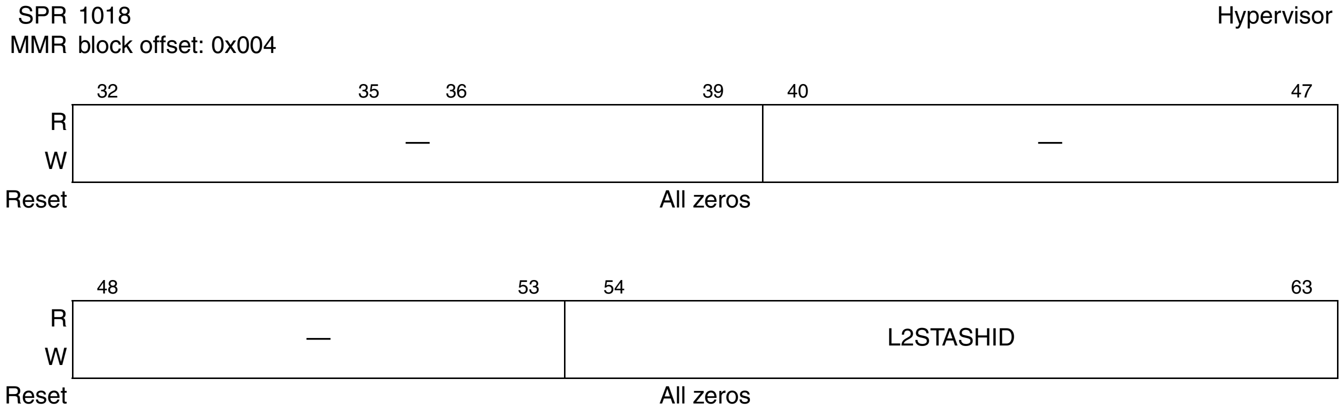


Figure 3-52. L2 Cache Control and Status Register 1 (L2CSR1)

This table describes the L2CSR1 fields.

Table 3-44. L2CSR1 Field Descriptions

Bits	Name	Description
32–39	—	Implementation dependent.
40–53	—	Reserved, should be cleared.
54–63	L2STASHID	L2 cache stash ID. <CS> Contains the cache target identifier to be used for external stash operations that are directed to this processor's L2 cache. A value of 0 for L2STASHID prevents the L2 cache from accepting external stash operations. **Note:** Implementations may not implement all 10 bits and may support only a subset of possible values for those bits that are implemented.

3.11.4 L2 Error Registers

L2 cache error detection, reporting, and injection allow flexible handling of ECC and parity errors in the L2 data and tag arrays.

3.11.4.1 L2 Error Control and Reporting

Error detection is controlled by setting fields in L2ERRDIS, shown in Figure 3-53. Error detection is controlled separately for tags and data. Each of the fields is specified as a series of "disable" bits reflecting the default reset behavior of 0 values enabling all error detection. To prevent errors from being detected, software must set the appropriate L2ERRDIS bit to disable the appropriate error checking.

When an error is detected in the L2 cache, the appropriate L2ERRDET bit is set and an implementation specific interrupt is generated for software to notice and handle the error. The generation of the interrupt when an error is detected is controlled by the L2 (L2ERRINTEN). Bits in this register control whether an interrupt is to be generated based on the type and location of the error in the L2 cache. Software may read from the L2ERRDET and may clear bits by writing a 1 into a bit position, but cannot directly write the contents of the register. L2ERRDET is shown in Figure 3-54; L2ERRINTEN is shown in Figure 3-55.

Single-bit errors that are corrected via ECC are not reported unless the appropriate L2ERRDET bit is set and the number of errors reach a threshold specified in L2ERRCTL. This prevents normal correctable ECC errors from generating error reports, but still causes errors to be generated when there may be a faulty condition in the L2 cache. L2ERRCTL is shown in Figure 3-56. If the error detection method is changed between ECC and parity, the cache must be flushed to ensure proper operation.

3.11.4.2 Error Capture

When an error is detected and reported, information about the error is posted into several L2 cache error capture registers (L2ERRADDR, L2ERREADDR, L2ERRATTR, L2CAPTDATAHI, L2CAPTDATALO, and L2CAPTECC). Only the first reported error information is saved. After software has examined the error information, it should clear L2ERRATTR[VALINFO] to allow the next detected error to have information posted.

L2ERRATTR, shown in Figure 3-57, contains implementation specific bits but should contain information about the transaction that caused the error. Software can clear L2ERRATTR[VALINFO] to enable the error capture registers to record information about the next error.

L2ERRADDR and L2ERREADDR, shown in Figure 3-58 and Figure 3-59, contain the physical address associated with the captured error. Processors that do not support more than 32 bits of physical address are not required to implement L2ERREADDR.

L2CAPTDATAHI and L2CAPTDATALO, shown in Figure 3-60 and Figure 3-61, contain data associated with the captured error. The contents of these registers is implementation specific. .

L2CAPTECC, shown in Figure 3-62, contains ECC information (the syndrome and the checksum) if the captured error was an ECC error.

3.11.4.3 Error Injection

The L2 cache includes support for injecting errors into the L2 data, data ECC, or tag. This may be used to test error recovery software by deterministically creating error scenarios.

The preferred method for error injection is to set all data pages (except a scratch page) to cache inhibited, set L2CSR[L2DO] to prevent allocation of instruction accesses, and invalidate the L2 by setting L2CSR[L2FI]. The following code triggers an error, then detects it (assume that **r**3 contains an address in a cached scratch page):

```
dcbz    r3              // allocates the line in the L1 in the modified state
dcbtls  2,0,r3          // forces the line from the L1 and allocates the line in the L2
lwz     r4,0(r3)        // touching the line causes an error if error injection is set
                        // up
```

Data or tag errors are injected into the line, according to the error injection settings in L2ERRINJHI, L2ERRINJLO, and L2ERRINJCTL, at allocation. The final load detects and reports the error (if enabled) and allows software to examine the offending data, address, and attributes.

Note that software must clear the L2ERRINJCTL error injection enable bits and the L2 must be invalidated (by setting L2CSR[L2I]) before resuming L2 normal operation.

The contents of the L2ERRINJCTL (error injection control), the L2ERRINJHI (error injection mask high), and the L2ERRINJLO (error injection mask low) are implementation-dependent registers. It is recommended that L2ERRINJCTL be used to control the type of error injected and L2ERRINJLO and L2ERRINJHI be used to specify how the data is to be corrupted.

L2ERRINJCTL, L2ERRINJLO, and L2ERRINJHI are shown in Figure 3-63, Figure 3-64, and Figure 3-65.

3.11.4.4 L2 Cache Error Disable Register (L2ERRDIS)

L2ERRDIS, shown in Figure 3-53, provides error disable control for the L2 cache of the processor.

This register is hypervisor privileged.

Writing to this register requires synchronization.

SPR 725 Hypervisor
MMR block offset: 0xe44

	32 56	57	58	59	60	61	62	63
R W	—	TMBECCDIS	TSBECCDIS	TPARDIS	MBECCDIS	SBECCDIS	PARDIS	L2CFGDIS
Reset	All zeros							

Figure 3-53. L2 Cache Error Disable Register (L2ERRDIS)

This table describes the L2ERRDIS fields.

Table 3-45. L2ERRDIS Field Descriptions

Bits	Name	Description
32–56	—	Reserved, should be cleared.
57	TMBECCDIS	Tag Multiple-bit ECC error disable. Valid only if L2CFG0[L2CTEHA] = 0b10. 0 Tag Multiple-bit ECC error detection enabled. 1 Tag Multiple-bit ECC error detection disabled. **Note:** When error injection is being performed, TMBECCDIS (=0) and L2CSR0[L2PE] (=1) should always be configured to insure that errors are always detected. If they are not set when error injection is performed, the result is undefined.
58	TSBECCDIS	Tag ECC error disable. Valid only if L2CFG0[L2CTEHA] = 0b10. 0 Tag Single-bit ECC error detection enabled. 1 Tag Single-bit ECC error detection disabled. **Note:** When error injection is being performed, TSBECCDIS (=0) and L2CSR0[L2PE] (=1) should always be configured to insure that errors are always detected. If they are not set when error injection is performed, the result is undefined.
59	TPARDIS	Tag parity error disable. Valid only if L2CFG0[L2CTEHA] =0b01. 0 Tag parity error detection enabled. 1 Tag parity error detection disabled. **Note:** When error injection is being performed, TPARDIS (=0) and L2CSR0[L2PE] (=1) should always be configured to insure that errors are always detected. If they are not set when error injection is performed, the result is undefined.

Table 3-45. L2ERRDIS Field Descriptions (continued)

Bits	Name	Description
60	MBECCDIS	Data Multiple-bit ECC error disable. Valid only if L2CFG0[L2CDEHA] = 0b10. 0 Data Multiple-bit ECC error detection enabled. 1 Data Multiple-bit ECC error detection disabled. When error injection is being performed, MBECCDIS (=0) and L2CSR0[L2PE] (=1) should always be configured to insure that errors are always detected. If they are not set when error injection is performed, the result is undefined.
61	SBECCDIS	Data ECC error disable. Valid only if L2CFG0[L2CDEHA] = 10. 0 Data Single-bit ECC error detection enabled. 1 Data Single-bit ECC error detection disabled. When error injection is being performed, SBECCDIS (=0) and L2CSR0[L2PE] (=1) should always be configured to insure that errors are always detected. If they are not set when error injection is performed, the result is undefined.
62	PARDIS	Data parity error disable. Valid only if L2CFG0[L2CDEHA] = 0b01. 0 Data parity error detection enabled. 1 Data parity error detection disabled. When error injection is being performed, PARDIS (=0) and L2CSR0[L2PE] (=1) should always be configured to insure that errors are always detected. If they are not set when error injection is performed, the result is undefined.
63	L2CFGDIS	L2 configuration error disable 0 L2 configuration error detection enabled 1 L2 configuration error detection disabled

3.11.4.5 L2 Cache Error Detect Register (L2ERRDET)

L2ERRDET, shown in Figure 3-54, provides error detection information for the L2 cache of the processor.

This register is hypervisor privileged.

Writing to this register requires synchronization.

SPR 991 Hypervisor R/clear
MMR block offset: 0x0

	32–56	57	58	59	60	61	62	63
R W	—	TMBECCERR	TSBECCERR	TPARERR	MBECCERR	SBECCERR	PARERR	L2CFGERR
Reset	All zeros							

Figure 3-54. L2 Cache Error Detect Register (L2ERRDET)

This table describes the L2ERRDET fields.

Table 3-46. L2ERRDET Field Descriptions

Bits	Name	Description
32–56	—	Reserved, should be cleared.
57	TMBECCERR	Tag Multiple-bit ECC error detected. Writing a 1 to this bit location will reset the bit. Valid only if L2CFG0[L2CTEHA] = 0b10. 0 Tag Multiple-bit ECC error not detected. 1 Tag Multiple-bit ECC error detected.

Table 3-46. L2ERRDET Field Descriptions (continued)

Bits	Name	Description
58	TSBECCERR	Tag ECC error detected. Writing a 1 to this bit location will reset the bit. Valid only if L2CFG0[L2CTEHA] = 0b10. 0 Tag Single-bit ECC not detected. 1 Tag Single-bit ECC error detected.
59	TPARERR	Tag parity error detected. Writing a 1 to this bit location reset the bit. Valid only if L2CFG0[L2CTEHA] = 0b01. 0 Tag parity error not detected. 1 Tag parity error detected.
60	MBECCERR	Data Multiple-bit ECC error detected. Writing a 1 to this bit location resets the bit. Valid only if L2CFG0[L2CDEHA] = 0b10. 0 Multiple-bit data ECC error not detected. 1 Multiple-bit data ECC error detected.
61	SBECCERR	Data ECC error detected. Writing a 1 to this bit location resets the bit. Valid only if L2CFG0[L2CDEHA] = 0b10. 0 Single-bit data ECC error not detected. 1 Single-bit data ECC error detected.
62	PARERR	Data parity error detected. Writing a 1 to this bit location resets the bit. Valid only if L2CFG0[L2CDEHA] = 0b01. 0 Data parity error not detected. 1 Data parity error detected.
63	L2CFGERR	L2 configuration error detected. Writing a 1 to this bit location will reset the bit. 0 L2 configuration error not detected. 1 L2 configuration error detected.

3.11.4.6 L2 Cache Error Interrupt Enable Register (L2ERRINTEN)

L2ERRINTEN, shown in Figure 3-55, provides error interrupt control for the L2 cache of the processor.

This register is hypervisor privileged.

Writing to this register requires synchronization..

SPR 720 Hypervisor
MMR block offset: 0xe48

	32 56	57	58	59	60	61	62	63
R W	—	TMBECCINTEN	TSBECCINTEN	TPARINTEN	MBECCINTEN	SBECCINTEN	PARINTEN	L2CFGINTENR
Reset	All zeros							

Figure 3-55. L2 Cache Error Interrupt Enable Register (L2ERRINTEN)

This table describes L2ERRINTEN fields.

Table 3-47. L2ERRINTEN Field Descriptions

Bits	Name	Description
32–56	—	Reserved, should be cleared.
57	TMBECCINTEN	Tag Multiple-bit ECC error interrupt reporting enable. Valid only if L2CFG0[L2CTEHA] = 0b10. 0 Tag multiple-bit ECC error interrupt reporting disabled. 1 Tag multiple-bit ECC error interrupt reporting enabled.
58	TSBECCINTEN	Tag ECC interrupt reporting enable. Valid only if L2CFG0[L2CTEHA] = 0b10. 0 Tag single-bit ECC error interrupt reporting disabled. 1 Tag single-bit ECC error interrupt reporting enabled.
59	TPARINTEN	Tag parity error interrupt reporting enable. Valid only if L2CFG0[L2CTEHA] = 0b01. 0 Tag parity error interrupt reporting disabled. 1 Tag parity error interrupt reporting enabled through a machine check exception.
60	MBECCINTEN	Data multiple-bit ECC error interrupt reporting enable. Valid only if L2CFG0[L2CDEHA] = 0b10. 0 Data multiple-bit ECC error interrupt reporting disabled. 1 Data multiple-bit ECC error interrupt reporting enabled through a machine check exception.
61	SBECCINTEN	Data ECC error interrupt reporting enable. Valid only if L2CFG0[L2CDEHA] = 0b10. 0 Data single-bit ECC error interrupt reporting disabled. 1 Data single-bit ECC error interrupt reporting enabled through a machine check exception.
62	PARINTEN	Data parity error interrupt reporting enable. Valid only if L2CFG0[L2CDEHA] = 0b01. 0 Data parity error interrupt reporting disabled. 1 Data parity error interrupt reporting enabled through a machine check exception.
63	L2CFGINTEN	L2 configuration error interrupt reporting enable. 0 L2 configuration interrupt reporting disabled. 1 L2 configuration error interrupt reporting enabled.

3.11.4.7 L2 Cache Error Control Register (L2ERRCTL)

L2ERRCTL, shown in Figure 3-56, provides thresholds and counts for errors detected in the L2 cache of the processor.

This register is hypervisor privileged.

Writing to this register requires synchronization.

SPR 724 Hypervisor
MMR block offset: 0xe58

	32 … 39	40 … 47	48 … 55	56 … 63
R W	—	L2CTHRESH	L2TCCOUNT	L2CCOUNT
Reset	All zeros			

Figure 3-56. L2 Cache Error Control Register (L2ERRCTL)

This table describes the L2ERRCTL fields.

Table 3-48. L2ERRCTL Field Descriptions

Bits	Name	Description
32–39	—	Reserved, should be cleared.
40–47	L2CTHRESH	L2 cache threshold. Threshold value for the number of ECC single-bit errors that are detected before reporting an error condition. L2CTHRESH is compared to L2CCOUNT each time a single-bit ECC error is detected.
48–55	L2TCCOUNT	L2 tag ECC single-bit error count. Counts ECC single-bit errors in the L2 tags detected. If L2TCCOUNT equals the ECC single-bit error trigger threshold (L2CTHRESH), an error is reported if single-bit error reporting for tags is enabled. Software should clear this value when such an error is reported to reset the count.
56–63	L2CCOUNT	L2 data ECC single-bit error count. Counts ECC single-bit errors in the L2 data detected. If L2CCOUNT equals the ECC single-bit error trigger threshold (L2CTHRESH), an error is reported if single-bit error reporting for data is enabled. Software should clear this value when such an error is reported to reset the count.

3.11.4.8 L2 Cache Error Attribute Register (L2ERRATTR)

L2ERRATTR, shown in Figure 3-57, provides extended information for errors detected in the L2 cache of the processor.

This register is hypervisor privileged.

Writing to this register requires synchronization.

SPR 721 Hypervisor
MMR block offset: 0xe4c

	32 … 62	63
R W	—	VALINFO
Reset	All zeros	

Figure 3-57. L2 Cache Error Attribute Register (L2ERRATTR)

This table describes the L2ERRATTR fields.

Table 3-49. L2ERRATTR Field Descriptions

Bits	Name	Description
32–62	—	Implementation specific fields describing transaction type, size, etc. for captured error. See the core reference manual.
63	VALINFO	L2 capture registers valid. 0 L2 capture registers contain no valid information or no enabled errors were detected. 1 L2 capture registers contain information of the first detected error which has reporting enabled. Software must clear this bit to unfreeze error capture so error detection hardware can overwrite the capture address/data/attributes for a newly detected error.

3.11.4.9 L2 Cache Error Address Capture Registers (L2ERRADDR, L2ERREADDR)

L2ERRADDR, shown in Figure 3-58, provides the low order bits of the real address of a captured error detected in the L2 cache of the processor. L2ERREADDR, shown in Figure 3-59, provides the high order bits of the real address of a captured error detected in the L2 cache of the processor.

These registers are hypervisor privileged.

Writing to these registers requires synchronization.

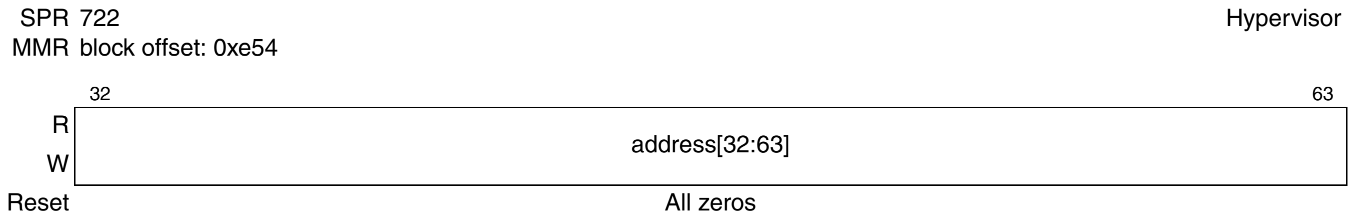


Figure 3-58. L2 Cache Error Address Capture Register (L2ERRADDR)

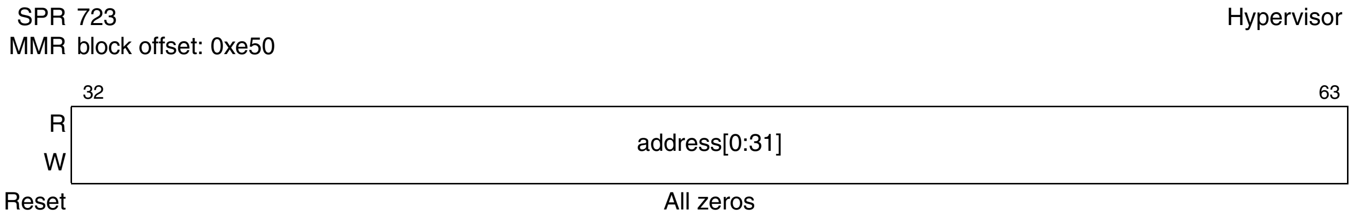


Figure 3-59. L2 Cache Error Extended Address Capture Register (L2ERREADDR)

3.11.4.10 L2 Cache Error Capture Data Registers (L2ECAPTDATALO, L2ECAPTDATAHI)

L2ECAPTDATALO, shown in Figure 3-60, provides the low order bits of implementation specific data of a captured error detected in the L2 cache of the processor. L2ECAPTDATAHI, shown in Figure 3-61, provides the high order bits of implementation specific data of a captured error detected in the L2 cache of the processor.

These registers are hypervisor privileged.

Writing to these registers requires synchronization.

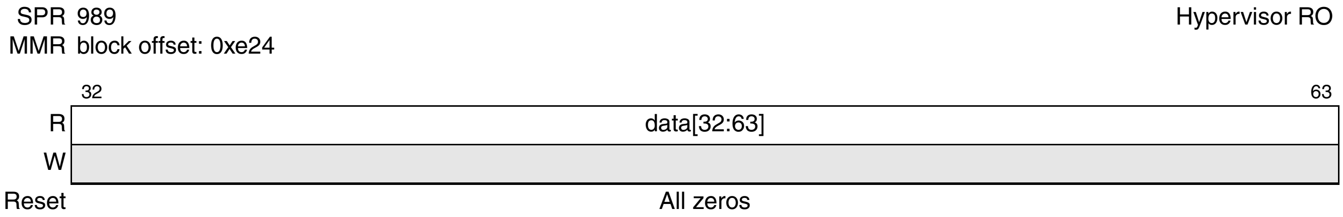


Figure 3-60. L2 Cache Error Capture Data Low Register (L2CAPTDATALO)

SPR 988 Hypervisor RO
MMR block offset: 0xe20

	32–63
R	data[0:31]
W	
Reset	All zeros

Figure 3-61. L2 Cache Error Capture Data High Register (L2CAPTDATAHI)

3.11.4.11 L2 Cache Error Capture ECC Syndrome Register (L2CAPTECC)

L2CAPTECC, shown in Figure 3-62, provides the calculated and stored ECC syndrome of a captured error detected in the L2 cache of the processor.

This register is hypervisor privileged.

Writing to this register requires synchronization.

SPR 990 Hypervisor RO
MMR block offset: 0xe28

	32–39	40–55	56–63
R	ECCSYND	—	ECCCHKSUM
W			
Reset	All zeros		

Figure 3-62. L2 Cache Error Capture ECC Syndrome Register (L2CAPTECC)

This table describes the L2CAPTECC fields.

Table 3-50. L2CAPTECC Field Descriptions

Bits	Name	Description
32–39	ECCSYND	The calculated ECC syndrome of the captured error.
40–55	—	Reserved, should be 0.
56–63	ECCCHKSUM	The stored ECC of the captured error.

3.11.4.12 L2 Cache Error Injection Control Register (L2ERRINJCTL)

L2ERRINJCTL, shown in Figure 3-63, provides controls for injecting errors into the L2 cache of the processor. The method for specifying and performing the injection is implementation specific, but implementations should provide ways to inject errors into both the tags and the data arrays of the L2 cache.

This register is hypervisor privileged.

Writing to this register requires synchronization.

NOTE: Software Considerations

When error injection is being performed, software must ensure that L2PE is already set and individual error disables are clear when performing injection to the L2 cache. This allows the L2 cache to be properly configured to detect the errors. If detection is not enabled, results will be unpredictable and undefined when injection is performed.

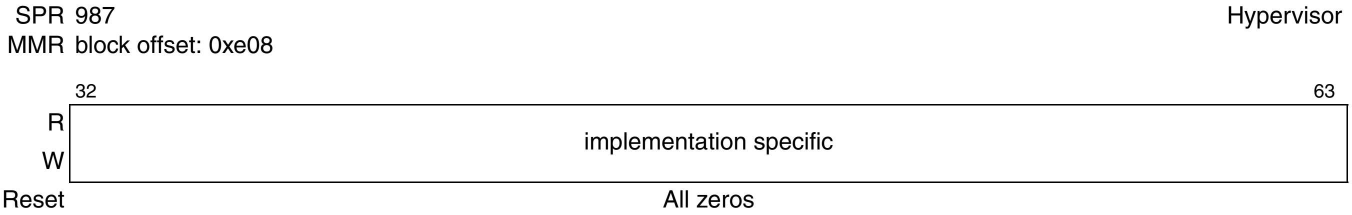


Figure 3-63. L2 Cache Error Injection Control Register (L2ERRINJCTL)

3.11.4.13 L2 Cache Error Injection Mask Low and High Registers (L2ERRINJLO, L2ERRINJHI)

L2ERRINJLO, shown in Figure 3-64, and L2ERRINJHI, shown in Figure 3-65, provide the injection mask describing how errors are to be injected into the L2 cache of the processor. The format of the injection mask is implementation specific.

These registers are hypervisor privileged.

Writing to these registers requires synchronization.

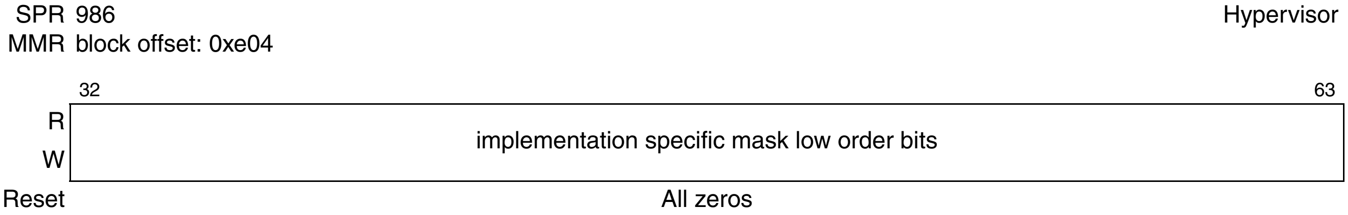


Figure 3-64. L2 Cache Error Injection Mask Low Register (L2ERRINJLO)

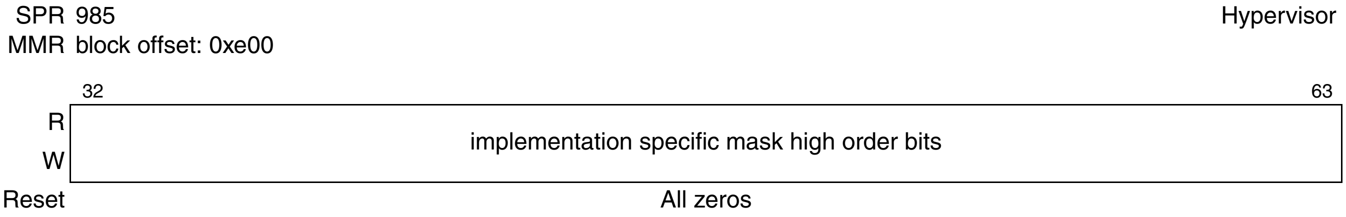


Figure 3-65. L2 Cache Error Injection Mask High Register (L2ERRINJHI)

3.12 MMU Registers

This section describes the following MMU registers and their fields:

- Logical partition ID register (LPIDR) <E.HV>
- Process ID registers (PID*n*)
- MMU control and status register 0 (MMUCSR0)
- MMU configuration register (MMUCFG)
- TLB configuration registers (TLB*n*CFG)
- MMU assist registers (MAS*n*)

3.12.1 Logical Partition ID Register (LPIDR) <E.HV>

The LPIDR register, shown in Figure 3-66, contains the logical partition ID currently in use for the processor. The logical partition ID is part of the virtual address and is used during address translation comparing LPIDR to the TLPID field in the TLB entry to determine a matching TLB entry.

An implementation may choose to implement fewer bits than the architectural definition of LPIDR. Non-implemented bits should read as zero. The number of bits implemented should be reflected in MMUCFG[LPIDSIZE].

This register is hypervisor privileged.

Writing to this register requires synchronization.

SPR 338 Hypervisor

	32 51	52 63
R W	—	Logical Partition ID
Reset	All zeros	

Figure 3-66. Logical Partition ID Register (LPIDR)

3.12.2 Process ID Register (PID)

The PID register, shown in Figure 3-67, contains the process ID currently in use for the processor. Process ID values are normally assigned by the operating system and are used to provide distinct address spaces for processes. The process ID is part of the virtual address and is used during address translation comparing PID to the TID field in the TLB entry to determine a matching TLB entry.

An implementation may choose to implement fewer bits than the architectural definition of PID. Non-implemented bits should read as zero. The number of bits implemented should be reflected in MMUCFG[PIDSIZE].

Some Freescale devices implement multiple PID registers. The number implemented is indicated by the value of MMUCFG[NPIDS]. PID1 and PID2 are shown as part of Figure 3-66.

Freescale devices may implement multiple PID registers. The number implemented is indicated by the value of MMUCFG[NPIDS]. PID1 and PID2 are shown as part of Figure 3-67. Processors that implement more than one PID may describe PID as PID0. Both refer to the same register and SPR number. Multiple PID registers is being phased out of the architecture. Newer processor implementations should contain only PID.

These registers are hypervisor privileged.

Writing to these registers requires synchronization.

NOTE: Software Considerations

> One processors that implement multiple PID registers, it is suggested PID (PID0) denote private mappings for a process and for other PID registers to handle mappings that may be common to multiple processes. This method allows for processes sharing address space to also share TLB entries if the shared space is mapped at the same virtual address and with the same permissions in each process.

SPR 48 (PID) (633 PID1; 634 PID2) — Guest supervisor

	32 49	50 63
R W	—	Process ID
Reset	All zeros	

Figure 3-67. Process ID Registers (PID, PID1, PID2)

3.12.3 MMU Control and Status Register 0 (MMUCSR0)

MMUCSR0 is used for general control of the L1 and L2 MMUs.

This register is hypervisor privileged.

Writing to this register requires synchronization.

The format of MMUCSR0 differs by MMU architecture version. The format for MMU architecture version 1 is shown in Figure 3-68.

SPR 1012 — Hypervisor

	32 40	41 44	45 48	49 52	53 56	57	58	59 60	61	62	63
R W	—	TLB3_PS	TLB2_PS	TLB1_PS	TLB0_PS	TLB2_FI	TLB3_FI	—	TLB0_FI	TLB1_FI	TLB_EI
Reset	All zeros										

Figure 3-68. MMU Control and Status Register 0 for MMU V1 (MMUCSR0)

This table describes the MMUCSR0 fields.

Table 3-51. MMUCSR0 Field Descriptions

Bits	Name	Description
32–40	—	Reserved, should be cleared.
41–56	TLB*n*_PS	TLB*n* array page size. A 4-bit field specifies the page size for TLB*n* array. Page size encoding is defined in Section 6.5.3, “TLB Concept.” For each TLB array *n*, the field is implemented only if TLB*n*CFG[AVAIL] = 0 and TLB*n*CFG[MINSIZE] ≠ TLB*n*CFG[MAXSIZE]. If the value of TLB*n*_PS is not in the range defined by TLB*n*CFG[MINSIZE,MAXSIZE] the page size is set to TLB*n*CFG[MINSIZE]. 41–44 TLB3_PS TLB3 array page size 45–48 TLB2_PS TLB2 array page size 49–52 TLB1_PS TLB1 array page size 53–56 TLB0_PS TLB0 array page size
57–62	TLB*n*_FI	TLB invalidate all bit for the TLB*n* array. 0 If this bit reads as a 1, an invalidate all operation for the TLBn array is in progress. Hardware will set this bit to 0 when the invalidate all operation is completed. Writing a 0 to this bit during an invalidate all operation is ignored. 1 TLB*n* invalidation operation. Hardware initiates a TLBn invalidate all operation. When this operation is complete, this bit is cleared. Writing a 1 during an invalidate all operation produces an undefined result. If the TLB array supports IPROT, entries that have IPROT set are not invalidated. 57 TLB2_FITLB2 invalidate-all bit for the TLB2 array. 58 TLB3_FITLB3 Invalidate-all bit for the TLB3 array. 59–60Reserved 61 TLB0_FITLB0 invalidate-all bit for the TLB0 array. 62 TLB1_FITLB1 invalidate-all bit for the TLB1 array.
63	TLB_EI	TLB error injection enable. If set, any writes that occur to TLB entries in any array will inject errors if that array supports error detection. 0 TLB error injection is disabled (normal operation) 1 TLB error injection is enabled. Any writes to TLB arrays that support error detection have errors injected. If a TLB array does not support error detection, setting this bit will have no effect on any operations on that array.

3.12.4 MMU Configuration Register (MMUCFG)

MMUCFG, shown in Figure 3-69, gives configuration information about the implementation’s MMU.

This register is hypervisor privileged.

NOTE: Software Considerations

Since this register is hypervisor privileged, the hypervisor can emulate accesses to it and present virtual MMU capabilities to the guest.

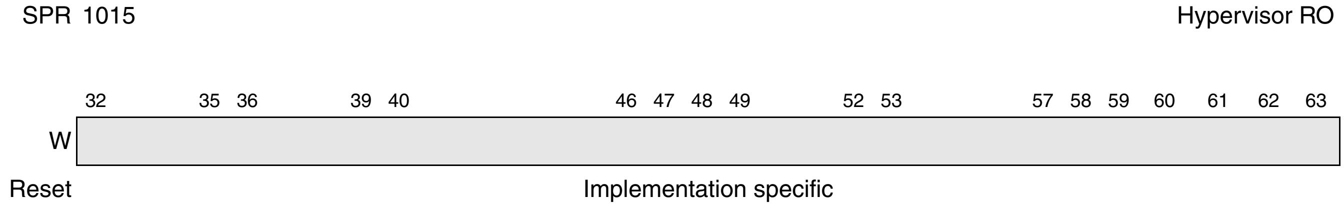


Figure 3-69. MMU Configuration Register (MMUCFG)

This table describes the MMUCFG fields.

Table 3-52. MMUCFG Field Descriptions

Bits	Name	Description
32–39	—	Reserved
36–39	LPIDSIZE	LPIDR size. <E.HV> Indicates the number of LPIDR bits implemented. The processor implements only the least significant bits in LPIDR . The maximum number of bits is 12. A value of 0 indicates that embedded hypervisor functionality is not present.
40–46	RASIZE	Real address size. Number of bits in a physical address supported by the implementation.
47–48	—	Reserved, should be cleared.
49–52	NPIDS	Number of PID registers. Indicates the number of PID registers supported by the implementation. Note: *Earlier versions of the architecture allowed for more than 1 PID register. Current implementations will always have this field set to 1*.
53–57	PIDSIZE	PID register size.The value of PIDSIZE is one less than the number of bits implemented for each of the PID registers implemented by the processor. The processor implements only the least significant PIDSIZE+1 bits in the PID registers. The maximum number of PID register bits that may be implemented is 14.
58–59	—	Reserved, should be cleared.
60–61	NTLBS	Number of TLBs. The value of NTLBS is one less than the number of software-accessible TLB structures that are implemented by the processor. NTLBS is set to one less than the number of TLB structures so that its value matches the maximum value of MAS0[TLBSEL]. 00 1 TLB 01 2 TLBs 10 3 TLBs 11 4 TLBs
62–63	MAVN	MMU architecture version number. Indicates the version number of the architecture of the MMU implemented by the processor. 00 Version 1.0 01 Version 2.0 10 Reserved 11 Reserved

3.12.5 TLB Configuration Registers (TLB*n*CFG)

TLB*n*CFG registers provide information about each TLB that is implemented. For example, TLB0CFG corresponds to TLB0. If a TLB array is not implemented, its associated TLB*n*CFG register is either not implemented, or the NENTRY field contains zero.

These registers are hypervisor privileged.

NOTE: Software Considerations

Since this register is hypervisor privileged, the hypervisor can emulate accesses to it and present virtual MMU capabilities to the guest.

The format of TLB*n*CFG is shown in Figure 3-70.

SPR 688 (TLB0CFG); 689 (TLB1CFG); 690 (TLB2CFG); 691 (TLB3CFG) Hypervisor RO

	32 39	40 43	44 47	48	49	50 51	52 63
R	ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	—	NENTRY
W							
Reset	Implementation-specific value						

Figure 3-70. TLB Configuration Register *n* (TLB0CFG)

This table describes the TLB*n*CFG fields. Values are implementation specific.

Table 3-53. TLB*n*CFG Field Descriptions

Bits	Name	Description
32–39	ASSOC	Associativity of TLB*n.* Number of ways of associativity of the TLB*n* array. A value of 0 or a value equal to NENTRY is fully associative.
40–43	MINSIZE	Minimum page size of TLB*n.* Page size encoding is defined by the TSIZE field of MAS1 in Section 3.12.6.2, “MAS Register 1 (MAS1).” 0001 Indicates smallest page size is 4 Kbytes 0002 Indicates smallest page size is 16 Kbytes …
44–47	MAXSIZE	Maximum page size of TLB*n.* Page size encoding is defined by the TSIZE field of MAS1 in Section 3.12.6.2, “MAS Register 1 (MAS1).” 0001 Indicates maximum page size is 4 Kbytes 0002 Indicates maximum page size is 16 Kbytes …
48	IPROT	Invalidate protect capability of TLB*n* array. 0 Indicates invalidate protection capability not supported. 1 Indicates invalidate protection capability supported.
49	AVAIL	Page size availability of TLB*n* array. 0 Fixed selectable page size from MINSIZE to MAXSIZE (all TLB entries are the same size). 1 Variable page size from MINSIZE to MAXSIZE (each TLB entry can be sized separately).
50	HES	Hardware entry select. Indicates that the TLB array supports MAS0[HES] where hardware will determine which TLB entry will be written based on MAS2[EPN] 0 The TLB array does not support hardware entry select 1 The TLB array supports hardware entry select
50–51	—	Reserved, should be cleared.
52–63	NENTRY	Number of entries in TLB*n.* A value of zero indicates that the TLB*n* array is not implemented.

3.12.6 MMU Assist Registers (MAS*n*)

MMU assist registers are used to manage pages and TLBs. Note that some fields are redefined by implementations.

The MMU Assist Registers (MAS) registers are used to transfer data to and from the TLB arrays. MAS registers can be read and written by software using **mfspr** and **mtspr** instructions. Execution of a **tlbre** instruction causes the TLB entry specified by MAS0[TLBSEL], MAS0[ESEL], and MAS2[EPN] to be copied to the MAS registers. Conversely, execution of a **tlbwe** instruction causes the TLB entry specified by MAS0[TLBSEL], MAS0[ESEL], and MAS2[EPN] to be written with contents of the MAS registers. MAS registers may also be updated by hardware on the occurrence of an Instruction or Data TLB Error Interrupt or as the result of a **tlbsx** instruction.

MAS5 and MAS8 are hypervisor privileged and are implemented only if category Embedded.Hypervisor is supported. All other MAS*n* registers are guest supervisor privileged. MAS7 is not required to be implemented if the processor supports 32 bits or less of real address.

3.12.6.1 MAS Register 0 (MAS0)

MAS0 is used for MMU read/write and replacement control.

This register is guest supervisor privileged.

Writing to this register requires synchronization.

The format of MAS0 is shown in Figure 3-71.

SPR 624 Guest supervisor

	32 33	34 35	36 47	48 51	52 63
R W	—	TLBSEL	ESEL	—	NV
Reset	All zeros				

Figure 3-71. MAS Register 0 (MAS0)

This table describes MAS0 fields.

Table 3-54. MAS0 Field Descriptions

Bits	Name	Comments or Function when Set
32–33	—	Reserved, should be cleared.
34–35	TLBSEL	Selects TLB for access. 00 TLB0 01 TLB1 10 TLB2 11 TLB3

Table 3-54. MAS0 Field Descriptions (continued)

Bits	Name	Comments or Function when Set
36–47	ESEL	Entry select. Identifies an entry in the selected array to be used for **tlbwe** and **tlbre**. Valid values for ESEL when accessing TLB entries are from 0 to TLB*n*CFG[ASSOC] - 1. That is, ESEL selects the way from a set of entries determined by MAS3[EPN]. For fully associative TLB arrays, ESEL ranges from 0 to TLB*n*CFG[NENTRY] - 1. ESEL is also updated on TLB error exceptions (misses) and **tlbsx** hit and miss cases.
48–51	—	Reserved, should be cleared.
52–63	NV	Next victim. Indicates the next victim to be targeted for a TLB miss replacement. If the TLB selected by MAS0[TLBSEL] does not support NV, this field is undefined. NV computation is implementation-dependent. NV is updated on TLB error exceptions (misses), **tlbsx** hit and miss cases, as shown in Table 6-11, and on execution of **tlbre** if the accessed TLB array supports NV. If NV is updated by a supported TLB array, NV always presents a value that can be used in MAS0[ESEL]. N

3.12.6.2 MAS Register 1 (MAS1)

Figure 3-72 describes the format of MAS1. The MAS1 register contains fields for selecting a TLB entry during translation.

This register is guest supervisor privileged.

Writing to this register requires synchronization.

The format of MAS1 is shown in Figure 3-72.

SPR 625 Guest supervisor

	32	33	34 47	48 50	51	52 55	56 63
R W	V	IPROT	TID	—	TS	TSIZE	—
Reset	All zeros						

Figure 3-72. MAS Register 1 (MAS1)

The MAS1 fields are described in this table.

Table 3-55. MAS1 Field Descriptions—Descriptor Context and Configuration Control

Bits	Name	Description
32	V	Valid bit. 0 This TLB entry is invalid. 1 This TLB entry is valid.
33	IPROT	Invalidate protect. Set to protect this TLB entry from invalidate operations due to execution of **tlbilx** <E.HV>, **tlbivax**, broadcast invalidations from another processor, or flash invalidations. Only implemented for TLB entries in TLB arrays where TLB*n*CFG[IPROT] is indicated. 0 Entry is not protected from invalidation 1 Entry is protected from invalidation.
34–35	—	Reserved, should be cleared.
36–47	TID	Translation identity. During translation, TID is compared with the current process IDs (PIDs) to select a TLB entry. A TID value of 0 defines an entry as global and matches with all process IDs.
48–50	—	Reserved, should be cleared.
51	TS	Translation space. During translation, TS is compared with AS (MSR[IS] or MSR[DS]) , depending on the type of access) to select a TLB entry.
52–55	TSIZE	Translation size. Defines the page size of the TLB entry. For TLB arrays that contain fixed-sized TLB entries, TSIZE is ignored. For variable page-size TLB arrays, the page size is 4^{TSIZE} Kbytes. TSIZE must be between TLB*n*CFG[MINSIZE] and TLB*n*CFG[MAXSIZE]. Valid TSIZE values for TLB page sizes are described in Section 6.5.3.2, "TLB Entry Page Size."
56–63	—	Reserved, should be cleared.

3.12.6.3 MAS Register 2 (MAS2)

MAS2, shown in Figure 3-73, contains fields for specifying the effective page address and the storage attributes for a TLB entry.

This register is guest supervisor privileged.

Writing to this register requires synchronization.

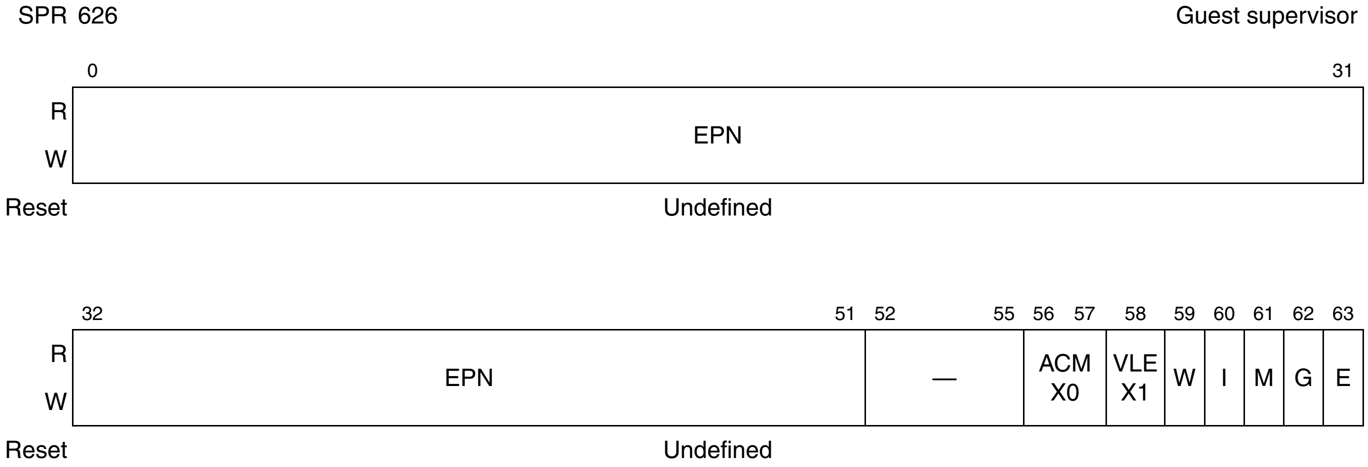


Figure 3-73. MAS Register 2 (MAS2)

The MAS2 fields are described in this table.

Table 3-56. MAS2 Field Descriptions—EPN and Page Attributes

Bits	Name	Description
0–51	EPN	Effective page number. Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be zero. EPN[0–31] are accessible only in 64-bit implementations as the upper 32 bits of the effective address of the page.
52–55	—	Reserved, should be cleared.
56–57	ACM X0	Alternate coherency mode. Allows an implementation to employ multiple coherency methods. If the M attribute (memory coherence required) is not set for a page (M=0), the page has no coherency associated with it and ACM is ignored. If the M = 1 attribute for a page, ACM determines the coherency domain (or protocol) used. ACM values are implementation dependent. **Note:** Some previous implementations may have a storage bit in the bit 57 position labeled as X0.
58	VLE X1	VLE mode.<VLE> Identifies pages that contain instructions from the VLE instruction set. This attribute is implemented only if the processor supports VLE. Setting both VLE and E is considered a programming error; an attempt to fetch instructions from a page so marked produces an ISI byte ordering exception and sets ESR[BO]. 0 Instructions fetched from the page are decoded and executed as instructions defined as non-category VLE instructions. 1 Instructions fetched from the page are decoded and executed as VLE (and associated extensions) instructions. **Note:** Some implementations have a bit in this position labeled as X1. Software should not use the presence of this bit (the ability to set to 1 and read a 1) to determine if the implementation supports the VLE extension.

Table 3-56. MAS2 Field Descriptions—EPN and Page Attributes (continued)

Bits	Name	Description
59	W	Write-through 0 This page is considered write-back with respect to the caches in the system. 1 All stores performed to this page are written through the caches to main memory.
60	I	Caching-inhibited 0 Accesses to this page are considered cacheable. 1 The page is considered caching-inhibited. All loads and stores to the page bypass the caches and are performed directly to main memory. A read or write to a caching-inhibited page affects only the memory element specified by the operation.
61	M	Memory coherence required 0 Memory coherence is not required. 1 Memory coherence is required. Loads and stores to this page are coherent with loads and stores from other processors (and devices) in the system, assuming all such devices participate in the coherence protocol.
62	G	Guarded 0 Accesses to this page are not guarded and can be performed before it is known if they are required by the sequential execution model. 1 Loads and stores to this page are performed without speculation (that is, they are known to be required).
63	E	Endianness. Determines endianness for the corresponding page. Little-endian operation is true little endian, which differs from the modified little-endian byte-ordering model optionally available in previous devices that implement the PowerPC architecture. 0 The page is accessed in big-endian byte order. 1 The page is accessed in true little-endian byte order.

3.12.6.4 MAS Register 3 (MAS3)

MAS3 contains fields for specifying the real page address and the permission attributes for a TLB entry.

This register is guest supervisor privileged.

Writing to this register requires synchronization.

The format of MAS3 is shown in Figure 3-74.

SPR 627 Guest supervisor

	32 51	52 53	54 57	58	59	60	61	62	63
R W	RPNL	—	U0–U3	UX	SX	UW	SW	UR	SR
Reset	All zeros								

Figure 3-74. MAS Register 3 (MAS3)

MAS3 fields are described in this table.

Table 3-57. MAS3 Field Descriptions–RPNL and Access Control

Bits	Name	Description
32–51	RPNL	Real page number bits 32–51. Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be zero. If the physical address space exceeds 32 bits, RPNU is accessed through MAS7.
52–53	—	Reserved, should be cleared.
54–57	U0–U3	User bits. Associated with a TLB entry and used by system software. For example, these bits may be used to hold information useful to a page scanning algorithm or be used to mark more abstract page attributes.
58–63	UX,SX UW,SW UR,SR	Permission bits (UX, SX, UW, SW, UR, SR). User and supervisor read, write, and execute permission bits. Effects of the permission bits are described in Section 6.5.6, “Permission Attributes.”

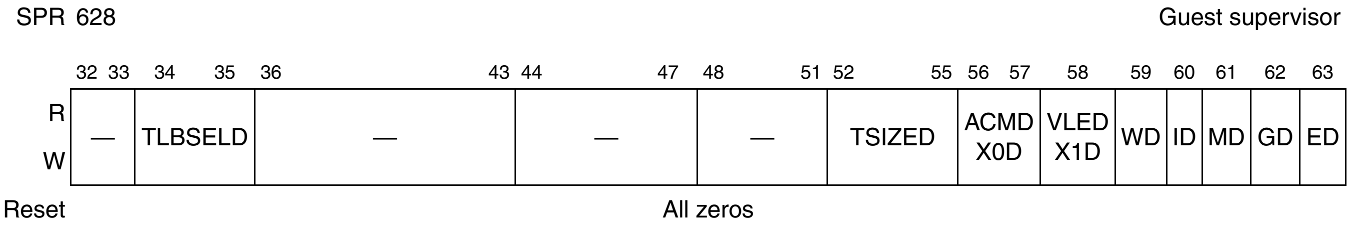
3.12.6.5 MAS Register 4 (MAS4)

MAS4 contains fields for specifying default information to be pre-loaded on certain MMU-related exceptions.

This register is guest supervisor privileged.

Writing to this register requires synchronization.

The format of MAS4 is shown in Figure 3-75.



SPR 628 Guest supervisor

	32 33	34 35	36 43	44 47	48 51	52 55	56 57	58	59	60	61	62	63
R W	—	TLBSELD	—	—	—	TSIZED	ACMD X0D	VLED X1D	WD	ID	MD	GD	ED
Reset	All zeros												

Figure 3-75. MAS Register 4 (MAS4)

The MAS4 fields are described in Figure 3-58.

Table 3-58. MAS4 Field Descriptions—Hardware Replacement Assist Configuration

Bits	Name	Description
32–33	—	Reserved, should be cleared.
34–35	TLBSELD	TLBSEL default value. Specifies the default value loaded in MAS0[TLBSEL] on a TLB miss exception. See Table 6-11.”
36–43	—	Reserved, should be cleared.
44–47	—	Reserved, should be cleared. Previous versions of EIS, implemented by some processors, defined this field as TIDSELD. TID default selection value. Specifies which of the current PID registers should be used to load MAS1[TID] on a TLB miss exception. PID registers are addressed as follows: 0000 = PID0 (PID) 0001 = PID1 ... 1110 = PID14 A value that references a non-implemented PID register causes a value of 0 to be placed in MAS1[TID]. See the implementations documentation for a list of supported PIDs.
48–51	—	Reserved, should be cleared.
52–55	TSIZED	Default TSIZE value. Specifies the default value loaded into MAS1[TSIZE] on a TLB miss exception.
57	ACMD	Default ACM value Specifies the default value loaded into MAS2[ACM] on a TLB miss exception. Only implemented if MAS2[ACM] is implemented.
58	VLED <VLE>	Default VLE value. . Specifies the default value loaded into MAS2[VLE] on a TLB miss exception.
59	WD	Default W value. Specifies the default value loaded into MAS2[W] on a TLB miss exception.
60	ID	Default I value. Specifies the default value loaded into MAS2[I] on a TLB miss exception.
61	MD	Default M value. Specifies the default value loaded into MAS2[M] on a TLB miss exception.
62	GD	Default G value. Specifies the default value loaded into MAS2[G] on a TLB miss exception.
63	ED	Default E value. Specifies the default value loaded into MAS2[E] on a TLB miss exception.

3.12.6.6 MAS Register 5 (MAS5) <E.HV>

MAS5, shown in Figure 3-76, contains hypervisor fields for specifying LPID and GS values to be used when searching TLB entries with **tlbsx** or specifying LPID values for invalidation with **tlbilx**. SLPID and SGS fields are used in place of LPIDR and MSR[GS] when comparing to TLPID and TGS fields in the TLB entry.

This register is hypervisor privileged.

Writing to this register requires synchronization.

SPR 339 Hypervisor

	32	33	51	52	63
R W	SGS	—		SLPID	
Reset	All zeros				

Figure 3-76. MAS Register 5 (MAS5)

The MAS5 fields are described in this table.

Table 3-59. MAS5 Field Descriptions—Hypervisor Search PIDs

Bits	Name	Description
32	SGS	Search GS. Specifies the GS value used when searching the TLB during execution of **tlbsx**. SGS is compared with the TGS field of each TLB entry to find a matching entry.
33–51	—	Reserved, should be cleared.
52–63	SLPID	Search LPID. Specifies the GS value used when searching the TLB during execution of **tlbsx** or specifying the LPID value for invalidation during execution of **tlbilx** . SLPID is compared with TLPID field of each TLB entry to find a matching entry. Only MMUCFG[LPIDSIZE] bits of this field are implemented.

3.12.6.7 MAS Register 6 (MAS6)

MAS6 specifies PID and AS values when using **tlbsx** to search TLB entries or specifying a PID value for invalidation using **tlbilx** <E.HV> to invalidate TLB entries.

This register is guest supervisor privileged.

Writing to this register requires synchronization.

The format of MAS6 is shown in Figure 3-77.

SPR 630 Guest supervisor

	32	33	34	47	48	62	63
R W	—		SPID (SPID0)		—		SAS
Reset	All zeros						

Figure 3-77. MAS Register 6 for MMU V1 (MAS6)

MAS6 fields are described in this table.

Table 3-60. MAS 6 Field Descriptions—Search PIDs and Search AS

Bits	Name	Description
32–33	—	Reserved, should be cleared.
34–47	SPID (SPID0)	Search PID. Specifies the PID value to be used when searching the TLB during execution of **tlbsx** or specifying the PID value for invalidation using **tlbilx** <E.HV>. SPID is compared to the TID field of each TLB entry to find a matching entry. Only MMUCFG[PIDSIZE] bits of this field are implemented.
48–62	—	Reserved, should be cleared.
63	SAS	Address space value for searches. Specifies the AS value used when executing **tlbsx** to search the TLB.

3.12.6.8 MAS Register 7 (MAS7)

MAS7, shown in Figure 3-78, is implemented only for processors that support more than 32 bits of physical address. It contains the high-order address bits of the RPN. Processors only implement the low-order bits required beyond 32 necessary to support the physical or logical addresses <E.HV> supported.

This register is guest supervisor privileged.

Writing to this register requires synchronization.

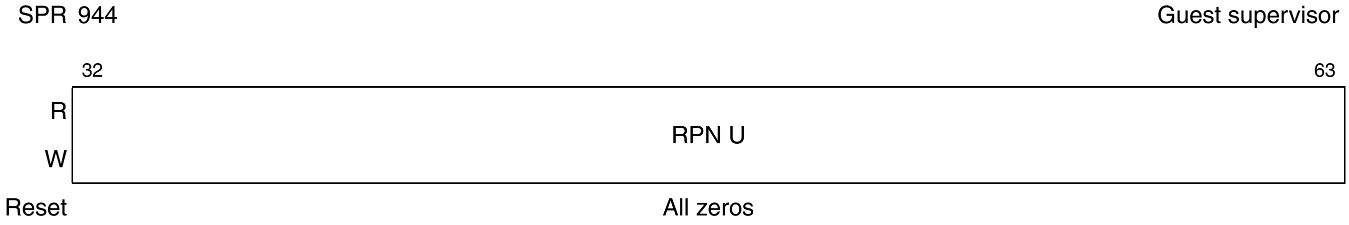


Figure 3-78. MAS Register 7 (MAS7)

The MAS7 fields are described in this table.

Table 3-61. MAS 7 Field Descriptions—High Order RPNU

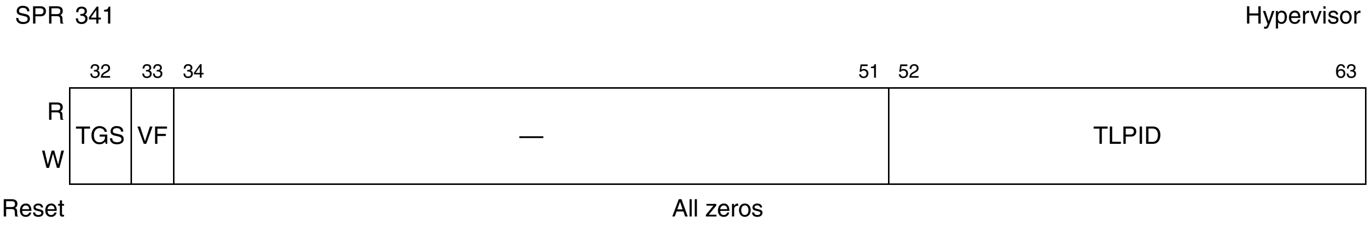
Bits	Name	Description
32–63	RPNU	Real page number (bits 0–31). RPNU is accessed through MAS3.

3.12.6.9 MAS Register 8 (MAS8) <E.HV>

MAS8, shown in Figure 3-79, contains hypervisor fields used for selecting a TLB entry during translation and for identifying translations for virtualization faults.

This register is hypervisor privileged.

Writing to this register requires synchronization.



SPR 341 Hypervisor

	32	33	34–51	52–63
R W	TGS	VF	—	TLPID
Reset	All zeros			

Figure 3-79. MAS Register 8 (MAS8)

The MAS8 fields are described in this table.

Table 3-62. MAS8 Field Descriptions

Bits	Name	Description
32	TGS	Translation guest space. During translation, TGS is compared with MSR[GS] to select a TLB entry.
33	VF	Virtualization fault. If set, a DSI occurs on data accesses to this page, regardless of the permission bit settings. 0 Data accesses translated by this TLB entry occur normally. 1 Data accesses translated by this TLB entry always cause a data storage interrupt **Note:** Hypervisor software should always deny execute access on pages marked with VF.
34–51	—	Reserved, should be cleared.
52–63	TLPID	Translation logical partition ID. During translation, Compared with the LPIDR register to select a TLB entry. A TLPID value of 0 defines an entry as global and matches all LPID values.Only MMUCFG[LPIDSIZE] bits of this field are implemented.

3.12.6.10 64-bit Access to MAS Register Pairs <64-bit, E.HV>

Certain MMU Assist registers can be accessed in pairs in single **mfspr** or **mtspr** instruction. The register pairs are listed in Table 3-63. Software should take special consideration when using MAS register pairs since the programming model is only available on 64-bit implementations. For **mtspr**, all 64 bits are written from the source GPR to the MAS pair and for **mfspr** all 64 bits are read from the MAS pair and are written to the GPR, regardless of computation mode. If compatibility with 32-bit implementations is desired, MAS register pair should not be used and the MAS registers should be addressed individually.

Table 3-63. MAS Register Pairs

Name	SPR Number	Privilege	Bits 0-31	Bits 32-63
MAS0_MAS1	373	Guest supervisor	MAS0	MAS1
MAS5_MAS6	348	Hypervisor	MAS5	MAS6
MAS7_MAS3	372	Guest supervisor	MAS7	MAS3
MAS8_MAS1	349	Hypervisor	MAS8	MAS1

3.12.7 External PID Registers <E.PD>

External PID load and store context registers, External PID Load Context (EPLC) and External PID Store Context (EPSC) are used to specify context for external load and store PID instructions. Fields in EPLC and EPSC replace values used in the normal address translation when external load and store PID instructions are executed. How translations are affected by external PID instructions is described in Section 6.5.4, “Address Translation.”

3.12.7.1 External PID Load Context Register (EPLC) <E.PD>

EPLC, shown in Figure 3-80, contains fields to provide the context for external PID instructions which perform load accesses.

This register is guest supervisor privileged. However, the EGS and ELPID may be modified only in hypervisor state. <E.HV>

Writing to this register requires synchronization.

SPR 947 Guest supervisor

	32	33	34	35	36 47	48 49	50 63
R W	EPR	EAS	EGS	—	ELPID	—	EPID
Reset	All zeros						

Figure 3-80. External PID Load Context (EPLC) Format

The EPLC fields are described in this table.

Table 3-64. EPLC Fields—External PID Load Context

Bits	Name	Descriptions
0–31	—	Reserved, should be cleared.
32	EPR	External load context PR bit. Used in place of MSR[PR] for load permission checking when an external PID load instruction executes. 0 Supervisor mode 1 User mode
33	EAS	External load context AS bit. Used in place of MSR[DS] for load translation when an external PID load instruction is executed. Compared with TLB[TS] during translation. 0 Address space 0 1 Address space 1
34	EGS	External load context GS bit. <E.HV> Used in place of MSR[GS] for load translation when an external PID load instruction is executed. Compared with TLB[TGS] during translation.This field is only writable in hypervisor state. A **mtspr** in guest supervisor state will leave this field unmodified. 0 Hypervisor address space 1 Guest address space

Table 3-64. EPLC Fields—External PID Load Context (continued)

Bits	Name	Descriptions
35	—	Reserved, should be cleared.
36–47	ELPID	External load context LPID value. <E.HV> Used in place of LPIDR value for load translation when an external PID load instruction is executed. Compared with TLB[TLPID] during translation. This field is only writable in hypervisor state. A **mtspr** in guest supervisor state will leave this field unmodified. Only MMUCFG[LPIDSIZE] bits of this field are implemented.
48–49	—	Reserved, should be cleared.
50–63	EPID	External load context PID value. Used in place of all PID register values for load translation when an external PID load instruction is executed. Compared with TLB[TID] during translation.Only MMUCFG[PIDSIZE] bits of this field are implemented.

3.12.7.2 External PID Store Context (EPSC) Register <E.PD>

EPLC, shown in Figure 3-81, contains fields to provide the context for external PID instructions which perform store accesses.

This register is guest supervisor privileged. However, the EGS and ELPID may be modified only in hypervisor state. <E.HV>

Writing to this register requires synchronization.

SPR 948 Guest supervisor

	32	33	34	35	36 … 47	48 49	50 … 63
R W	EPR	EAS	EGS	—	ELPID	—	EPID
Reset	All zeros						

Figure 3-81. External PID Store Context (EPSC) Format

The EPSC fields are described in this table.

Table 3-65. EPSC Fields—External PID Store Context

Bits	Name	Descriptions
0–31	—	Reserved, should be cleared.
32	EPR	External store context PR bit. Used in place of MSR[PR] for load permission checking when an external PID store instruction executes. 0 Supervisor mode 1 User mode
33	EAS	External store context AS bit. Used in place of MSR[DS] for store translation when an external PID store instruction is executed. Compared with TLB[TS] during translation. 0 Address space 0 1 Address space 1
34	EGS	External store context GS bit. <E.HV> Used in place of MSR[GS] for store translation when an external PID load instruction is executed. Compared with TLB[TGS] during translation.This field is only writable in hypervisor state. A **mtspr** in guest supervisor state will leave this field unmodified. 0 Hypervisor address space 1 Guest address space
35	—	Reserved, should be cleared.
36–47	ELPID	External store context LPID value. <E.HV> Used in place of LPIDR value for store translation when an external PID store instruction is executed. Compared with TLB[TLPID] during translation. This field is only writable in hypervisor state. A **mtspr** in guest supervisor state will leave this field unmodified. Only MMUCFG[LPIDSIZE] bits of this field are implemented.
48–49	—	Reserved, should be cleared.
50–63	EPID	External store context PID value. Used in place of all PID register values for store translation when an external PID store instruction is executed. Compared with TLB[TID] during translation.Only MMUCFG[PIDSIZE] bits of this field are implemented.

3.13 Debug Registers

This section describes debug-related registers that are accessible to software running on the processor. These registers are intended for use by special debug tools and debug software, and not by general application or operating system code.

The debug facility registers are described here appear in implementations in varying degrees. It is likely that some registers or fields within registers are implemented differently. Users should consult the Core Reference Manual for the implementation.

3.13.1 Internal and External Debug Facility Control Registers

Some of the internal debug facility registers can be modified, if affected by the external debug facility. Some processors use the value of DBCR0[EDM] to block all internal debug facilities from being used. Some processors use the DBRR0 and EDBRAC0 registers to control allocation of individual resources to the internal or external debug facilities. Consult the Core Reference Manual.

3.13.2 Debug Control Registers (DBCR0–DBCR6)

The debug control registers are used to enable debug events, reset the processor, control timer operation during debug events, and set the debug mode of the processor.

3.13.2.1 Debug Control Register 0 (DBCR0)

DBCR0, shown in Figure 3-82, is used to control the debug internal debug facility and in particular to enable specific debug events.

This register is hypervisor privileged.

Writing to this register requires synchronization.

SPR 308 Hypervisor

	32	33	34–35	36	37	38	39	40	41	42	43	44–45	46–47
R	EDM	IDM	RST	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1	DAC2
W[1]													
Reset	All zeros												

	48	49–56	57	58	59	60–62	63
R W[1]	RET	—	CIRPT	CRET	VLES	—	FT
Reset	All zeros						

Figure 3-82. Debug Control Register 0 (DBCR0)

[1] Individual bits in DBCR0 may only be written by software when the associated resource defined by EDBRAC0 is allocated to the internal debug facility. Some processors restrict all bits from being written when DBCR0[EDM] is set.

This table provides bit definitions for DBCR0.

Table 3-66. DBCR0 Field Descriptions

Bits	Name	Description
32	EDM	External debug mode. Indicates whether the processor is in external debug mode. 0 The processor is not in external debug mode. 1 The processor is in external debug mode. In some implementations, if EDM = 1, some debug registers are locked and cannot be accessed. Refer to the implementation documentation for any additional implementation-specific behavior.
33	IDM	Internal debug mode. 0 Internal debug events are disabled. 1 Internal debug events are enabled. A qualified debug condition will generate an internal debug event by setting the corresponding bit in the DBSR. If MSR[DE] = 1, the occurrence of a debug event or the recording of an earlier debug event in the DBSR when MSR[DE] = 0 will cause a debug interrupt. Software note: *Software must clear debug event status in the DBSR in the debug interrupt handler when a debug interrupt is taken before re-enabling interrupts through MSR[DE]. Otherwise, redundant debug interrupts are taken for the same debug event.*
34–35	RST	Reset control. 0*x* Default (No action) 1*x* A hard reset is performed on the processor. This field is always cleared on the cycle after it is written. **Note:** In integrated devices, the hard reset that is initiated is defined by the SoC. SoC devices should define it to perform a hard reset on the processor core that initiated the request.
36	ICMP	Instruction completion debug event enable 0 ICMP debug events are disabled. 1 ICMP debug events are enabled.
37	BRT	Branch taken debug event enable 0 BRT debug events are disabled. 1 BRT debug events are enabled.
38	IRPT	Interrupt taken debug event enable. 0 IRPT debug events are disabled. 1 IRPT debug events are enabled.
39	TRAP	Trap debug event enable 0 TRAP debug events are disabled. 1 TRAP debug events are enabled.
40	IAC1	Instruction address compare 1 debug event enable 0 IAC1 debug events are disabled. 1 IAC1 debug events are enabled.
41	IAC2	Instruction address compare 2 debug event enable. 0 IAC2 debug events are disabled. 1 IAC2 debug events are enabled.
42	IAC3	Instruction address compare 3 debug event enable 0 IAC3 debug events are disabled. 1 IAC3 debug events are enabled.
43	IAC4	Instruction address compare 4 debug event enable 0 IAC4 debug events are disabled. 1 IAC4 debug events are enabled.

Table 3-66. DBCR0 Field Descriptions (continued)

Bits	Name	Description
44–45	DAC1	Data address compare 1 debug event enable 00 DAC1 debug events are disabled. 01 DAC1 debug events are enabled only for store-type data storage access. 10 DAC1 debug events are enabled only for load-type data storage access. 11 DAC1 debug events are enabled for any data storage access.
46–47	DAC2	Data address compare 2 debug event enable 00 DAC2 debug events are disabled. 01 DAC2 debug events are enabled only for store-type data storage access. 10 DAC2 debug events are enabled only for load-type data storage access. 11 DAC2 debug events are enabled for any data storage access.
48	RET	Return debug event enable. 0 RET debug events are disabled. 1 RET debug events are enabled.
49–56	—	Reserved, should be cleared.
57	CIRPT	Critical interrupt taken debug event. <E.ED> 0 Critical interrupt taken debug events are disabled. 1 Critical interrupt taken debug events are enabled.
58	CRET	Critical interrupt return debug event. <E.ED> 0 Critical interrupt return debug events are disabled. 1 Critical interrupt return debug events are enabled.
59	VLES	VLE status <VLE> 0 CRET debug events are disabled. 1 An ICMP, BRT, TRAP, RET, CRET, IAC, or DAC debug event occurred on a VLE instruction.
60–62	—	Reserved
63	FT	Freeze timers on debug event 0 Enable clocking of Time Base. 1 Disable clocking of timers whenever any DBSR bit is set (except MRR).

3.13.2.2 Debug Control Register 1 (DBCR1)

Figure 3-83 shows DBCR1, which is used to configure instruction address comparison debug events.

This register is hypervisor privileged.

Writing to this register requires synchronization.

NOTE: Software Considerations

> Some processors do not implement the IAC3US, IAC3ER, IAC4US, IAC4ER, or IAC34M fields.

SPR 309 Hypervisor

	32 33	34 35	36 37	38 39	40 41	42 47	48 49	50 51	52 53	54 55	56 57	58 63
R W[1]	IAC1US	IAC1ER	IAC2US	IAC2ER	IAC12M	—	IAC3US	IAC3ER	IAC4US	IAC4ER	IAC34M	—
Reset	All zeros											

Figure 3-83. Debug Control Register 1 (DBCR1)

[1] Individual bits in DBCR1 may only be written by software when the associated resource defined by EDBRAC0 is allocated to the internal debug facility. Some processors restrict all bits from being written when DBCR0[EDM] is set.

This table describes DBCR1 fields.

Table 3-67. DBCR1 Field Descriptions

Bits	Name	Description
32–33	IAC1US	Instruction address compare 1 user/supervisor mode 00 IAC1 debug events unaffected by MSR[PR]. 01 Reserved. Setting this value causes IAC1 behavior to be undefined. 10 IAC1 debug events can occur only if MSR[PR]=0 (supervisor mode). 11 IAC1 debug events can occur only if MSR[PR]=1 (user mode). Software Note: *DBCR1 provides no controls to distinguish between guest state and hypervisor state. However, all debug events can be prevented from occurring in hypervisor state using EPCR[DUVD].*
34–35	IAC1ER	Instruction address compare 1 effective/real mode 00 IAC1 debug events are based on effective addresses. 01 IAC1 debug events are based on real addresses. 10 IAC1 debug events are based on effective addresses and can occur only if MSR[IS]=0. 11 IAC1 debug events are based on effective addresses and can occur only if MSR[IS]=1. **Note:** Some processors do not implement real address compares. Consult the core reference manual.
36–37	IAC2US	Instruction address compare 2 user/supervisor mode 00 IAC2 debug events unaffected by MSR[PR]. 01 Reserved. Setting this value causes IAC2 behavior to be undefined. 10 IAC2 debug events can occur only if MSR[PR]=0 (supervisor mode). 11 IAC2 debug events can occur only if MSR[PR]=1 (user mode).
38–39	IAC2ER	Instruction address compare 2 effective/real mode 00 IAC2 debug events are based on effective addresses. 01 IAC2 debug events are based on real addresses. 10 IAC2 debug events are based on effective addresses and can occur only if MSR[IS]=0. 11 IAC2 debug events are based on effective addresses and can occur only if MSR[IS]=1. **Note:** Some processors do not implement real address compares. Consult the core reference manual.

Table 3-67. DBCR1 Field Descriptions (continued)

Bits	Name	Description
40–41	IAC12M	Instruction address compare 1/2 mode[1] 00 Exact address compare. IAC1 debug events can occur only if the instruction fetch address equals the value in IAC1. IAC2 debug events can occur only if the instruction fetch address equals the value in IAC2. 01 Address bit match. IAC1 debug events can occur only if the instruction fetch address, ANDed with the contents of IAC2, equals the value in IAC1, also ANDed with the contents of IAC2. IAC2 debug events do not occur. The debug event is enabled by DBCR0[IAC1], DBCR0[IAC2] is ignored. If IAC1US ≠ IAC2US or IAC1ER ≠ IAC2ER, results are boundedly undefined. 10 Inclusive address range compare. IAC1 debug events can occur only if the instruction fetch address is greater than or equal to the value specified in IAC1 and less than the value specified in IAC2. The debug event is enabled by DBCR0[IAC1], DBCR0[IAC2] is ignored. If IAC1US ≠ IAC2US or IAC1ER ≠ IAC2ER, results are boundedly undefined. 11 Exclusive address range compare. IAC1 debug events can occur only if the instruction fetch address is less than the value specified in IAC2 or greater than the value specified in IAC2. The debug event is enabled by DBCR0[IAC1], DBCR0[IAC2] is ignored. If IAC1US ≠ IAC2US or IAC1ER ≠ IAC2ER, results are boundedly undefined. When the instruction address compare mode is anything other than exact address compare mode (IAC12M ≠ 0b00), it is implementation-dependent if both or either associated DBSR[IAC*n*] bits may be set. For IAC1 and IAC2 debug events, either or both DBSR[IAC1] and DBSR[IAC2] bits may be set
42–47	—	Reserved, should be cleared.
48–49	IAC3US	Instruction address compare 3 user/supervisor mode 00 IAC3 debug events unaffected by MSR[PR].. 01 Reserved. Setting this value causes IAC3 behavior to be undefined. 10 IAC3 debug events can occur only if MSR[PR]=0 (supervisor mode). 11 IAC3 debug events can occur only if MSR[PR]=1 (user mode).
50–51	IAC3ER	Instruction address compare 3 effective/real mode 00 IAC3 debug events are based on effective addresses. 01 IAC3 debug events are based on real addresses. 10 IAC3 debug events are based on effective addresses and can occur only if MSR[IS]=0. 11 IAC3 debug events are based on effective addresses and can occur only if MSR[IS]=1. **Note:** Some processors do not implement real address compares. Consult the core reference manual.
52–53	IAC4US	Instruction address compare 4 user/supervisor mode 00 IAC4 debug events unaffected by MSR[PR]. 01 Reserved. Setting this value causes IAC4 behavior to be undefined. 10 IAC4 debug events can occur only if MSR[PR]=0 (supervisor mode). 11 IAC4 debug events can occur only if MSR[PR]=1 (user mode).
54–55	IAC4ER	Instruction address compare 4 effective/real mode 00 IAC4 debug events are based on effective addresses. 01 IAC4 debug events are based on real addresses. 10 IAC4 debug events are based on effective addresses and can occur only if MSR[IS]=0. 11 IAC4 debug events are based on effective addresses and can occur only if MSR[IS]=1. **Note:** Some processors do not implement real address compares. Consult the core reference manual.

Table 3-67. DBCR1 Field Descriptions (continued)

Bits	Name	Description
56–57	IAC34M	Instruction address compare 3/4 mode 00 Exact address compare. IAC3 debug events can occur only if the instruction fetch address equals the value in IAC3. IAC4 debug events can occur only if the instruction fetch address equals the value in IAC4. 01 Address bit match. IAC3 debug events can occur only if the instruction fetch address, ANDed with the contents of IAC4, equals the value in IAC3, also ANDed with the contents of IAC4. IAC4 debug events do not occur. The debug event is enabled by DBCR0[IAC3], DBCR0[IAC4] is ignored. If IAC3US ≠ IIAC4US or IAC3ER ≠ IAC4ER, results are boundedly undefined. 10 Inclusive address range compare. IAC3 debug events can occur only if the instruction fetch address is greater than or equal to the value specified in IAC3 and less than the value specified in IAC4. The debug event is enabled by DBCR0[IAC3], DBCR0[IAC4] is ignored. If IAC3US ≠ IAC4US or IAC3ER ≠ IAC4ER, results are boundedly undefined. 11 Exclusive address range compare. IAC3 debug events can occur only if the instruction fetch address is less than the value specified in IAC3 or greater than the value specified in IAC4. The debug event is enabled by DBCR0[IAC3], DBCR0[IAC4] is ignored. If IAC3US ≠ IAC4US or IAC3ER ≠ IAC4ER, results are boundedly undefined. When the instruction address compare mode is anything other than exact address compare mode (IAC34M ≠ 0b00), it is implementation-dependent if both or either associated DBSR[IAC*n*] bits may be set. For IAC3 and IAC4 debug events, either or both DBSR[IAC3] and DBSR[IAC4] bits may be set.
58–63	—	Reserved, should be cleared.

[1] When MSR[CM] = 0, IAC*n*[0–31] are treated as zero for the purpose of comparison with the fetch address. When MSR[CM] = 1, bits 0–61 (bits 0–62 <VLE>) of the fetch address are compared to IAC*n*[0–61] (IAC*n*[0–62] <VLE>)

3.13.2.3 Debug Control Register 2 (DBCR2)

Figure 3-84 shows DBCR2, which is used to configure data address debug events.

This register is hypervisor privileged.

Writing to this register requires synchronization.

NOTE: Software Considerations

Some processors do not implement the DACLINK1 or DACLINK2 fields.

SPR 310 Hypervisor

	32 33	34 35	36 37	38 39	40 41	42	43	44 63
R W[1]	DAC1US	DAC1ER	DAC2US	DAC2ER	DAC12M	DACLINK1	DACLINK2	—
Reset	All zeros							

Figure 3-84. Debug Control Register 2 (DBCR2)

[1] Individual bits in DBCR2 may only be written by software when the associated resource defined by EDBRAC0 is allocated to the internal debug facility. Some processors restrict all bits from being written when DBCR0[EDM] is set.

This table describes DBCR2 fields.

Table 3-68. DBCR2 Field Descriptions

Bits	Name	Description
32–33	DAC1US	Data address compare 1 user/supervisor mode 00 DAC1 debug events unaffected by MSR[PR]. 01 Reserved. Setting this value causes DAC1 behavior to be undefined. 10 DAC1 debug events can occur only if MSR[PR]=0 (supervisor mode). 11 DAC1 debug events can occur only if MSR[PR]=1 (user mode). Software Note: *DBCR1 provides no controls to distinguish between guest state and hypervisor state. However, all debug events can be prevented from occurring in hypervisor state using EPCR[DUVD].*
34–35	DAC1ER	Data address compare 1 effective/real mode 00 DAC1 debug events are based on effective addresses. 01 DAC1 debug events are based on real addresses. 10 DAC1 debug events are based on effective addresses and can occur only if MSR[DS]=0. 11 DAC1 debug events are based on effective addresses and can occur only if MSR[DS]=1. **Note:** Some processors do not implement real address compares. Consult the core reference manual.
36–37	DAC2US	Data address compare 2 user/supervisor mode 00 DAC2 debug events unaffected by MSR[PR]. 01 Reserved. Setting this value causes DAC2 behavior to be undefined. 10 DAC2 debug events can occur only if MSR[PR]=0 (supervisor mode). 11 DAC2 debug events can occur only if MSR[PR]=1 (user mode).
38–39	DAC2ER	Data address compare 2 effective/real mode 00 DAC2 debug events are based on effective addresses. 01 DAC2 debug events are based on real addresses. 10 DAC2 debug events are based on effective addresses and can occur only if MSR[DS]=0. 11 DAC2 debug events are based on effective addresses and can occur only if MSR[DS]=1. **Note:** Some processors do not implement real address compares. Consult the core reference manual.
40–41	DAC12M	Data address compare 1/2 mode[1] 00 Exact address compare.DAC1 debug events can occur only if the data storage address equals the value in DAC1. DAC2 debug events can occur only if the data storage address equals the value in DAC2. DAC1US, DAC1ER, and DBCR0[DAC1] are used for DAC1 conditions. DAC2US, DAC2ER, and DBCR0[DAC2] are used for DAC2 conditions 01 Address bit match. DAC1 debug events can occur only if the data storage address, ANDed with the contents of DAC2, equals the value in DAC1, also ANDed with the contents of DAC2. DAC2 debug events do not occur. The debug event is enabled by DBCR0[DAC1], DBCR0[DAC2] is ignored. If DAC1US ≠ DAC2US or DAC1ER ≠ DAC2ER, results are boundedly undefined. 10 Inclusive address range compare. DAC1 debug events can occur only if the data storage address is greater than or equal to the value specified in DAC1 and less than the value specified in DAC2. The debug event is enabled by DBCR0[DAC1], DBCR0[DAC2] is ignored. If DAC1US ≠ DAC2US or DAC1ER ≠ DAC2ER, results are boundedly undefined. 11 Exclusive address range compare. DAC1 debug events can occur only if the data storage address is less than the value specified in DAC2 or greater than the value specified in DAC2. The debug event is enabled by DBCR0[DAC1], DBCR0[IDC2] is ignored.[2] If DAC1US ≠ DAC2US or DAC1ER ≠ DAC2ER, results are boundedly undefined. When the data address compare mode is anything other than exact address compare mode (DAC12M ≠ 0b00), it is implementation-dependent if both or either associated DBSR[DAC*n*] bits may be set. For DAC1 and DAC2 debug events, either or both DBSR[DAC1] and DBSR[DAC2] bits may be set.

Table 3-68. DBCR2 Field Descriptions (continued)

Bits	Name	Description
42	DACLINK1	Data address compare 1 link to instruction address compare 1 0 No effect 1 DAC1 debug events are linked to IAC1 debug conditions. IAC1 debug conditions do not affect DBSR. When linked to IAC1, the DAC1 debug event is qualified based on whether the instruction also generated an IAC1 debug condition. If DACLINK1 is set both the data address compare mode for DAC1 (DAC12M) and the instruction address compare mode for IAC1 (IAC12M) must be in exact address compare mode (=0b00) or the result is boundedly undefined.
43	DACLINK2	Data address compare 2 link to instruction address compare 1 0 No effect 1 DAC2 debug events are linked to IAC2 debug conditions. IAC2 debug conditions do not affect DBSR. When linked to IAC2, the DAC2 debug event is qualified based on whether the instruction also generated an IAC2 debug condition. If DACLINK2 is set both the data address compare mode for DAC2 (DAC12M) and the instruction address compare mode for IAC2 (IAC12M) must be in exact address compare mode (=0b00) or the result is boundedly undefined.
44–63	—	Reserved, should be cleared.

[1] When MSR[CM] = 0, DAC*n*[0–31] are treated as zero for the purpose of comparison with the data address. When MSR[CM] = 1, bits 0–63 of the data address are compared to DAC*n*[0–63].

[2] If DAC1 > DAC2 or DAC1 = DAC2, a valid condition may occur on every data storage address.

3.13.2.4 Debug Control Register 3 (DBCR3)

If DBCR3, shown in Figure 3-85, is implemented, its contents are implementation specific. Consult the core and integrated-device documentation.

This register is hypervisor privileged.

Writing to this register requires synchronization.

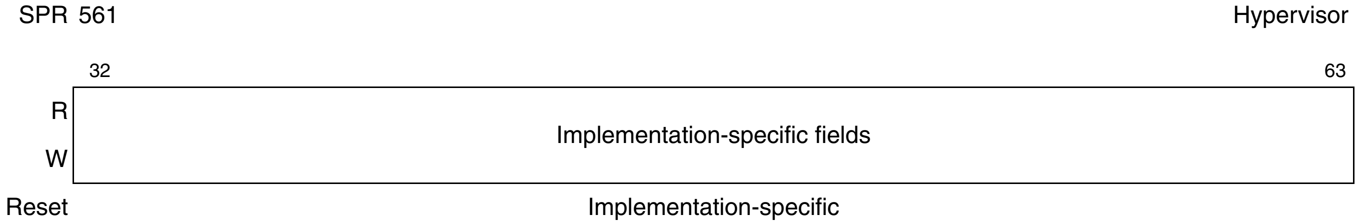


Figure 3-85. Debug Control Register 3 (DBCR3)

3.13.2.5 Debug Control Register 4 (DBCR4)

If DBCR4, shown in Figure 3-86, is implemented, its contents are implementation specific. Consult the core and integrated-device documentation.

This register is hypervisor privileged.

Writing to this register requires synchronization.

SPR 563 Hypervisor

	32–63
R W	Implementation-specific fields
Reset	Implementation-specific

Figure 3-86. Debug Control Register 4 (DBCR4)

3.13.2.6 Debug Control Register 5 (DBCR5)

Figure 3-87 shows DBCR5, which is used to configure instruction address comparison debug events.

This register is hypervisor privileged.

Writing to this register requires synchronization.

SPR 564 Hypervisor

	32 33	34 35	36 37	38 39	40 41	42 47	48 49	50 51	52 53	54 55	56 57	58 63
R W[1]	IAC5US	IAC5ER	IAC6US	IAC6ER	IAC56M	—	IAC7US	IAC7ER	IAC8US	IAC8ER	IAC78M	—
Reset	All zeros											

Figure 3-87. Debug Control Register 5 (DBCR5)

[1] Individual bits in DBCR5may only be written by software when the associated resource defined by EDBRAC0 is allocated to the internal debug facility. Some processors restrict all bits from being written when DBCR0[EDM] is set.

This table describes DBCR5 fields.

Table 3-69. DBCR5 Field Descriptions

Bits	Name	Description
32–33	IAC5US	Instruction address compare 5 user/supervisor mode 00 IAC5 debug events unaffected by MSR[PR]. 01 Reserved. Setting this value causes IAC5 behavior to be undefined. 10 IAC5 debug events can occur only if MSR[PR]=0 (supervisor mode). 11 IAC5 debug events can occur only if MSR[PR]=1 (user mode). Software Note: *DBCR1 provides no controls to distinguish between guest state and hypervisor state. However, all debug events can be prevented from occurring in hypervisor state using EPCR[DUVD].*
34–35	IAC5ER	Instruction address compare 5 effective/real mode 00 IAC5 debug events are based on effective addresses. 01 IAC5 debug events are based on real addresses. 10 IAC5 debug events are based on effective addresses and can occur only if MSR[IS]=0. 11 IAC5 debug events are based on effective addresses and can occur only if MSR[IS]=1. **Note:** Some processors do not implement real address compares. Consult the core reference manual.

Table 3-69. DBCR5 Field Descriptions (continued)

Bits	Name	Description
36–37	IAC6US	Instruction address compare 6 user/supervisor mode 00 IAC6 debug events unaffected by MSR[PR]. 01 Reserved. Setting this value causes IAC6 behavior to be undefined. 10 IAC6 debug events can occur only if MSR[PR]=0 (supervisor mode). 11 IAC6 debug events can occur only if MSR[PR]=1 (user mode).
38–39	IAC6ER	Instruction address compare 6 effective/real mode 00 IAC6 debug events are based on effective addresses. 01 IAC6 debug events are based on real addresses. 10 IAC6 debug events are based on effective addresses and can occur only if MSR[IS]=0. 11 IAC6 debug events are based on effective addresses and can occur only if MSR[IS]=1. **Note:** Some processors do not implement real address compares. Consult the core reference manual.
40–41	IAC56M	Instruction address compare 5/6 mode[1] 00 Exact address compare. IAC5 debug events can occur only if the instruction fetch address equals the value in IAC5. IAC6 debug events can occur only if the instruction fetch address equals the value in IAC6. 01 Address bit match. IAC5 debug events can occur only if the instruction fetch address, ANDed with the contents of IAC6, equals the value in IAC5, also ANDed with the contents of IAC6. IAC6 debug events do not occur. The debug event is enabled by DBCR0[IAC5], DBCR0[IAC6] is ignored. If IAC5US ≠ IAC6US or IAC5ER ≠ IAC6ER, results are boundedly undefined. 10 Inclusive address range compare. IAC5 debug events can occur only if the instruction fetch address is greater than or equal to the value specified in IAC5 and less than the value specified in IAC6. The debug event is enabled by DBCR0[IAC5], DBCR0[IAC6] is ignored. If IAC5US ≠ IAC6US or IAC5ER ≠ IAC6ER, results are boundedly undefined. 11 Exclusive address range compare. IAC5 debug events can occur only if the instruction fetch address is less than the value specified in IAC6 or greater than the value specified in IAC6. The debug event is enabled by DBCR0[IAC5], DBCR0[IAC6] is ignored. If IAC5US ≠ IAC6US or IAC5ER ≠ IAC6ER, results are boundedly undefined. When the instruction address compare mode is anything other than exact address compare mode (IAC56M ≠ 0b00), it is implementation-dependent if both or either associated DBSR[IAC*n*] bits may be set. For IAC5 and IAC6 debug events, either or both DBSR[IAC5] and DBSR[IAC6] bits may be set
42–47	—	Reserved, should be cleared.
48–49	IAC7US	Instruction address compare 7 user/supervisor mode 00 IAC7 debug events unaffected by MSR[PR].. 01 Reserved. Setting this value causes IAC7behavior to be undefined. 10 IAC7 debug events can occur only if MSR[PR]=0 (supervisor mode). 11 IAC7 debug events can occur only if MSR[PR]=1 (user mode).
50–51	IAC7ER	Instruction address compare 7 effective/real mode 00 IAC7 debug events are based on effective addresses. 01 IAC7 debug events are based on real addresses. 10 IAC7 debug events are based on effective addresses and can occur only if MSR[IS]=0. 11 IAC7 debug events are based on effective addresses and can occur only if MSR[IS]=1. **Note:** Some processors do not implement real address compares. Consult the core reference manual.
52–53	IAC8US	Instruction address compare 8 user/supervisor mode 00 IAC8 debug events unaffected by MSR[PR]. 01 Reserved. Setting this value causes IAC8 behavior to be undefined. 10 IAC8 debug events can occur only if MSR[PR]=0 (supervisor mode). 11 IAC8 debug events can occur only if MSR[PR]=1 (user mode).

Table 3-69. DBCR5 Field Descriptions (continued)

Bits	Name	Description
54–55	IAC8ER	Instruction address compare 8 effective/real mode 00 IAC8 debug events are based on effective addresses. 01 IAC8 debug events are based on real addresses. 10 IAC8 debug events are based on effective addresses and can occur only if MSR[IS]=0. 11 IAC8 debug events are based on effective addresses and can occur only if MSR[IS]=1. **Note:** Some processors do not implement real address compares. Consult the core reference manual.
56–57	IAC78M	Instruction address compare 7/8 mode 00 Exact address compare. IAC7 debug events can occur only if the instruction fetch address equals the value in IAC7. IAC8 debug events can occur only if the instruction fetch address equals the value in IAC8. 01 Address bit match. IAC7 debug events can occur only if the instruction fetch address, ANDed with the contents of IAC8, equals the value in IAC7, also ANDed with the contents of IAC8. IAC8 debug events do not occur. The debug event is enabled by DBCR0[IAC7], DBCR0[IAC8] is ignored. If IAC7US ≠ IIAC8US or IAC7ER ≠ IAC8ER, results are boundedly undefined. 10 Inclusive address range compare. IAC7 debug events can occur only if the instruction fetch address is greater than or equal to the value specified in IAC7 and less than the value specified in IAC8. The debug event is enabled by DBCR0[IAC7], DBCR0[IAC8] is ignored. If IAC7US ≠ IAC8US or IAC7ER ≠ IAC8ER, results are boundedly undefined. 11 Exclusive address range compare. IAC7 debug events can occur only if the instruction fetch address is less than the value specified in IAC7 or greater than the value specified in IAC8. The debug event is enabled by DBCR0[IAC7], DBCR0[IAC8] is ignored. If IAC7US ≠ IAC8US or IAC7ER ≠ IAC8ER, results are boundedly undefined. When the instruction address compare mode is anything other than exact address compare mode (IAC78M ≠ 0b00), it is implementation-dependent if both or either associated DBSR[IAC*n*] bits may be set. For IAC7 and IAC8 debug events, either or both DBSR[IAC7] and DBSR[IAC8] bits may be set.
58–63	—	Reserved, should be cleared.

[1] When MSR[CM] = 0, IAC*n*[0–31] are treated as zero for the purpose of comparison with the fetch address. When MSR[CM] = 1, bits 0–61 (bits 0–62 <VLE>) of the fetch address are compared to IAC*n*[0–61] (IAC*n*[0–62] <VLE>)

3.13.2.7 Debug Control Register 6 (DBCR6)

If DBCR6, shown in Figure 3-88, is implemented, its contents are implementation specific. Consult the core and integrated-device documentation.

This register is hypervisor privileged.

Writing to this register requires synchronization.

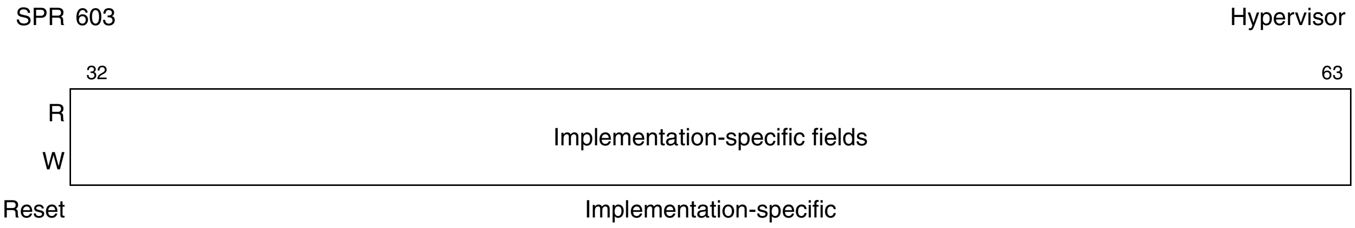


Figure 3-88. Debug Control Register 6 (DBCR6)

3.13.2.8 Debug External Control Register 0 (DBECR0)

If DBECR0, shown in Figure 3-89, is implemented, its contents are implementation specific. Consult the core and integrated-device documentation.

This register is hypervisor privileged.

Writing to this register requires synchronization.

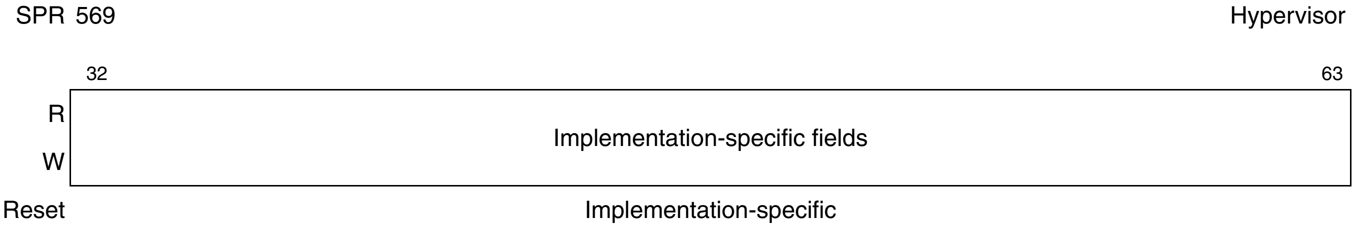


Figure 3-89. Debug External Control Register 0 (DBECR0)

3.13.3 Debug Counter Register (DBCNT)

DBCNT, shown in Figure 3-90, is used to count certain enabled debug conditions by decrementing the CNT1 and CNT2 fields in DBCNT when such debug conditions occur. When a count value reaches zero, a debug count event is signaled if enabled and the counter value is frozen. DBCNT is not considered to be a part of EIS and requires other implementation specific debug controls to be implemented. Consult the core and integrated-device documentation.

This register is hypervisor privileged.

Writing to this register requires synchronization.

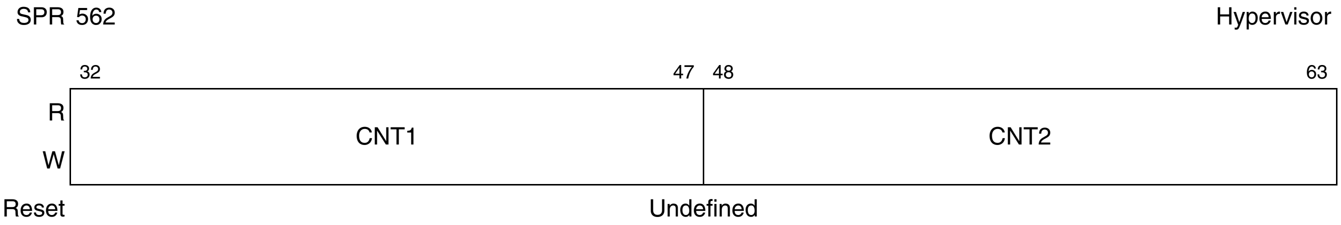


Figure 3-90. Debug Counter Register (DBCNT)

3.13.4 Debug Status Register (DBSR/DBSRWR)

DBSR, shown in Figure 3-91, provides status information for debug events and for the most recent processor reset. The DBSR is set through hardware, but is read through software using **mfspr** and cleared by writing a bit mask of ones to clear individual bits; writing zeros has no effect.

<Embedded.Hypervisor>:
DBSRWR is used to write DBSR to a specific value. DBSRWR is an alias to the same physical register as

DBSR, except that it allows a value to be written directly. Writing DBSRWR changes the value of the DBSR which, if nonzero, may cause later imprecise debug interrupts.

NOTE: Software Considerations <E.HV>

> DBSRWR should only be used to restore a DBSR value for operations such as a partition switch.

These registers are hypervisor privileged.

Writing to these registers require synchronization.

SPR: 304 (DBSR) Hypervisor R/Clear
306 (DBSRWR) Hypervisor WO

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
R	IDE	UDE	MRR		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1R	DAC1W	DAC2R	DAC2W
W	w1c	w1c	w1c		w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c
Reset	All zeros															

Enhanced Debug

	48	49	50	51	52	53	56	57	58	59	60	63
R	RET	IAC5	IAC6	IAC7	IAC8	—		CIRPT	CRET	DNI	—	
W	w1c	w1c	w1c	w1c	w1c			w1c	w1c	w1c		
Reset	All zeros											

Figure 3-91. Debug Status Register (DBSR/DBSRWR)

This table describes DBSR bit definitions.

Table 3-70. DBSR Field Descriptions

Bits	Name	Description
32	IDE	Imprecise debug event. 0 No imprecise debug events have occurred. 1 An imprecise debug event has occurred while MSR[DE] = 0 and a debug event causes its respective DBSR bit to be set. Software Note: *Imprecise debug events have been removed from EREF, although certain processors may support them. Software should not depend on the value of this bit.*
33	UDE	Unconditional debug event. 0 No unconditional debug events have occurred. 1 An unconditional debug event occurred. The source of unconditional debug events is implementation specific, however on many implementations it is a signal to the processor core from another core or other parts of the integrated logic. See the core and integrated device reference manuals.Note that UDE events are asynchronous. UDE debug events are not suppressed by the setting of EPCR[DUVD] .<E.HV>

Table 3-70. DBSR Field Descriptions (continued)

Bits	Name	Description
34–35	MRR	Most recent reset. Set when a reset occurs. Set to zero at power-up. See the core reference manual documentation. 0x No reset occurred since these bits were last cleared by software. 1x A previous reset occurred.
36	ICMP	Instruction complete debug event. 0 No instruction complete debug events have occurred. 1 An instruction complete debug event occurred. Set if an instruction completion debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, DBCR0[ICMP] = 1, and MSR[DE] = 1.
37	BRT	Branch taken debug event. 0 No branch taken debug events have occurred. 1 A branch taken debug event occurred. Set if a branch taken debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, DBCR0[BRT] = 1, and MSR[DE] = 1.
38	IRPT	Interrupt taken debug event. 0 No interrupt taken debug events have occurred. 1 An interrupt taken debug event occurred. Set if an interrupt taken debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[IRPT] = 1.
39	TRAP	Trap instruction debug event. 0 No trap instruction debug events have occurred. 1 A trap instruction debug event occurred. Set if a trap instruction debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[TRAP] = 1.
40	IAC1	Instruction address compare 1 debug event. 0 No instruction address compare 1 debug events have occurred. 1 An instruction address compare 1 debug event occurred. Set if an instruction address compare 1 debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[IAC1] = 1.
41	IAC2	Instruction address compare 2 debug event. 0 No instruction address compare 2 debug events have occurred. 1 An instruction address compare 2 debug event occurred. Set if an instruction address compare 2 debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[IAC2] = 1.
42	IAC3	Instruction address compare 3 debug event. 0 No instruction address compare 3 debug events have occurred. 1 An instruction address compare 3 debug event occurred. Set if an instruction address compare 3 debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[IAC3] = 1.
43	IAC4	Instruction address compare 4 debug event. 0 No instruction address compare 4 debug events have occurred. 1 An instruction address compare 4 debug event occurred. Set if an instruction address compare 4 debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[IAC4] = 1.
44	DAC1R	Data address compare 1 read debug event. 0 No data address compare 1 read debug events have occurred. 1 A data address compare 1 read debug event occurred. Set if a data address compare 1 read debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[DAC1] = 0b10 or 0b11.

Table 3-70. DBSR Field Descriptions (continued)

Bits	Name	Description
45	DAC1W	Data address compare 1 write debug event. 0 No data address compare 1 write debug events have occurred. 1 A data address compare 1 write debug event occurred. Set if a data address compare 1 write debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[DAC1] = 0b01 or 0b11..
46	DAC2R	Data address compare 2 read debug event. 0 No data address compare 2 read debug events have occurred. 1 A data address compare 2 read debug event occurred. Set if a data address compare 2 read debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[DAC2] = 0b10 or 0b11.
47	DAC2W	Data address compare 2 write debug event. 0 No data address compare 2 write debug events have occurred. 1 A data address compare 2 write debug event occurred. Set if a data address compare 2 write debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[DAC2] = 0b01 or 0b11..
48	RET	Interrupt return debug event. 0 No interrupt return debug events have occurred. 1 An interrupt return debug event occurred. Set if an interrupt return debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[RET] = 1.
49	IAC5	Instruction address compare 5 debug event. 0 No instruction address compare 5debug events have occurred. 1 An instruction address compare 5 debug event occurred. Set if an instruction address compare 5 debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[IAC1] = 1.
50	IAC6	Instruction address compare 6 debug event. 0 No instruction address compare 6 debug events have occurred. 1 An instruction address compare 6 debug event occurred. Set if an instruction address compare 6 debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[IAC2] = 1.
51	IAC7	Instruction address compare 7 debug event. 0 No instruction address compare 7 debug events have occurred. 1 An instruction address compare 7 debug event occurred. Set if an instruction address compare 7 debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[IAC3] = 1.
52	IAC8	Instruction address compare 8 debug event. 0 No instruction address compare 8 debug events have occurred. 1 An instruction address compare 8 debug event occurred. Set if an instruction address compare 8 debug condition occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[IAC4] = 1.
53–56	—	Reserved, should be cleared.
57	CIRPT	Critical interrupt taken debug event. <E.ED> 0 No critical interrupt taken debug event has occurred. 1 A critical interrupt taken debug event occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[CIRPT] = 1
58	CRET	Critical interrupt return debug event. <E.ED> 0 No critical interrupt return debug event has occurred. 1 A critical interrupt return debug event occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and DBCR0[CRET] = 1.

Table 3-70. DBSR Field Descriptions (continued)

Bits	Name	Description
59	DNI	**dni** instruction debug event. <E.ED> 0 No **dni** instruction debug event has occurred. 1 A **dni** instruction debug event has occurred while DBCR0[IDM] = 1, the resource is controlled by the internal debug facility, and MSR[DE] = 1.
59–63	—	Reserved, should be cleared.

3.13.5 Instruction Address Compare Registers (IAC*n*)

IAC*n*, shown in Figure 3-92, specify instruction addresses to be compared. A debug event may be enabled to occur on an attempt to execute an instruction from an address specified in an IAC*n*, inside or outside a range specified either by IAC*n* pairs (1,2 or 3,4 or 5,6 or 7,8), or to blocks of addresses specified by the combination of IAC*n* pairs. IAC1 and IAC2 are also used to specify the instruction address for DACLINK1 and DACLINK2 capability.

Which bits of IAC*n* participate in the comparison to the instruction address depends on computation mode and whether the core implements category VLE as shown in the following table.

Table 3-71. IAC Bits Used to Compare to Instruction Address

MSR[CM]	VLE Implemented	IAC*n* Bits Compared
0	no	32–61
0	yes	32–62
1	no	0–61
1	yes	0–62

For implementations that do not support category VLE, instruction addresses must be word-aligned, therefore IAC*n*[62–63] are reserved. For implementations that support category VLE, instruction addresses must be half-word-aligned, therefore IAC*n*[63] is reserved.

These registers are hypervisor privileged.

Writing to these registers requires synchronization.

IAC1 and IAC2 are required to be implemented by EIS. Other IAC*n* registers may not be implemented.

SPR 312 (IAC1), 313 (IAC2), 314 (IAC3), 315 (IAC4),
565 (IAC5), 566 (IAC6), 567 (IAC7), 568 (IAC8) Hypervisor

	0 63
R W	instruction address
Reset	All zeros

Figure 3-92. Instruction Address Compare Registers (IAC1–IAC8)

3.13.6 Data Address Compare Registers (DAC*n*)

DAC*n*, shown in Figure 3-93, specify the addresses used for data address comparison. A debug event may be enabled to occur on loads, stores, or cache operations to an address specified in DAC*n* pairs (1,2), or to blocks of addresses specified by the combination of DAC*n* pairs. DAC1 and DAC2 are also used to specify the data address for DACLINK1 and DACLINK2 capability.

The contents of DAC*n* are compared to the address generated by a data storage access instruction.

These registers are hypervisor privileged.

Writing to these registers requires synchronization.

DAC1 and DAC2 are required to be implemented by EIS.

SPR 316 (DAC1), 317 (DAC2) Hypervisor

	0 63
R W	Data address
Reset	All zeros

Figure 3-93. Data Address Compare Registers (DAC*n*)

3.13.7 Data Value Compare Registers (DVC1 and DVC2)

DVC1 and DVC2 are shown in Figure 3-94. A DAC1R, DAC1W, DAC2R, or DAC2W debug event may be enabled to occur upon loads or stores of a specific data value specified in either or both DVC1 and DVC2. DBCR2[DVC1M,DVC1BE] control how the specified value is compared with the DVC1 contents; DBCR2[DVC2M,DVC2BE] control how the value is compared with the DVC2 contents.

The registers DVC1U and DVC2U provide aliases to the upper 32 bits of the DVC1 and DVC2 registers respectively.

These registers are hypervisor privileged.

Writing to these registers requires synchronization.

DVC1 and DVC2 are not required to be implemented by EIS.

NOTE: Architecture Considerations

DVC1U and DVCU2 provide aliases for accessing the upper 32-bits of DVC1 and DVC2. These are required for implementations which support data value compare and have 64-bit data types, but are only a 32-bit implementation. Arguably, any 32-bit implementation which implements data value compare and category FP would implement these since the floating-point data can be 64 bits.

SPR 318 (DVC1); 319 (DVC2) Hypervisor

	0 — 63
R W	Data value
Reset	All zeros

Figure 3-94. Data Value Compare Registers (DVC1–DVC2)

3.13.8 Nexus and External Debug Related Registers

Some processor cores implement run control, data acquisition, tracing, and other functions as defined by the Nexus standard and vendor specific extensions. The Nexus functionality is part of the external debug capabilities provided by the processor core and the integrated device, and is not part of EIS. Some of the registers are accessible to running software on the processor core and EIS defines the SPR numbers for those registers. For programmer convenience, those Nexus and external debug related registers which have EIS assigned SPR numbers are briefly described here. Consult the reference manuals for the core and integrated device for more information about these registers.

3.13.8.1 Nexus SPR Access Registers

The Nexus SPR access registers provide access to the other external debug related registers. Software supplies an index to address the external debug register by writing that index to the Nexus SPR configuration register (NSPC) with a **mtspr**. After NSPC has been written, access to the external registers registers can be made by using **mtspr** and **mfspr** instructions to read and write the Nexus SPR data register (NSPD).

NOTE: Implementation Considerations

External debug registers are normally memory-mapped, and the index is an offset into the base of that memory map.

3.13.8.1.1 Nexus SPR Configuration Register (NSPC)

The NSPC, shown in Figure 3-95, provides a mechanism for software to access Nexus and other external debug resources (through SPR instructions).

This register is hypervisor privileged.

Writing to this register requires synchronization.

NSPC is not required to be implemented by EIS.

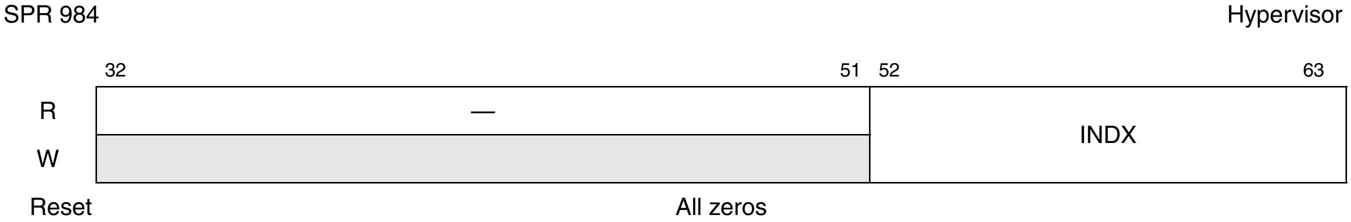


Figure 3-95. Nexus SPR Configuration Register (NSPC)

This table provides the bit definitions for NSPC. See the core reference manual .for the list of the external debug registers that can be accessed.

Table 3-72. NSPC Field Descriptions

Bits	Name	Description
32–51	—	Reserved, should be cleared
52–63	INDX	Register index

3.13.8.1.2 Nexus SPR Data Register (NSPD)

The NSPD, shown in Figure 3-96, provides a mechanism to transfer data to and from the external debug register indicated by the contents of NSPC. For write operations, the write data should be placed into NSPD using **mtspr**. For read operations, the read data is acquired from NSPD using **mfspr**.

This register is hypervisor privileged.

Writing to this register requires synchronization.

NSPD is not required to be implemented by EIS.

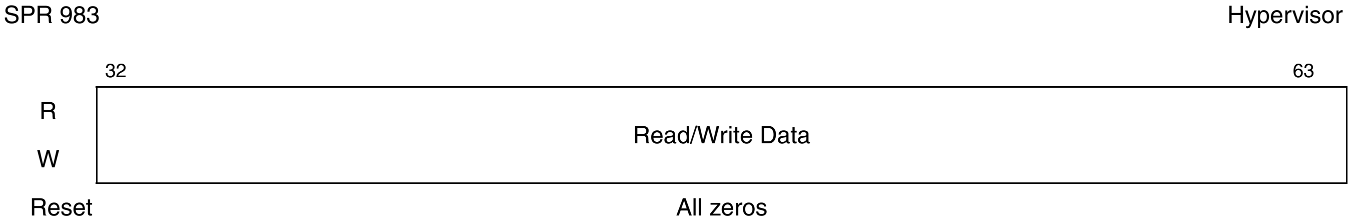


Figure 3-96. Nexus SPR Data Register (NSPD)

3.13.8.2 Debug Event Select Register (DEVENT)

DEVENT, shown in Figure 3-97, allows instrumented software to internally generate signals when an **mtspr** instruction is executed and this register is accessed. The content of the DEVENT register is implementation specific as are any associated capabilities. See the core and integrated device reference manuals.

DEVENT is not required to be implemented by EIS.

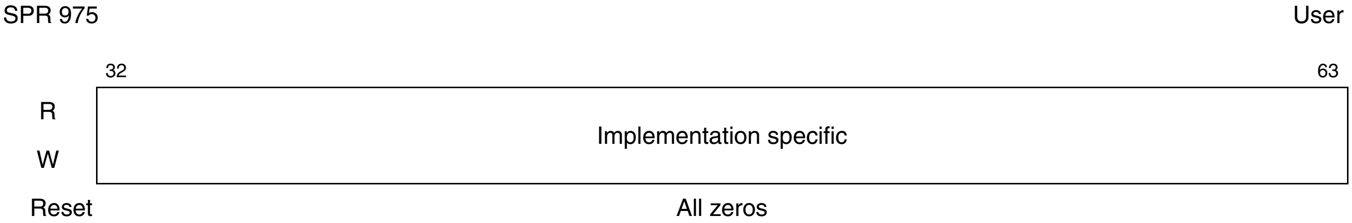


Figure 3-97. Debug Event Register (DEVENT)

3.13.8.3 Debug Data Acquisition Message (DDAM)

DDAM, shown in Figure 3-98, is a portal for software to log information messages to the external debug port. The definition of what actions are taken when a **mtspr** is executed to DDAM are defined by the integrated device. See the core and integrated device reference manuals.

DDAM is not required to be implemented by EIS.

t

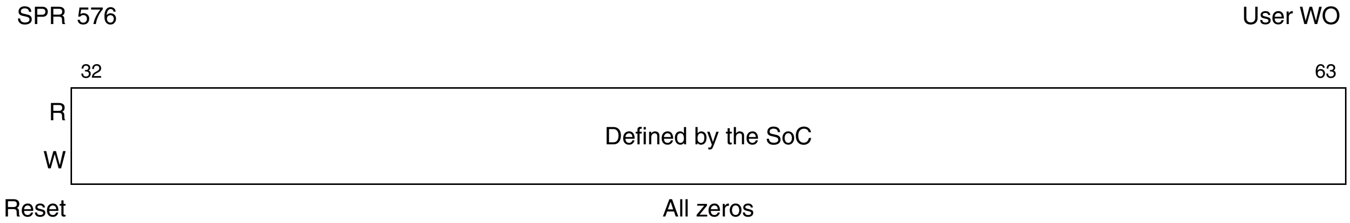


Figure 3-98. Debug Data Acquisition Message (DDAM)

3.13.8.4 Nexus Process ID Register (NPIDR)

NPIDR, is shown in Figure 3-99,allows the full process ID utilized by the OS to be transmitted within Nexus Ownership Trace Messages. When **mtspr** is performed to this register, the contents are transferred to the Nexus port.

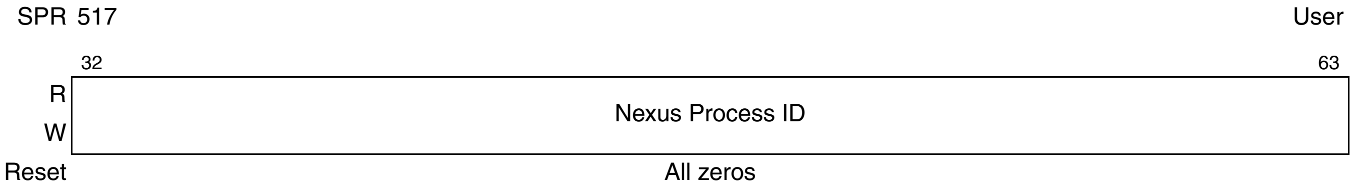


Figure 3-99. Nexus Process ID Register (NPIDR)

3.14 Processor Management Registers

3.14.1 Core Device Control and Status Register 0 (CDCSR0)

CDCSR0, shown in Figure 3-100, provides fields for software to query and control subsystems within the processor. A device is such a subsystem which normally embodies a group of instructions and associated registers. The primary purpose of these fields is to allow software to determine the status of a given subset of processor operations, and controls to change the state of that subset. This is useful for power management and cleaner board support packages based on a specific processor core family, where the processor may appear with different capabilities. If a processor does not have the capability to change states of any of the given devices, when this register is implemented, it is a static read-only value that reflects whether a core contains categories FP, V, or SP.

Core devices controlled by CDCSR0 that provide an MSR available bit to allow execution of instructions in the device, always cause unavailable exceptions if the device is aware, and one of the following conditions exist:

- The state is not ready (note that the state cannot be ready if the device is not present);
- The Available bit in the MSR is not set.

The control and status fields for a core device are-8 bit values that include core device status indicators and core device control. The fields are shown in Figure 3-101 and the CDCSR0 is shown in Figure 3-100.

SPR 696 Hypervisor

	32 39	40 47	48 55	56 63
R W	Floating Point Device	AltiVec Device	—	SPE Device
Reset	Implementation dependent (see bit fields above)			

Figure 3-100. Core Device Control and Status Register 0 (CDCSR0)

The CDCSR0 fields are described in this table.

Table 3-73. CDCSR0 Fields

Bits	Name	Descriptions
32–39	Floating-point device	Device field for floating-point core device and instructions.
40–47	AltiVec device	Device field for AltiVec core device and instructions.
48–55	—	Reserved.
56–63	SPE device	Device field for SPE core device and instructions.

Device control can be performed for each device that is Aware, Present, and not in the No Capability State, as described in Figure 3-100. Writing a value to the CNTL field causes the core device to transition to a new state. When the core device has properly completed the transition, the value of the CNTL field is set to 0 by hardware. A value of 0 for CNTL implies that no operation is in progress. Writing a nonzero value to CNTL while another operation is in progress (for example, CNTL is already nonzero) produces undefined behavior.

The 8-bit CDCSR0 device field format fields are described in Table 3-101.

	0	1	2 4	5 7
Field	AWARE	PRESENT	STATE	CNTL
Reset	Implementation Dependent			0
Type	Read Only	Read Only	Read Only	Read / Write

Figure 3-101. CDCSR0 Device Field Format

Table 3-74. CDCSR0 Device Field Format Fields

Bits	Name	Descriptions
0	AWARE	Aware status. Indicates whether the processor is aware of the programming model of the device. Processor families should always reflect the same AWARE status for all versions of the processor. AWARE status implies that the processor decodes instructions in the device and does not define those instructions as illegal. AWARE status is static for a given implementation and for a given processor family. 0 The processor is not aware of the programming model for the device and takes an illegal program interrupt when an instruction associated with the device attempts execution. 1 The processor is aware of the device and its associated instructions and programming model. It takes an unavailable interrupt (if the core device is controlled by an MSR available bit) if the device state is not ready.
1	PRESENT	Present status. Reflects whether the device itself is physically present in the processor. A processor may be aware of a device, but the device may not be present if it is not physically part of the core. The present status is static for a given implementation. 0 The device is not physically present. 1 The device is physically present.
2–4	STATE	State status. Reflects whether a present device can execute instructions successfully if all other programming model requirements are met. 000 No capability.The device has no capability to transition to a ready state. This may be because power is not supplied to the device and no capability exists to supply power or the device is not present. 001 Ready. The processor is aware of the device, the device is present and is appropriately powered. 010 Standby. The device is currently in standby. The architectural state controlled by the device (register files and associated architected status registers) is still valid, but the device cannot execute instructions until it is brought into the Ready state. If software wishes to execute instructions, it should bring the device into the Ready state. 011 Off. The device is currently off. The architectural state controlled by the device (register files and associated architected status registers) is not valid and the device cannot execute instructions until it is brought into the Ready state. If software wishes to execute instructions, it should bring the device into the Ready state and restore the associated architectural state.
5–7	CNTL	Control operation for changing the state of the device. 000 No operation in progress. The CNTL field is set to 0 by hardware when a nonzero value is written to the field and hardware has completed the operation.

Table 3-75. CDCSR0 Device Architectural State

Device	Available Bit	Unavailable Interrupt	Architectural State
Floating Point	MSR[FP]	Floating-Point unavailable interrupt	FPR0–FPR31, FPSCR
AltiVec	MSR[SPV]	SPE/Embedded Floating-Point/AltiVec unavailable interrupt	VR0–VR31, VSCR
SPE	MSR[SPV]	SPE/Embedded Floating-Point/AltiVec unavailable interrupt	Upper 32 bits of GPR0–GPR31 (for 32-bit implementations only), SPEFSCR, ACC

If the device is not present (PRESENT = 0) and is aware (AWARE = 1), software should not allow the associated “available” bit to be set in the MSR. Doing so will produce undefined behavior.

This table details the expected behavior based on PRESENT, AWARE, STATE, and the associated “available” bit:

Table 3-76. Behavior of CDCSR0 Devices Based on State

PRESENT	AWARE	STATE	“Available” Bit in MSR	Operation
x	0	x	x	Program interrupt - illegal operation
0	1	x[1]	0	Unavailable interrupt
1	1	not Ready	0	Unavailable interrupt
1	1	not Ready	1	Unavailable interrupt
1	1	Ready	0	Unavailable interrupt
1	1	Ready	1	Execute instruction

[1] Devices that are not present should always have a STATE of 0b000 (No capability).

3.15 Performance Monitor Registers (PMRs) <E.PM>

The EIS defines a set of register resources used exclusively by the performance monitor. PMRs are similar to the SPRs and are accessed by **mtpmr** and **mfpmr**, which are also defined by the EIS. Table 3-77 lists PMRs with their name and privilege level.

Processors will generally implement only a certain number of performance counters and thus only a subset of the registers defined will be implemented. For each PMC*x* register implemented, a set of corresponding PMLCa*x* and PMLCb*x* registers will be implemented. Any processor that implements PMRs will always implement PMGC0. See the core reference manual to determine which PMRs are implemented. Also note that it may be common for processors to implement more fields in some of the PMRs which are implementation specific that offer more control over the counting of events.

PMRs are either guest supervisor privileged or user RO privileged. User RO privileged PMRs are aliased to specific guest supervisor PMRs to allow these registers to be read in user mode.

The contents of a PMR can be read into a GPR using **mfpmr r**D**,**PMRN. GPR contents can be written into a PMR using **mtpmr** PMRN**,r**S, where PMRN represents the PMR number.

Executing **mtpmr** to a guest supervisor privileged PMR number in user mode results in a privilege exception. Executing **mtpmr** or **mfpmr** to an unimplemented or undefined PMR number results in an illegal instruction exception.

<Embedded.Hypervisor>:
Executing **mfpmr** to user privileged PMRs returns all zeros in the destination register if MSRP[PMMP] is set. Executing mfpmr or mtpmr to a guest supervisor privileged PMR while in guest supervisor state if MSRP[PMMP] is set causes an embedded hypervisor privilege exception.

Table 3-77. Freescale EIS PMRs

Number	Name	Privilege	Description	Section/Page
0-15	*User Performance Monitor Counter Registers*			
0	UPMC0	User RO[1]	User Performance Monitor Counter Register 0. Alias to PMC0.	
1	UPMC1	User RO[1]	User Performance Monitor Counter Register 1. Alias to PMC1.	
2	UPMC2	User RO[1]	User Performance Monitor Counter Register 2. Alias to PMC2.	
3	UPMC3	User RO[1]	User Performance Monitor Counter Register 3. Alias to PMC3.	
4	UPMC4	User RO[1]	User Performance Monitor Counter Register 4. Alias to PMC4.	
5	UPMC5	User RO[1]	User Performance Monitor Counter Register 5. Alias to PMC5.	
6	UPMC6	User RO[1]	User Performance Monitor Counter Register 6. Alias to PMC6.	
7	UPMC7	User RO[1]	User Performance Monitor Counter Register 7. Alias to PMC7.	
8	UPMC8	User RO[1]	User Performance Monitor Counter Register 8. Alias to PMC8.	
9	UPMC9	User RO[1]	User Performance Monitor Counter Register 9. Alias to PMC9.	
10	UPMC10	User RO[1]	User Performance Monitor Counter Register 10. Alias to PMC10.	
11	UPMC11	User RO[1]	User Performance Monitor Counter Register 11. Alias to PMC11.	
12	UPMC12	User RO[1]	User Performance Monitor Counter Register 12. Alias to PMC12.	

Table 3-77. Freescale EIS PMRs

Number	Name	Privilege	Description	Section/Page
13	UPMC13	User RO[1]	User Performance Monitor Counter Register 13. Alias to PMC13.	
14	UPMC14	User RO[1]	User Performance Monitor Counter Register 14. Alias to PMC14.	
15	UPMC15	User RO[1]	User Performance Monitor Counter Register 15. Alias to PMC15.	
16-31	*Supervisor Performance Monitor Counter Registers*			
16	PMC0	Guest supervisor[2]	Supervisor Performance Monitor Counter Register 0. Alias to UPMC0.	
17	PMC1	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 1. Alias to UPMC1.	
18	PMC2	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 2. Alias to UPMC2.	
19	PMC3	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 3. Alias to UPMC3.	
20	PMC4	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 4. Alias to UPMC4.	
21	PMC5	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 5. Alias to UPMC5.	
22	PMC6	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 6. Alias to UPMC6.	
23	PMC7	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 7. Alias to UPMC7.	
24	PMC8	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 8. Alias to UPMC8.	
25	PMC9	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 9. Alias to UPMC9.	
26	PMC10	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 10. Alias to UPMC10.	
27	PMC11	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 11. Alias to UPMC11.	
28	PMC12	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 12. Alias to UPMC12.	
29	PMC13	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 13. Alias to UPMC13.	
30	PMC14	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 14. Alias to UPMC14.	
31	PMC15	Guest supervisor [2]	Supervisor Performance Monitor Counter Register 15. Alias to UPMC15.	
128-143	*User Performance Monitor Local Control A Registers*			

Table 3-77. Freescale EIS PMRs

Number	Name	Privilege	Description	Section/Page
128	UPMLCA0	User RO[1]	User Performance Monitor Local Control A Register 0. Alias to PMLCA0.	
129	UPMLCA1	User RO[1]	User Performance Monitor Local Control A Register 1. Alias to PMLCA1.	
130	UPMLCA2	User RO[1]	User Performance Monitor Local Control A Register 2. Alias to PMLCA2.	
131	UPMLCA3	User RO[1]	User Performance Monitor Local Control A Register 3. Alias to PMLCA3.	
132	UPMLCA4	User RO[1]	User Performance Monitor Local Control A Register 4. Alias to PMLCA4.	
133	UPMLCA5	User RO[1]	User Performance Monitor Local Control A Register 5. Alias to PMLCA5.	
134	UPMLCA6	User RO[1]	User Performance Monitor Local Control A Register 6. Alias to PMLCA6.	
135	UPMLCA7	User RO[1]	User Performance Monitor Local Control A Register 7. Alias to PMLCA7.	
136	UPMLCA8	User RO[1]	User Performance Monitor Local Control A Register 8. Alias to PMLCA8.	
137	UPMLCA9	User RO[1]	User Performance Monitor Local Control A Register 9. Alias to PMLCA9.	
138	UPMLCA10	User RO[1]	User Performance Monitor Local Control A Register 10. Alias to PMLCA10.	
139	UPMLCA11	User RO[1]	User Performance Monitor Local Control A Register 11. Alias to PMLCA11.	
140	UPMLCA12	User RO[1]	User Performance Monitor Local Control A Register 12. Alias to PMLCA12.	
141	UPMLCA13	User RO[1]	User Performance Monitor Local Control A Register 13. Alias to PMLCA13.	
142	UPMLCA14	User RO[1]	User Performance Monitor Local Control A Register 14. Alias to PMLCA14.	
143	UPMLCA15	User RO[1]	User Performance Monitor Local Control A Register 15. Alias to PMLCA15.	
144-159	*Supervisor Performance Monitor Control A Registers*			
144	PMLCA0	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 0. Alias to UPMLCA0.	
145	PMLCA1	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 1. Alias to UPMLCA1.	
146	PMLCA2	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 2. Alias to UPMLCA2.	

Table 3-77. Freescale EIS PMRs

Number	Name	Privilege	Description	Section/Page
147	PMLCA3	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 3. Alias to UPMLCA3.	
148	PMLCA4	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 4. Alias to UPMLCA4.	
149	PMLCA5	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 5. Alias to UPMLCA5.	
150	PMLCA6	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 6. Alias to UPMLCA6.	
151	PMLCA7	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 7. Alias to UPMLCA7.	
152	PMLCA8	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 8. Alias to UPMLCA8.	
153	PMLCA9	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 9. Alias to UPMLCA9.	
154	PMLCA10	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 10. Alias to UPMLCA10.	
155	PMLCA11	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 11. Alias to UPMLCA11.	
156	PMLCA12	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 12. Alias to UPMLCA12.	
157	PMLCA13	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 13. Alias to UPMLCA13.	
158	PMLCA14	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 14. Alias to UPMLCA14.	
159	PMLCA15	Guest supervisor [2]	Supervisor Performance Monitor Local Control A Register 15. Alias to UPMLCA15.	
256-271	*User Performance Monitor Local Control B Registers*			
256	UPMLCB0	User RO[1]	User Performance Monitor Local Control B Register 0. Alias to PMLCB0.	
257	UPMLCB1	User RO[1]	User Performance Monitor Local Control B Register 1. Alias to PMLCB1.	
258	UPMLCB2	User RO[1]	User Performance Monitor Local Control B Register 2. Alias to PMLCB2.	
259	UPMLCB3	User RO[1]	User Performance Monitor Local Control B Register 3. Alias to PMLCB3.	
260	UPMLCB4	User RO[1]	User Performance Monitor Local Control B Register 4. Alias to PMLCB4.	
261	UPMLCB5	User RO[1]	User Performance Monitor Local Control B Register 5. Alias to PMLCB5.	

Table 3-77. Freescale EIS PMRs

Number	Name	Privilege	Description	Section/Page
262	UPMLCB6	User RO[1]	User Performance Monitor Local Control B Register 6. Alias to PMLCB6.	
263	UPMLCB7	User RO[1]	User Performance Monitor Local Control B Register 7. Alias to PMLCB7.	
264	UPMLCB8	User RO[1]	User Performance Monitor Local Control B Register 8. Alias to PMLCB8.	
265	UPMLCB9	User RO[1]	User Performance Monitor Local Control B Register 9. Alias to PMLCB9.	
266	UPMLCB10	User RO[1]	User Performance Monitor Local Control B Register 10. Alias to PMLCB10.	
267	UPMLCB11	User RO[1]	User Performance Monitor Local Control B Register 11. Alias to PMLCB11.	
268	UPMLCB12	User RO[1]	User Performance Monitor Local Control B Register 12. Alias to PMLCB12.	
269	UPMLCB13	User RO[1]	User Performance Monitor Local Control B Register 13. Alias to PMLCB13.	
270	UPMLCB14	User RO[1]	User Performance Monitor Local Control B Register 14. Alias to PMLCB14.	
271	UPMLCB15	User RO[1]	User Performance Monitor Local Control B Register 15. Alias to PMLCB15.	
272-287	*Supervisor Performance Monitor Control B Registers*			
272	PMLCB0	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 0. Alias to UPMLCB0.	
273	PMLCB1	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 1. Alias to UPMLCB1.	
274	PMLCB2	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 2. Alias to UPMLCB2.	
275	PMLCB3	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 3. Alias to UPMLCB3.	
276	PMLCB4	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 4. Alias to UPMLCB4.	
277	PMLCB5	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 5. Alias to UPMLCB5.	
278	PMLCB6	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 6. Alias to UPMLCB6.	
279	PMLCB7	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 7. Alias to UPMLCB7.	
280	PMLCB8	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 8. Alias to UPMLCB8.	

Table 3-77. Freescale EIS PMRs

Number	Name	Privilege	Description	Section/Page
281	PMLCB9	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 9. Alias to UPMLCB9.	
282	PMLCB10	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 10. Alias to UPMLCB10.	
283	PMLCB11	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 11. Alias to UPMLCB11.	
284	PMLCB12	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 12. Alias to UPMLCB12.	
285	PMLCB13	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 13. Alias to UPMLCB13.	
286	PMLCB14	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 14. Alias to UPMLCB14.	
287	PMLCB15	Guest supervisor [2]	Supervisor Performance Monitor Local Control B Register 15. Alias to UPMLCB15.	
384	UPMGC0	User RO[1]	User Performance Monitor Global Control Register 0. Alias to PMGC0. Defined by the Performance Monitor APU.	
400	PMGC0	Guest supervisor [2]	Supervisor Performance Monitor Global Control Register 0. Alias to UPMGC0. Defined by the Performance Monitor APU.	

[1] When category Embedded.Hypervisor is implemented, user accessible performance monitor registers return a value of 0 when read if MSRP[PMMP] = 1.

[2] When category Embedded.Hypervisor is implemented, guest supervisor accessible performance monitor registers cause an embedded hypervisor privilege exception when read or written in guest supervisor state if MSRP[PMMP] = 1.

3.15.1 Global Control Register 0 (PMGC0/UPMGC0)

PMGC0, shown in Figure 3-102, controls all performance monitor counters. PMGC0 contents are reflected to UPMGC0, which is readable in user mode.

PMGC0 is guest supervisor privileged. UPMGC0 is user privileged.

Writing to PMGC0 requires synchronization.

PMR 400 (PMGC0)
384 (UPMGC0)

PMGC0: Guest supervisor
UPMGC0: User RO

	32	33	34	35 — 63
R W	FAC	PMIE	FCECE	—
Reset	All zeros			

Figure 3-102. Performance Monitor Global Control Register 0 (PMGC0)/ User Performance Monitor Global Control Register 0 (UPMGC0)

This table describes PMGC0 fields.

Table 3-78. PMGC0/UPMGC0 Field Descriptions

Bits	Name	Description
32	FAC	Freeze all counters. When FAC is set by hardware or software, PMLCx[FC] maintains its current value until it is changed by software. FAC can only be cleared by software. 0 The PMCs are incremented (if permitted by other performance monitor control bits). 1 The PMCs are not incremented.
33	PMIE	Performance monitor interrupt enable 0 Performance monitor interrupts are disabled. 1 Performance monitor interrupts are enabled and occur when an enabled condition or event occurs.
34	FCECE	Freeze counters on enabled condition or event 0 PMCs can be incremented (if permitted by other performance monitor control bits). 1 PMCs can be incremented (if permitted by other performance monitor control bits) only until an enabled condition or event occurs. When an enabled condition or event occurs, PMGC0[FAC] is set. It is up to software to clear FAC.
35–63	—	Reserved, should be cleared.

3.15.2 Local Control A Registers (PMLCa*n*/UPMLCa*n*)

The local control A registers 0–15 (PMLCa0–PMLCa15/UPMLCa0–UPMLCa15), shown in Figure 3-103, function as event selectors and provide local control for the corresponding performance monitor counters. PMLCa*x* works with the corresponding PMLCb*x* register. PMLCa*x* contents are reflected to UPMLCa*x*, which is readable in user mode.

PMLCa*n* are guest supervisor privileged. UPMLCa*n* are user privileged.

Writing to PMLCa*n* requires synchronization.

PMR 144 (PMLCa0)
145 (PMLCa1)
...
159 (PMLCa15)

128 (UPMLCa0)
129 (UPMLCa1)
...
143 (UPMLCa15)

PMLCa*n*: Guest supervisor
UPMLCa*n*: User RO

	32	33	34	35	36	37	38	39 47	48 61	62	63
R W	FC	FCS	FCU	FCM1	FCM0	CE	—	EVENT	—	FCGS1	FCGS0
Reset	All zeros										

**Figure 3-103. Local Control A Registers (PMLCa*n*)/
User Local Control A Registers (UPMLCa*n*)**

This table describes PMLCa*n*/UPMLCa*n* fields. PMC*x* refers to the performance monitor counter associated with the PMLCa*x* register (for example, PMC0 is associated with PMLCa0).

Table 3-79. PMLCa*n*/UPMLCa*n* Field Descriptions

Bits	Name	Description
32	FC	Freeze counter 0 The PMC*x* can be incremented (if permitted by other performance monitor control bits). 1 The PMC*x* can not be incremented.
33	FCS	Freeze counter in supervisor state 0 The PMC*x* can be incremented (if permitted by other performance monitor control bits). 1 The PMC*x* can not be incremented if MSR[PR] = 0.
34	FCU	Freeze counter in user state 0 The PMC*x* can be incremented (if permitted by other performance monitor control bits). 1 The PMC*x* can not be incremented if MSR[PR] = 1.
35	FCM1	Freeze counter while mark = 1 0 The PMC*x* can be incremented (if permitted by other performance monitor control bits). 1 The PMC*x* can not be incremented if MSR[PMM] = 1.
36	FCM0	Freeze counter while mark = 0 0 The PMC*x* can be incremented (if permitted by other performance monitor control bits). 1 The PMC*x* can not be incremented if MSR[PMM] = 0.
37	CE	Condition enable 0 PMC*x* overflow conditions cannot occur. (PMC*x* cannot cause interrupts, cannot freeze counters.) 1 Overflow conditions occur when the most-significant-bit of PMC*x* is equal to one. It is recommended that CE be cleared when counter PMC*x* is selected for chaining.
38	—	Reserved, should be cleared.
39–47	EVENT	Event selector. Up to 512 events selectable. Processors may implement fewer number of EVENT bits, but the implemented bits will be in the lower-order bit positions.

Table 3-79. PMLCa*n*/UPMLCa*n* Field Descriptions (continued)

Bits	Name	Description
48–61	—	Reserved, should be cleared.
62	FCGS1	Freeze counters in guest state. <E.HV> 0 The performance counter PMC*x* can be incremented when MSR[GS] = 1 if enabled by all other performance monitor controls.. 1 The performance counter PMC*x* is frozen and cannot be incremented when MSR[GS] = 1.
63	FCGS0	Freeze counters in hypervisor state. <E.HV> 0 The performance counter PMC*x* can be incremented when MSR[GS] = 0 if enabled by all other performance monitor controls.. 1 The performance counter PMC*x* is frozen and cannot be incremented when MSR[GS] = 0.

3.15.3 Local Control B Registers (PMLCb*n*/UPMLCb*n*)

Local control B registers 0–15 (PMLCb0–PMLCa15/UPMLCb0–UPMLCb15), shown in Figure 3-104, specify a threshold value and a multiple to apply to a threshold event selected for the corresponding performance monitor counter. PMLCb*x* works with the corresponding PMLCa*x*. PMLCb*x* contents are reflected to UPMLCb*x*, which is readable in user mode.

PMLCb*n* are guest supervisor privileged. UPMLCb*n* are user privileged.

Writing to PMLCb*n* requires synchronization.

NOTE: Software Considerations

By varying the threshold value, software can profile event characteristics that are subject to thresholding. For example, if PMC1 is configured to count cache misses that last longer than the threshold value, software can measure the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.

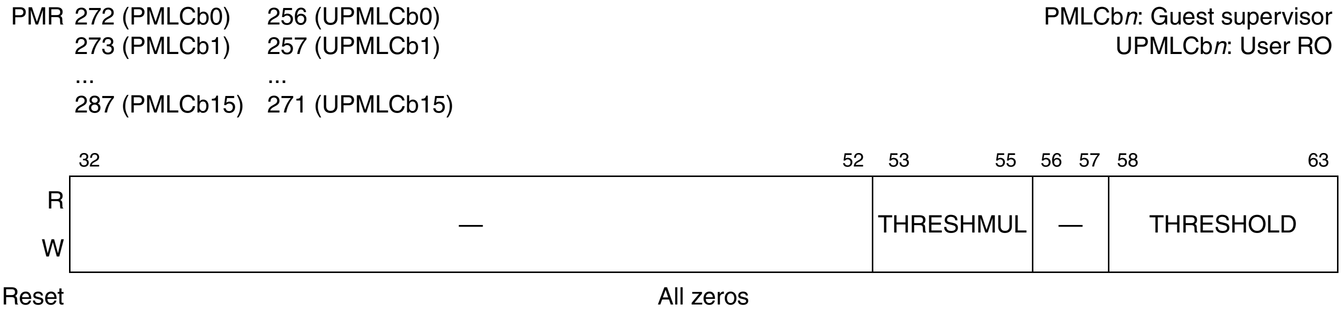

PMR 272 (PMLCb0) 256 (UPMLCb0)
273 (PMLCb1) 257 (UPMLCb1)
... ...
287 (PMLCb15) 271 (UPMLCb15)
PMLCb*n*: Guest supervisor
UPMLCb*n*: User RO
32
52
53
55
56
57
58
63
R
W
—
THRESHMUL
—
THRESHOLD
Reset
All zeros

**Figure 3-104. Local Control B Registers (PMLCb*n*)/
User Local Control B Registers (UPMLCb*n*)**

This table describes PMLCb*n*/UPMLCb*n* fields. PMC*x* refers to the performance monitor counter associated with the PMLCb*x* register (for example, PMC0 is associated with PMLCb0).

Table 3-80. PMLCb*n*/UPMLCb*n* Field Descriptions

Bits	Name	Description
32–52	—	Reserved, should be cleared.
53–55	THRESHMUL	Threshold multiple 000 Threshold field is multiplied by 1 (PMLCb*x*[THRESHOLD] * 1) 001 Threshold field is multiplied by 2 (PMLCb*x*[THRESHOLD] * 2) 010 Threshold field is multiplied by 4 (PMLCb*x*[THRESHOLD] * 4) 011 Threshold field is multiplied by 8 (PMLCb*x*[THRESHOLD] * 8) 100 Threshold field is multiplied by 16 (PMLCb*x*[THRESHOLD] * 16) 101 Threshold field is multiplied by 32 (PMLCb*x*[THRESHOLD] * 32) 110 Threshold field is multiplied by 64 (PMLCb*x*[THRESHOLD] * 64) 111 Threshold field is multiplied by 128 (PMLCb*x*[THRESHOLD] * 128)
56–57	—	Reserved, should be cleared.
58–63	THRESHOLD	Threshold. Only events exceeding this value are counted. Such events, their duration, and the granularity with which the threshold value is interpreted are implementation-dependent. By varying the threshold, software can profile event characteristics. For example, if PMC1 is configured to count cache misses that last longer than the threshold value, software can characterize cache miss durations for a given program by repeatedly monitoring the program with different threshold values.

3.15.4 Performance Monitor Counter Registers (PMC*n*/UPMC*n*)

Performance monitor counter registers (PMC0–PMC15/UPMC0–UPMC15), shown in Figure 3-105, are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter can be enabled to count events selected by the corresponding PMLCa*x*[EVENT] field. PMC*x* works with the corresponding PMLCa*x* and PMLCb*x* registers. PMC*x* contents are reflected to UPMC*x*, which is readable in user mode.

PMC*n* are guest supervisor privileged. UPMC*n* are user privileged.

Writing to PMC*n* requires synchronization.

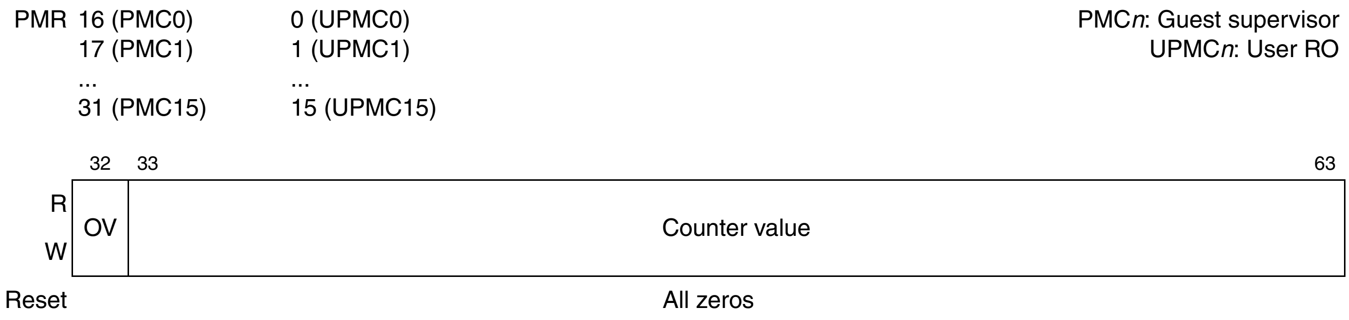


Figure 3-105. Performance Monitor Counter Registers (PMC*n*)/ User Performance Monitor Counter Registers (UPMC*n*)

Table 3-81 describes PMC*n*/UPMC*n* fields. PMCLa*x* refers to the performance monitor local control a register associated with the PMC*x* register (for example, PMC0 is associated with PMLCa0).

Table 3-81. PMC*n*/UPMC*n* Field Descriptions

Bits	Name	Description
32	OV	Overflow. Indicates whether this counter has reached its maximum value. 0 Counter has not reached an overflow state. 1 Counter has reached an overflow state.
33–63	Counter Value	Indicates the number of occurrences of the event specified in PMLCa*x*[EVENT].

Counters overflow when the high-order bit becomes set; that is, they reach the value 2,147,483,648 (0x8000_0000). A performance monitor interrupt handler can easily identify overflowed counters, even if the interrupt is masked for many cycles (during which the counters may continue incrementing).

NOTE: Software Considerations

> Initializing PMCx registers to overflowed values is strongly discouraged. If an overflowed value is loaded into a PMCx that held a non-overflowed value and the PMCx associated controls and interrupt enable conditions are met, the performance monitor interrupt will occur before any events are counted.

The overflow condition is considered to be level-sensitive. If PMCx[OV] = 1 the counter is considered to be in an overflow condition. The response to the overflow condition depends on the configuration as follows:

- If PMLCa*x*[CE] = 0, no special actions occur on overflow; the counter continues incrementing when events occur (and other performance monitor control have not frozen the counter) and no exception is signaled.
- If PMLCa*x*[CE] = 1 and PMCG0[FCECE] = 1, all counters (PMC*n*) are frozen when the counter is in an overflow condition (PMC*x*[OV] = 1).
- If PMLCa*x*[CE] = 1 and PMCG0[PMIE] = 1, a performance monitor exception is signaled when the counter is in an overflow condition (PMC*x*[OV] = 1). A performance monitor interrupt occurs when the exception is signaled and the interrupt is enabled (MSR[EE] = 1 or MSR[GS] = 1 <E.HV>). The exception condition stays signalled until the overflow condition is cleared by software or the PMCG0[PMIE] bit is cleared by software. Since the exception condition is level-sensitive, if the performance monitor interrupt is masked (MSR[EE] = 0 and MSR[GS] = 0 <E.HV>) when the overflow condition is present, and software clears the overflow condition prior to the interrupt becoming unmasked, the interrupt will not occur.

The following sequence is recommended for setting counter values and configurations:

1. Set PMGC0[FAC] to freeze the counters.
2. Using **mtpmr** instructions, initialize counters and configure control registers.
3. Release the counters by clearing PMGC0[FAC] with a final **mtpmr**.

3.16 SPE Registers

The SPE register model includes the following registers:

- Signal processing and embedded floating-point status and control register (SPEFSCR)

- A 64-bit accumulator (ACC)
- A 32-bit extension to the 32-bit GPRs for 32-bit implementations, to create a set of 64-bit registers for use by all SPE vector instructions and by the SPE embedded double-precision floating-point instructions.

These registers are described in the *SPE PEM*.

3.17 AltiVec Registers

The AltiVec technology defines the following registers:

- A set of 32 128-bit vector registers (VRs) to support multiple-element, single-instruction, multiple-data (SIMD) instructions
- Vector status and control register (VSCR)
- Vector save/restore register (VRSAVE). Some previous versions of the architecture used the same SPR number for USPRG0 as for VRSAVE. The USPRG0 mnemonic is deprecated and software should use VRSAVE as defined.

These registers and all register fields pertinent to AltiVec are described in the *AltiVec PEM*.

3.18 Device Control Registers (DCRs)

The EIS defines the existence of a DCR address space and the instructions to access them, but does not define particular DCRs. The on-chip DCRs exist architecturally outside the processor core.

DCRs may control the use of on-chip peripherals. Specific DCR definitions are provided in the core and integrated device reference manuals. DCR numbers are assigned by an implementation. EIS does not architect DCR numbers.

The contents of a DCR can be read into a GPR using **mfdcr r**D**,**DCRN. GPR contents can be written into a DCR using **mtdcr** DCRN**,r**S, where DCRN represents the DCR number.

Not all processor cores implement DCRs.

Chapter 4
Instruction Model

This chapter provides information about the instruction set as it is defined by the Power ISA and the Freescale EIS. It includes the following sections:

- Section 4.1, “Operand Conventions”
- Section 4.2, “Instruction Set Characteristics”
- Section 4.3, “Classes of Instructions”
- Section 4.4, “Forms of Defined Instructions”
- Section 4.5, “Instruction-Related Exceptions”
- Section 4.6, “Instruction Summary”
- Section 4.7, “Instruction Listing”

Full descriptions of each instruction, including opcode, syntax, affected registers, related interrupts, and RTL (register transfer language) are provided in Chapter 5, “Instruction Set”; such descriptions for the VLE, SPE, and AltiVec instructions are provided in their respective programming environments manuals (PEMs).

4.1 Operand Conventions

This section describes operand conventions as they are represented in the architecture. Detailed descriptions are provided of conventions used for storing values in registers and memory, accessing processor registers, and representing data in these registers.

4.1.1 Data Organization in Memory and Data Transfers

Bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Memory operands can be bytes, half words, words, double words, quad words <V>, or, for the load/store multiple instruction type and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

4.1.2 Alignment and Misaligned Accesses

The operand of a single-register memory access instruction has an alignment boundary equal to its length. An operand’s address is misaligned if it is not a multiple of its width.

The concept of alignment is also applied more generally to data in memory. For example, a 12-byte data item is said to be word-aligned if its address is a multiple of four.

Some instructions require their memory operands to have certain alignment. In addition, alignment can affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned.

Instructions are 32 bits (one word) long and must be word-aligned. Note, however, that category VLE provides both 16- and 32-bit instructions and in this case, instructions may be half word-aligned. See the VLE PEM.

This table lists characteristics for memory operands for single-register memory access instructions.

Table 4-1. Address Characteristics of Aligned Operands

Operand	Operand Length	Addr[60–63] if Aligned
Byte (or string)	8 bits	xxxx[1]
Half word	2 bytes	xxx0
Word	4 bytes	xx00
Double word	8 bytes	x000
Quad word	16 bytes	0000

[1] An x in an address bit position indicates that the bit can be 0 or 1 independent of the state of other bits in the address.

An **lmw**, **stmw**, **lbarx** <ER>, **lharx** <ER>, **lwarx**, **ldarx** <64>, **stbcx.**<ER>, **sthcx.**<ER>, **stwcx.,** or **stdcx.** <64> instruction for which the effective address is not properly aligned causes an alignment exception.

4.1.3 Atomic Accesses

A memory access is “single-copy atomic,” or simply “atomic,” if it is always performed in its entirety with no visible fragmentation. Atomic memory accesses are thus serialized: each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors.

Vector accesses are not guaranteed to be atomic. The following other types of single-register accesses are always atomic:

- Byte accesses (all bytes are aligned on byte boundaries)
- Halfword accesses aligned on halfword boundaries
- Word accesses aligned on word boundaries
- Doubleword accesses aligned on doubleword boundaries

No other accesses are guaranteed to be atomic. For example, the access caused by the following instructions is not guaranteed to be atomic:

- Any load or store instruction for which the operand is unaligned
- **lmw**, **stmw**
- Any cache management instruction

An access that is not atomic is performed as a set of smaller disjoint atomic accessed for which the order and the number of times the atomic accesses are performed may vary between implementations.

The results for several combinations of loads and stores to the same or overlapping locations are described below.

1. When two processors execute atomic stores to locations that do not overlap, and no other stores are performed to those locations, the contents of those locations are the same as if the two stores were performed by a single processor.
2. When two processors execute atomic stores to the same storage location, and no other store is performed to that location, the contents of that location are the result stored by one of the processors.
3. When two processors execute stores that have the same target location and are not guaranteed to be atomic, and no other store is performed to that location, the result is some combination of the bytes stored by both processors.
4. When two processors execute stores to overlapping locations, and no other store is performed to those locations, the result is some combination of the bytes stored by the processors to the overlapping bytes. The portions of the locations that do not overlap contain the bytes stored by the processor storing to the location.
5. When a processor executes an atomic store to a location, a second processor executes an atomic load from that location, and no other store is performed to that location, the value returned by the load is the contents of the location before the store or the contents of the location after the store.
6. When a load and a store with the same target location can be executed simultaneously, and no other store is performed to that location, the value returned by the load is some combination of the contents of the location before the store and the contents of the location after the store.

NOTE: Software Considerations

Non atomic memory accesses (particularly misaligned or **lmw**/**stmw** accesses), may be interrupted and parts of the accesses may be performed again after the interrupt returns. Also, the order for which non atomic accesses occur may vary between implementations. For example, a misaligned **lwz** instruction which spans a page boundary for which one of the page translations is not in the TLB, can cause the access to be partially performed to one of the pages and also result in an a data TLB error (for the page translation not in the TLB). When the data TLB error handler establishes the missing TLB entry and returns from the interrupt, the lwz will be re-executed, and the partially performed access will occur again along with the access to the page that previously resulted in a data TLB error.

4.2 Instruction Set Characteristics

Instructions are presented in the following functional categories:

NOTE

- AltiVec instructions are described in the AltiVec PEM.
- VLE instructions are described in the VLE PEM.
- SPE instructions are described in the SPE PEM.

Integer
Arithmetic and logical instructions. See Section 4.6.1.1, “Integer Instructions.”

Load and store
See Section 4.6.1.2, “Load and Store Instructions.” Note that the AltiVec and SPE instructions define additional load store instructions, which are listed in Section 4.7, “Instruction Listing,” and described fully in their respective programming environments manuals.

Floating-point
The Power ISA defines the following two floating-point instruction sets:
The category floating-point instructions, derived from the original PowerPC architecture. described in Section 4.6.1.3, “Floating-Point Instructions <FP>.”
The embedded floating-point vector and scalar arithmetic instructions that are defined as part of category SPE. These instructions are described in the SPE PEM.

Flow control
Branching, CR logical, trap, and other instructions that can alter instruction flow. See Section 4.6.1.8, “Branch and Flow Control Instructions.”

Processor control
Used for determining and controlling processor behavior. See Section 4.6.1.13, “Processor Control Instructions.”

Memory synchronization
Ensure that operations occur in the appropriate context within an out-of-order execution environment. See Section 4.6.1.15, “Memory Synchronization Instructions.”

Memory control
Provide control of caches and TLBs. See Section 4.6.1.17, “Cache Management Instructions.” and Section 4.6.2.2, “Supervisor-Level Memory Control Instructions.”

Note that these groupings do not indicate the execution unit that processes a particular instruction or group of instructions. This information, which is useful for scheduling instructions most effectively, is provided in the instruction timing chapter of the core reference manual.

Integer instructions operate on word or double word operands; floating-point instructions operate on single-precision and double-precision floating-point operands.

The base architecture provides for byte, half-word, word, and double word operand loads and stores between memory and a set of 32 general-purpose registers (GPRs). For base 32-bit implementations, GPRs do not support double-word operands.

It provides for word and double-word operand loads and stores between memory and a set of 32 floating-point registers (FPRs). <FP>

SPE instructions, in either 32- or 64-bit implementations, employ 64-bit GPRs for all vector or double-precision floating-point operations. <SPE>

AltiVec instructions employ a set of thirty-two, 128-bit vector registers (VRs) to support single-instruction, multiple-data (SIMD) operations. <V>

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

<Decorated Storage>:
Depending on the device to which decorated storage instructions address, the device may perform computation directly on a memory location. See the integrated device reference manual.

The description of each instruction includes the mnemonic and a formatted list of operands. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the frequently used instructions; see Appendix C, “Simplified Mnemonics,” for a complete list of simplified mnemonics. Programs written to be portable across the various assemblers for the architecture should not assume the existence of mnemonics not described in that document. Supported simplified mnemonics are also listed in the alphabetical listing of opcodes in Appendix B, “Instruction Set Listings.”

4.3 Classes of Instructions

An instruction falls into exactly one of the following three classes:

- Defined
- Illegal
- Reserved

The class is determined by examining the opcode, and the extended opcode if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction or of a reserved instruction, the instruction is illegal.

4.3.1 Defined Instruction Class

The defined instruction class contains all instruction defined in this document and those defined in other associated programming environments manuals referenced elsewhere in this document.

A defined instruction can have preferred and/or invalid forms, as described in Section 4.4.1, “Preferred Instruction Forms,” and Section 4.4.2, “Invalid Instruction Forms.” Instructions that are part of a category that is not supported are treated as illegal instructions, unless that particular encoding is defined by a category that is supported (in which case it is said to be defined as part of that category).

4.3.2 Illegal Instruction Class

This class of instructions contains the set of instructions described below. Illegal instructions are available for future extensions of the Power ISA; that is, some future version of the Power ISA may define any of these instructions to perform new functions.

Any attempt to execute an illegal instruction causes the system illegal instruction error handler to be invoked and has no other effect.

An instruction consisting entirely of binary 0s is guaranteed always to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized storage results in the invocation of the system illegal instruction error handler. The system illegal instruction handler may also be invoked for instructions that are part of the defined instruction class for which the processor does not implement the instruction.

With the exception of the instruction consisting entirely of binary 0s, the instructions in this class are available for future extensions of the Power ISA; that is, some future version of the Power ISA may define any of these instructions to perform new functions.

The following primary opcodes are illegal:

- 1, 5, 6, 57, 60, 61

The following primary opcodes have unused extended opcodes. All unused extended opcodes are illegal:

- 4, 19, 30, 31, 56, 58, 59, 62, 63

4.3.3 Reserved Instruction Class

Reserved instructions are allocated to specific purposes that are outside the scope of the Power ISA. This class of instructions contains the set of instructions described as follows:

- The instruction having primary opcode 0, except the instruction consisting entirely of binary 0s (which is an illegal instruction; Section 4.3.2, "Illegal Instruction Class") and the extended opcode 256 (Service Processor "Attention")
- Instructions for the Power Architecture® that have not been included in the Power ISA
- Implementation-specific instructions used to conform to the Power ISA specification
- Any other implementation-dependent instructions that are not defined in the Power ISA

Any attempt to execute a reserved instruction results in one of the following:

- Performs the actions described by the implementation if the instruction is implemented
- Causes an illegal instruction exception if the instruction is not implemented.

Instructions that are implementation specific are considered to be a part of the reserved instruction class and are not defined in this document. Any such implementation specific instructions are described in the core reference manual that implements the instructions. Software should carefully consider when using an implementation specific instruction since that instruction may not be implemented in any other processors, or even in a future major revision of a processor.

4.3.4 Reserved Fields and Reserved Values

Reserved fields in instructions are ignored by the processor, although this is being phased into the embedded environment and some processors will not ignore reserved fields and will treat non-zero bits in a reserved field as an illegal instruction.

Some defined instruction fields have reserved values; for example, where an instruction layout contains a fixed value. In such cases all other values for that field are reserved. In general, if an instruction is coded such that a defined field contains a reserved value, the instruction form is invalid (see Section 4.4.2, "Invalid Instruction Forms"). However, that does not apply to portions of defined fields that the instruction description specifies as being treated as reserved.

To maximize compatibility with future architecture extensions, software must ensure that reserved fields in instructions contain zero and that defined fields of instructions do not contain reserved values.

The handling of reserved bits in system registers, such as XER and FPSCR, is implementation-dependent. Unless otherwise stated, software can write any value to such a bit. A subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.

In some cases a defined field of a system register has certain values that are reserved. Software must not set a defined field of a system register to a reserved value.

References elsewhere in this document to a defined field (in an instruction or system register) that has reserved values assume the field does not contain a reserved value, unless otherwise stated or obvious from context.

NOTE: Software Considerations

Assemblers should report uses of reserved values of defined fields of instructions as errors.

NOTE: Software Considerations

Software must preserve reserved bits in system registers because they may be assigned a meaning in a future version of the architecture. To do so in implementation-independent fashion, software should do the following:

- Initialize such registers by supplying zeros for all reserved bits.
- Alter defined bits in the register by reading the register, altering only the desired bits, and then writing the new value back to the register.

However, software can alter XER and FPSCR status bits, preserving the reserved bits by executing instructions that have the side effect of altering the status bits. Similarly, software can alter any defined FPSCR bit by executing an FPSCR instruction. Such instructions are likely to yield better performance than the method described in the second bullet above.

4.4 Forms of Defined Instructions

Defined instructions may have preferred or invalid forms, as described in the following sections.

4.4.1 Preferred Instruction Forms

Some defined instructions have forms that are preferred that execute more efficiently than alternative forms, which may take significantly longer to execute. Instructions with preferred forms are as follows:

- The CR logical instructions
- The load/store multiple instructions
- The OR Immediate instruction (**ori**) (preferred form of no-op)
- The Move To Condition Register Fields (**mtcrf**) instruction
- The TLB Invalidate Virtual Address Indexed (**tlbivax**) instruction.
- The TLB Invalidate Local Indexed (**tlbilx**) instruction <E.HV>.

4.4.2 Invalid Instruction Forms

Some defined instructions can be coded in a form that is invalid. For example, if one or more fields of the instruction, excluding opcode fields, are coded incorrectly in a manner that can be deduced by examining only the instruction encoding.

In general, attempts to execute an invalid form either invoke an illegal instruction exception or yield boundedly undefined results. Any divergence from this is stated in the instruction descriptions. Some older processors may take an unimplemented operation exception instead of an illegal instruction exception.

Some instruction forms are invalid because the instruction contains a reserved value in a defined field (see Section 4.3.4, "Reserved Fields and Reserved Values"); these invalid forms are not discussed further. All other invalid forms are identified in the instruction descriptions.

References to instructions elsewhere in this document assume the instruction form is not invalid, unless otherwise stated or obvious from context.

NOTE: Software Considerations

Assemblers should report uses of invalid instruction forms as errors.

4.5 Instruction-Related Exceptions

Exceptions can be caused directly by the execution of an instruction or by an asynchronous event. Either kind of exception may cause one of several components of the system software to be invoked.

The exceptions that can be caused directly by the execution of an instruction include the following:

- An attempt to execute a reserved-illegal instruction (illegal instruction exception type program interrupt)
- An attempt by an application program to execute a privileged instruction (privileged instruction exception type program interrupt)
- An attempt by a guest supervisor system program to execute a hypervisor privileged instruction (embedded hypervisor privilege interrupt) <E.HV>
- Attempted execution of an instruction that is not provided by the implementation (illegal instruction exception type program interrupt). Some implementations may instead produce an unimplemented operation exception type program interrupt.
- An attempt by an application program to access a privileged SPR (privileged instruction exception type program interrupt)
- An attempt by an application program to access an SPR that does not exist (illegal instruction exception type program interrupt)
- An attempt by a system program to access an SPR that does not exist (boundedly undefined results or illegal instruction exception type program interrupt)
- An attempt by a guest supervisor system program to access an SPR that is hypervisor privileged (embedded hypervisor privilege interrupt) <E.HV>

- The execution of a defined instruction using an invalid form (illegal instruction exception type program interrupt or privileged instruction exception type program interrupt)
- An attempt to access a storage location that is either unavailable (instruction or data TLB error interrupt) or not permitted (instruction or data storage interrupt)
- An attempt to access storage with an effective address alignment not supported by the implementation (alignment interrupt)
- The execution of an **sc** instruction (system call interrupt or embedded hypervisor system call interrupt <E.HV>)
- The execution of a **trap** instruction whose trap condition is met (trap type program interrupt)
- The execution of an **ehpriv** instruction (embedded hypervisor privilege interrupt) <E.HV>
- The execution of a **dnh** instruction (illegal instruction exception type program interrupt or debug halt if enabled by external debugger)
- Attempted execution of a floating-point instruction when floating-point instructions are unavailable (floating-point unavailable interrupt) <FP>
- The execution of a floating-point instruction that causes a floating-point enabled exception to exist (enabled exception type program interrupt) <FP>
- The execution of an instruction which encounters an error reported by hardware, which cannot complete execution without the possibility that incorrect state could be propagated to architectural processor state (error report interrupt).
- Attempted execution of an SPE or embedded floating-point instruction which could modify the upper 32 bits of a GPR when the SPE or embedded floating-point instructions are unavailable (SPE unavailable interrupt) <SP, SP.FV, SP.FD>
- The execution of an embedded floating-point instruction that causes an embedded floating-point data exception to exist (embedded floating-point data interrupt) <SP.FV, SP.FS, SP.FD.>
- The execution of an embedded floating-point instruction that causes an embedded floating-point round exception to exist (embedded floating-point round interrupt) <SP.FV, SP.FS, SP.FD.>
- Attempted execution of a vector instruction when vector instructions are unavailable (AltiVec unavailable interrupt) <V>
- The execution of a vector floating-point instruction that uses a denormalized input value (AltiVec assist interrupt) <V.>
- Execution or attempted execution of an instruction and a precise synchronous debug event occurs (debug interrupt)

Exceptions that can be caused by an asynchronous event are defined by the guest supervisor and hypervisor programming models. Exceptions and the interrupts associated with them are described in Chapter 7, “Interrupts and Exceptions.”

The invocation of the system error handler is precise. Although PowerISA allows for imprecise exceptions for floating-point enabled exceptions, EIS requires that such exceptions be precise and if one of the imprecise modes for invoking the system floating-point enabled exception error handler is in effect (see Table 4-25), the invocation of the system floating-point enabled exception error handler is still handled precisely.

4.5.1 Boundedly Undefined

If instructions are encoded with incorrectly set bits in reserved fields, or attempt execution with operand values that are non sequitur for the operation, the results on execution can be said to be “boundedly undefined.” Boundedly undefined results include any result that is bounded by the current privilege level in relation to instruction execution and memory accesses. For example, if a user-level program executes an incorrectly coded instruction, the resulting undefined results are bounded in that a spurious change from user to supervisor state is not allowed and the level of privilege exercised by the program in relation to memory access and other system resources cannot be exceeded. Boundedly undefined results for a given instruction can vary between implementations and between execution attempts in the same implementation.

4.5.2 Instruction Forms

This section describes preferred instruction forms, addressing modes, and synchronization.

4.5.2.1 Preferred Instruction Forms (no-op)

The OR Immediate (**ori**) instruction has the following preferred form for expressing a no-op:

- **ori 0,0,0**

There are other forms of no-op which produce hints to the processor for improving performance and use of these hints inappropriately when a true no-op is needed may reduce performance

4.5.2.2 Invalid Instruction Forms

Some of the defined instructions have invalid forms. An instruction form is invalid if one or more fields of the instruction, excluding the opcode field(s), are coded incorrectly in a manner that can be deduced by examining only the instruction encoding.

Attempts to execute an invalid form of an instruction either causes an illegal instruction exception or yields boundedly undefined results. Any exceptions to this rule are stated in the instruction descriptions.

Some kinds of invalid form instructions can be deduced just from examining the instruction layout. These are listed below.

- Field shown as reserved but coded as nonzero
- Field shown as containing a particular value but coded as some other value

These invalid forms are not discussed further.

Other invalid instruction forms can be deduced by detecting an invalid encoding of one or more of the instruction operand fields. These kinds of invalid form are identified in the instruction descriptions.

- Branch conditional and branch conditional extended instructions (undefined encoding of BO field)
- Load with update instructions (**r**D = **r**A or **r**A = 0)
- Store with update instructions (**r**A = 0)
- Load multiple instruction (**r**A or **r**B in range of registers to be loaded)
- Load string immediate instructions (**r**A in range of registers to be loaded)

- Load string indexed instructions (**r**D = **r**A or **r**D = **r**B)
- Load/store floating-point with update instructions (**r**A = 0)

4.5.3 Addressing Modes

This section describes conventions for addressing memory and for calculating effective addresses (EAs).

4.5.3.1 Memory Addressing

A program references memory using the effective address computed by the processor when it executes a memory access or branch instruction (or certain other instructions described in Section 4.6.1.17.2, "User-Level Cache Management Instructions," and Section 4.6.2.2.1, "Supervisor-Level Cache Management Instructions") or when it fetches the next sequential instruction.

4.5.3.2 Memory Operands

Bytes in memory are numbered consecutively starting with zero. Each number is the address of the corresponding byte.

Memory operands may be bytes, half words, words, double words, or, for the load/store multiple load/store instructions, a sequence of words or bytes. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Byte ordering can be big or little endian (see Section 4.5.3.4, "Byte Ordering").

Operand length is implicit for each instruction with respect to memory alignment. The operand of a scalar memory access instruction has a natural alignment boundary equal to the operand length. In other words, the natural address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise, it is said to be misaligned. For more information about alignment, see Section 4.1.2, "Alignment and Misaligned Accesses."

4.5.3.3 Effective Address Calculation

The address computed by the processor when executing a memory access or branch instruction (or certain other instructions described in Section 4.6.1.17.2, "User-Level Cache Management Instructions," and Section 4.6.2.2, "Supervisor-Level Memory Control Instructions"), or when fetching the next sequential instruction, is called the effective address (EA) and specifies a byte in memory. For a memory access instruction, if the sum of the EA and the operand length exceeds the maximum EA, the memory access is considered to be undefined.

In 32-bit mode, the EA calculation is performed, and the upper 32 bits of the 64-bit EA are ignored and are treated as zero. In 64-bit mode, the result of the EA calculation is a 64-bit result.

Effective address arithmetic, except for next sequential instruction address computations, wraps around from the maximum address, 2^{32}– 1 in 32-bit mode and 2^{64}– 1 in 64-bit mode, to address 0.

For the purposes of describing data and instruction addressing modes, let m = 0 for 64-bit mode and m = 32 for 32-bit mode.

4.5.3.3.1 Data Memory Addressing Modes

The architecture supports the following data memory addressing modes:

- Base+displacement addressing mode—The 16-bit D field is sign-extended and added to the contents of the GPR designated by **r**A or to zero if **r**A = 0. Instructions that use this addressing mode are of the D instruction format.
- Base+index addressing mode—The contents of the GPR designated by **r**B are added to the contents of the GPR designated by **r**A or to zero if **r**A = 0. Instructions that use this addressing mode are of the X instruction format.
- Base+displacement extended addressing mode—The 12-bit DE field is sign-extended and added to the contents of the GPR designated by **r**A or to zero if **r**A = 0. Instructions that use this addressing mode are of the DE instruction format.
- Base+displacement extended scaled addressing mode—The 12-bit DES field is concatenated on the right with zeros, sign-extended, and added to the contents of the GPR designated by **r**A or to zero if **r**A = 0. Instructions that use this addressing mode are of the DES instruction format.

<category VLE>
VLE offers additional instruction formats to provide EA calculation.

Additional extensions to the ISA may provide additional addressing modes.

4.5.3.3.2 Instruction Memory Addressing Modes

Instruction memory addressing modes correspond with instructions forms, as follows:

Table 4-2. Instruction Memory Addressing Modes

Instruction Form	Instruction Memory Addressing Mode
I-form branch instructions	The 24-bit LI field is concatenated on the right with 0b00, sign-extended, and added either to the address of the branch instruction if AA = 0, or to 0 if AA = 1.
Taken B-form branch instructions	The 14-bit BD field is concatenated on the right with 0b00, sign-extended, and added either to the address of the branch instruction if AA = 0, or to 0 if AA = 1.
Taken XL-form branch instructions	The contents of bits LR[m–61] or CTR[m–61] are concatenated on the right with 0b00. The contents of bits LR[m–62] or CTR[m–62] are concatenated on the right with 0b0.<VLE>

Table 4-2. Instruction Memory Addressing Modes (continued)

Instruction Form	Instruction Memory Addressing Mode
Sequential instruction fetching (or non-taken branch instructions)	The value 4 is added to the address of the current instruction to form the EA of the next instruction. If the address of the current instruction is 0xFFFF_FFFC in 32-bit mode or 0xFFFF_FFFF_FFFF_FFFC in 64-bit mode, the address of the next sequential instruction is undefined.
Any branch instruction with LK = 1	The value 4 is added to the address of the current instruction and the result is placed into LR. If the address of the current instruction is 0xFFFF_FFFC in 32-bit mode or 0xFFFF_FFFF_FFFF_FFFC in 64-bit mode, the result placed into LR is undefined. Although some implementations may support next sequential instruction address computations wrapping from the highest address to 0 as part of the instruction flow, depending on this behavior may reduce the portability of their software. If code must span this boundary, software should place a non-linking branch at address 0xFFFF_FFFC in 32-bit mode or 0xFFFF_FFFF_FFFF_FFFC in 64-bit mode, which always branches to address 0 (either absolute or relative branches work).

4.5.3.4 Byte Ordering

If scalars (individual data items and instructions) were indivisible, there would be no such concept as byte ordering. It is meaningless to consider the order of bits or groups of bits within the smallest addressable unit of memory, because nothing can be observed about such order. Only when scalars, which the programmer and processor regard as indivisible quantities, can consist of more than one addressable unit of memory does the question of order arise.

For a machine in which the smallest addressable unit of memory is the 64-bit double word, there is no question of the ordering of bytes within double words. All transfers of individual scalars between registers and memory are of double words, and the address of the byte containing the high-order 8 bits of a scalar is no different from the address of a byte containing any other part of the scalar.

<category Vector>:
For EIS, a category Vector instruction that accesses memory which is marked as little-endian byte ordering is undefined.

For the PowerISA, as for most current computer architectures, the smallest addressable unit is the 8-bit byte. Many scalars are half words, words, and double words that consist of groups of bytes. When a word-length scalar is stored from a register to memory, the scalar occupies four consecutive byte addresses. It thus becomes meaningful to discuss the order of the byte addresses with respect to the value of the scalar: which byte contains the highest-order byte, which contains the next-highest, and so on.

Given a multiple-byte scalar, the choice of ordering is essentially arbitrary. There are 4! = 24 ways to specify the ordering of four bytes within a word, but only two of these orderings are sensible:

- Big-endian ordering assigns the lowest address to the highest-order (leftmost) 8 bits of the scalar, the next sequential address to the next highest-order 8 bits, and so on. That is, the big (most-significant) end of the scalar, considered as a binary number, comes first in memory.
- Little-endian ordering assigns the lowest address to the lowest-order (rightmost) 8 bits, the next sequential address to the next lowest-order 8 bits, and so on. That is, the little (least-significant) end of the scalar, considered as a binary number, comes first in memory.

The byte ordering is configured for each page through the TLBs. See Section 6.5.6, “Permission Attributes,” and Section 6.4.1, “Memory/Cache Access Attributes (WIMGE Bits).”

4.5.4 Synchronization

Changing a value in certain system registers and invalidating TLB entries can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed. For example, changing MSR[IS] from 0 to 1 has the side effect of changing address space. These effects need not occur in program order (that is, the strict order in which they occur in the program) and may require explicit synchronization by software.

Context synchronization is achieved by post- and presynchronizing instructions. An instruction is presynchronized by completing all instructions before dispatching the presynchronized instruction. Post-synchronizing is implemented by not dispatching any later instructions until the post-synchronized instruction is completely finished.

4.5.4.1 Memory Synchronization

Memory synchronization means that accesses to memory are properly ordered. Memory accesses may occur out of order in a weakly ordered memory system and memory synchronization provides a method to ensure that specified accesses occur in a specified order, or that accesses performed on one processor are completed before other instructions are initiated which may depend on those memory accesses being complete (with respect to the processor that is both performing the memory access and which such memory accesses must appear to complete before other instructions are initiated). Memory synchronization is performed by the **sync** and **mbar** instructions.

The **sync** 0 instruction provides a memory barrier throughout the memory hierarchy. It waits for preceding data memory accesses to reach the point of coherency (that is, visible to the entire memory hierarchy); then it is broadcast. No subsequent instructions in the stream are initiated until after **sync** 0 completes. Note that **sync** 0 uses the same opcode as the previously defined **msync** instruction. Both **msync** and **sync** (no operand) should be treated as an extended mnemonic for **sync** 0.

The **mbar** instruction also provides a memory barrier, but does not guarantee that the storage operations are complete before other instructions are initiated. Implementations may provide additional operand values for **mbar** that provide ordering to a subset of memory accesses. See Section 6.4.8, “Shared Memory.”

The **sync** instruction is used to provide ordering of memory accesses and operands other than zero can be used to perform less stringent (and thus higher performance) ordering. The **sync** 0 instruction is also used to ensure that memory access have all been completed on the processor executing the **sync** 0.

The **sync** instruction is described in Section 4.6.1.15, “Memory Synchronization Instructions.”

See Section 6.4.8.1, “Memory Access Ordering,” for detailed information.

4.5.4.2 Instruction Synchronization

Instruction synchronization means that instruction fetches to memory occur in the context synchronized by a context synchronizing instruction; that is, when changes are made to the context in which instructions are fetched (such as address space), those context changes are not guaranteed to occur until the instructions are synchronized. Instruction synchronization is performed when a context synchronizing instruction is executed and completes. Context synchronizing instructions are as follows:

- **sc**
- **isync**
- **rfi**
- **rfgi** <E.HV>
- **rfci**
- **rfdi** <E.ED>
- **rfmci**

Implementations may define other instructions as context synchronizing; however, software should not depend on such behavior if it is expected to run on other implementations.

In general, when executed, a context synchronizing instruction discards previously fetched instructions (which may have translated or produced other effects in the previously established context), and the instructions are re-fetched (and retranslated) once the context synchronizing instruction has completed.

Interrupts are also context synchronizing, and begin fetching instructions at the location prescribed by the interrupt vector.

CAUTION

Care must be taken when altering context associated with instruction fetch and instruction address translation. Altering MSR[IS], MSR[GS] <E.HV>, MSR[CM] <64>, LPIDR <E.HV>, and PID*n* can cause an implicit branch, where the change in translation or how instructions are fetched causes the processor to fetch instructions from a different real address than what would have resulted if the context was not changed.

Even though a single **mtmsr** instruction can change multiple context bits, those bits which are changed can occur individually and in any order until a context synchronizing instruction is executed. Implicit branches are not supported by the architecture and therefore software must ensure that all possible permutations of the previous bits in the MSR and the new bits established in the MSR by **mtmsr** all translate to the same physical address.

NOTE: Software Considerations

“Inline” changes, which occur when **mtmsr** or **mtspr** is used to change the context of an instruction fetch address translation (which affects the current instruction stream that is changing the context), are discouraged and should be avoided. This is because changes to the address space can cause an implicit branch. Use a return from interrupt instruction to atomically change both MSR context and execution address.

For more information on SPE load and store instructions, see the SPE PEM. <SPE>

4.5.4.2.1 Self-Modifying Code

When a processor modifies any memory location that can contain an instruction, software must ensure that the instruction cache is made consistent with data memory and that the modifications are made visible to the instruction fetching mechanism. This must be done even if the cache is disabled or if the page is marked caching-inhibited.

The following instruction sequence can be used to accomplish this when the instructions being modified are in memory that is memory-coherence required and one processor both modifies the instructions and executes them. (Additional synchronization is needed when one processor modifies instructions that another processor will execute.)

The following sequence synchronizes the instruction stream after software has performed the memory accesses to store the new instructions. In this sequence, location ‘addr’ is assumed to contain modified instructions (new instructions)

```
dcbst   addr    // Force modified instructions from data cache to memory
                // -- dcbst must be performed to each cache line containing
                // new instructions
sync            // order dcbst with icbi to ensure data pushed to memory
icbi    addr    // remove (invalidate) old instructions in instruction cache
                // -- icbi must be performed to each cache line containing
                // new instructions
sync            // ensure the ICBI invalidate is complete
isync           // ensure any old prefetched instructions are refetched
```

These operations are required because the data cache is a write-back cache. Because instruction fetching uses the instruction cache and the instruction cache is not kept coherent by hardware with the data cache, changes to items in the data cache cannot be reflected in the instruction cache and the current instruction pipeline. The **dcbst** (or **dcbf**) forces modified cache lines to be written to memory. The subsequent **icbi** invalidates the instruction cache lines, removing them from the instruction cache. The **isync** then removes any instructions that may have been prefetched and forces them to be refetched. Since the old instructions were removed from the instruction cache, when the processor fetches instructions they will miss in the instruction cache and will be loaded from memory. The **sync** operations ensure the order of the new instructions being written to memory, the invalidation of the instruction cache, and that the instruction cache has been invalidated before fetching new instructions.

Special care must be taken to avoid coherency paradoxes in systems that implement unified secondary caches, and designers should carefully follow the guidelines for maintaining cache coherency discussed in the user’s manual.

NOTE: Software Considerations

Because the optimal instruction sequence may vary between systems, many operating systems provide a system service to perform the function described above.

4.5.4.3 Synchronization Requirements

This section discusses synchronization requirements for special registers, certain instructions, and TLBs. The synchronization described in this section refers to the state of the processor that is performing the synchronization.

NOTE: Software Considerations

The information in this section describes synchronization requirements at an architectural level. More specific information can be found in the core reference manual.

An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed, is called a context-altering instruction. This section covers all of the context-altering instructions as well as instructions that require special synchronization. The software synchronization required for each is shown in Table 4-3, Table 4-4, and Table 4-5. Instructions that are not listed do not require explicit synchronization.

The notation "CSI" in the tables means any context-synchronizing instruction (**sc**, **isync**, **rfi**, **rfgi** <E.HV>, **rfci**, **rfdi** <E.ED>, or **rfmci**). A context-synchronizing interrupt (that is, any interrupt) can be used instead of a context-synchronizing instruction; if so, references in this section to the synchronizing instruction should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required, either before or after a context-altering instruction, the phrase "the synchronizing instruction before (or after) the context-altering instruction" should be interpreted as meaning the context-altering instruction itself.

The synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. The synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

If a sequence of instructions contains context-altering instructions and contains no instructions that are affected by any of the context alterations, no software synchronization is required within the sequence.

No software synchronization is required before altering the MSR because **mtmsr** is execution synchronizing.

NOTE: Software Considerations

Sometimes advantage can be taken of certain instructions that occur naturally in the program, such as the **rfi** at the end of an interrupt handler, which provides the required synchronization.

This table identifies the software synchronization requirements for data access for all context-altering instructions.

Table 4-3. Data Access Synchronization Requirements

Context Altering Instruction or Event	Required Before	Required After	Notes
mfspr (L1CSR0, L1CSR1)	**sync**	None	[1]
mtmsr (CM) <64>	None	CSI	—
mtmsr (DE)	None	CSI	—
mtmsr (DS)	None	CSI	—

Table 4-3. Data Access Synchronization Requirements (continued)

Context Altering Instruction or Event	Required Before	Required After	Notes
mtmsr (GS) <E.HV>	None	CSI	—
mtmsr (ME)	None	CSI	[2]
mtmsr (PR)	None	CSI	—
mtpmr (all)	None	CSI	—
mtspr (EPLC) <E.EP>	None	CSI	—
mtspr (EPSC) <E.EP>	None	CSI	—
mtspr (L1CSR0, L1CSR1)	**sync** followed by **isync**	**isync**	—
mtspr (L1CSR2)	**sync** followed by **isync**	**isync** followed by **sync**[3]	—
mtspr (L2CSR0)	**sync** followed by **isync**	**isync**	—
mtspr (L2CSR1)	**sync** followed by **isync**	**isync** followed by **sync**[3]	—
mtspr (LPIDR) <E.HV>	CSI	CSI	—
mtspr (PID, PID1, PID2)	CSI	CSI	—
tlbivax	CSI	**sync** followed by CSI	[4,5,6]
tlbilx <E.HV>	CSI	CSI	[4,5]
tlbwe	CSI	CSI	[4,5]

[1] A **sync** prior to reading L1CSR0 or L1CSR1 is required to examine any cache locking status from prior cache locking operations. The **sync** ensures that any previous cache locking operations have completed prior to reading the status.

[2] A context-synchronizing instruction is required after altering MSR[ME] to ensure that the alteration takes effect for subsequent machine check interrupts, which may not be recoverable and therefore may not be context-synchronizing.

[3] The additional **sync** after the **mtspr** is done is required if software is turning off stashing by setting the stash ID field of the register to zero. The **sync** ensures that any pending stash operations have finished.

[4] For data accesses, the context-synchronizing instruction before **tlbwe**, **tlbilx** <E.HV>, or **tlbivax** ensures that all memory accesses due to preceding instructions have completed to a point at which they have reported all exceptions they cause.

[5] The context-synchronizing instruction after **tlbwe**, **tlbilx** <E.HV>, or **tlbivax** ensures that subsequent accesses (data and instruction) use the updated value in any TLB entries affected. It does not ensure that all accesses previously translated by TLB entries being updated have completed with respect to memory; if these completions must be ensured, **tlbwe**, **tlbilx** <E.HV>, or **tlbivax** must be followed by a **sync** and by a context-synchronizing instruction.
The following sequence shows why data accesses must ensure that all storage accesses due to instructions before a **tlbwe** or **tlbivax** have completed to a point at which they have reported all exceptions they would cause. Assume that valid TLB entries exist for the target location when the sequence starts.

1.A program issues a load or store to a page.
2. The same program executes a **tlbwe** or **tlbivax** that invalidates the corresponding TLB entry.
3. The load or store instruction finally executes and gets a TLB miss exception that is semantically incorrect. To prevent it, a context-synchronizing instruction must be executed between steps 1 and 2.

[6] To insure that all TLB invalidations are completed and seen in all processors in the coherence domain, the global invalidation requires that a **tlbsync** be executed after the **tlbivax** as follows: **tlbivax**; **sync**; **tlbsync**; **sync**; **isync**. This code should be protected by a mutual exclusion lock such that only one processor at a time is executing this sequence as multiple simultaneous **tlbsync** operations may cause the integrated device to hang.

This table identifies the software synchronization requirements for instruction fetch and/or execution for context-altering instructions which require synchronization.

Table 4-4. Instruction Fetch and/or Execution Synchronization Requirements

Context Altering Instruction or Event	Required Before	Required After	Notes
mtmsr (CM) <64>	None	CSI	—
mtmsr (DE)	None	CSI	—
mtmsr (FE0) <FP>	None	CSI	—
mtmsr (FE1) <FP>	None	CSI	—
mtmsr (FP) <FP>	None	CSI	—
mtmsr (IS)	None	CSI	—
mtmsr (GS) <E.HV>	None	CSI	—
mtmsr (PR)	None	CSI	—
mtpmr (all)	None	CSI	—
mtspr (L1CSR0, L1CSR1)	**sync** followed by **isync**	**isync**	—
mtspr (L2CSR0, L2CSR1)	**sync** followed by **isync**	**isync**	—
mtspr (LPIDR) <E.HV>	None	CSI	—
mtspr (MAS*n*)	None	**isync**	[1]
mtspr (PID, PID1, PID2)	None	CSI	—
tlbivax	None	CSI	[2,3]
tlbilx <E.HV>	None	CSI	[2]
tlbwe	None	CSI	[2]

[1] MAS registers changes require an **isync** before subsequent instructions that use those updated values such as a **tlbwe**, **tlbre**, **tlbilx**, **tlbsx**, **tlbilx** <E.HV>, and **tlbivax**. Typically software will do several MAS updates and then perform a single **isync** prior to executing the TLB management instruction.

[2] The context-synchronizing instruction after **tlbwe**, **tlbilx** <E.HV>, or **tlbivax** ensures that subsequent accesses (data and instruction) use the updated value in any TLB entries affected. It does not ensure that all accesses previously translated by TLB entries being updated have completed with respect to memory; if these completions must be ensured, **tlbwe**, **tlbilx** <E.HV>, or **tlbivax** must be followed by an **sync** and by a context-synchronizing instruction.

[3] To ensure that all TLB invalidations are completed and seen in all processors in the coherence domain, the global invalidation requires that a **tlbsync** be executed after the **tlbivax** as follows: **tlbivax**; **sync**; **tlbsync**; **sync**; **isync**. This code should be protected by a mutual exclusion lock such that only one processor at a time is executing this sequence as multiple simultaneous **tlbsync** operations may cause the integrated device to hang.

This table identifies the software synchronization requirements for non context-altering instructions that require synchronization.

Table 4-5. Special Synchronization Requirements

Context Altering Instruction or Event	Required Before	Required Immediately After	Notes
mtspr (BUCSR)	None	**isync**	—
mtspr (DAC*n*)	None	**isync** followed by changing MSR[DE] from 0 to 1	[1]
mtspr (DBCR*n*)	None	**isync** followed by changing MSR[DE] from 0 to 1	[1]
mtspr (DBSR)	None	**isync** followed by changing MSR[DE] from 0 to 1	[1]
mtspr (DBSRWR) <E.HV>	None	**isync** followed by changing MSR[DE] from 0 to 1	[1]
mtspr (DVC*n*)	None	**isync** followed by changing MSR[DE] from 0 to 1	[1]
mtspr (EPCR[DUVD]) <E.HV>	None	**isync** followed by changing MSR[DE] from 0 to 1	[1,2]
mtspr (HID*n*)	**msync** followed by **isync**	**isync**	—
mtspr (IAC*n*)	None	**isync** followed by changing MSR[DE] from 0 to 1	[1]
mtspr (L2ERR*)	**msync** followed by **isync**	**isync**	—
mtspr (MMUCSR0)	None	**isync**	—
mtspr (NSPD)	None	**isync**	—

[1] Synchronization requirements for changing any debug facility registers require that the changes be followed by an **isync** and a transition of MSR[DE] from 0 to 1 before the results of the changes are guaranteed to be seen. Normally changes to the debug registers will occur in the debug interrupt routine when MSR[DE] is 0 and the subsequent return from the debug routine is likely to set MSR[DE] back to 1, which accomplishes the required synchronization. Software should only make changes to the debug facility registers when MSR[DE] = 0.

[2] Note that the special synchronization requirement applies only to changes to EPCR[DUVD]. If this bit is not changed, the synchronization requirements for EPCR is as described in the data or instruction execution tables above.

4.5.4.4 Context Synchronization

Context-synchronizing operations include instructions **isync**, **sc**, **rfi**, **rfgi**<E.HV>, **rfci**, **rfdi**, and **rfmci**, and most interrupts. An instruction or event is context-synchronizing if it satisfies the following requirements:

1. The operation is not initiated or, in the case of **isync**, does not complete until all instructions already in execution have completed to a point at which they have reported all exceptions they cause.
2. The instructions that precede the operation complete execution in the context (including such parameters as privilege level, address space, and memory protection) in which they were initiated.

3. If the operation is an interrupt or directly causes one (for example, **sc** directly causes a system call interrupt), the operation is not initiated until no interrupt-causing exception exists having higher priority than the exception associated with the interrupt. See Section 7.11, “Exception Priorities.”
4. The sequential execution model requires that instructions after the operation are fetched and executed in the context established by the operation. This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them speculatively may also be discarded, except as described in Section 6.4.8.1, “Memory Access Ordering.”

As described in Section 4.5.4.5, “Execution Synchronization,” a context-synchronizing operation is necessarily execution synchronizing. Unlike **sync** and **mbar**, such operations do not affect the order of memory accesses with respect to other mechanisms.

4.5.4.5 Execution Synchronization

An instruction is execution synchronizing if it satisfies items 1 and 2 of the definition of context synchronization (see Section 4.5.4.4, “Context Synchronization”). The **sync** instruction is treated like **isync** with respect to item 1 (that is, the conditions described in item 1 apply to completion of **sync**). Execution synchronizing instructions are **sync**, **mtmsr**, **wrtee**, and **wrteei**. All context-synchronizing instructions are execution-synchronizing.

Unlike a context-synchronizing operation, an execution-synchronizing instruction need not ensure that the instructions following it execute in the context established by that execution-synchronizing instruction. This new context becomes effective after the execution-synchronizing instruction completes and before or at a subsequent context-synchronizing operation.

4.6 Instruction Summary

Note that some instructions have record and/or overflow forms that have the following features:

- CR update for integer instructions—The dot (**.**) suffix on the mnemonic for integer computation instructions enables the update of the CR0 field. CR0 is updated based on the signed comparison of the result to 0. In 32-bit mode the results of the lower 32-bits of the result are compared to 0. In 64-bit mode the results of all 64-bits are compared to 0 <64>.
- Integer overflow option—The **o** suffix indicates that the overflow bit in the XER is enabled. In 32-bit mode, overflow (XER[OV]) is set if the carryout of bit 32 is not equal to the carryout of bit 33 in the final result of the operation. In 64-bit mode, overflow (XER[OV]) is set if the carryout of bit 0 is not equal to the carryout of bit 1 in the final result of the operation <64>. Summary overflow (XER[SOV]) is a sticky bit that is set when XER[OV] is set.
- CR update for floating-point instructions—The dot (**.**) suffix on the mnemonic for floating-point computation instructions enables the update of the CR1 field. CR1 is updated with the exception status copied from bits FPSCR[32:35]. <FP>
- CR update for vector instructions—The dot (**.**) suffix on the mnemonic for vector comparison instructions enables the update of the CR6 field. CR6 is updated based on whether the result is all zeros or all ones. <V>
- CR update for store conditional instructions —Store conditional instructions always include the dot (**.**) suffix and update CR0 based on whether the store was performed.

4.6.1 User-Level Instructions

This section discusses the user-level instructions defined in the architecture.

4.6.1.1 Integer Instructions

This section describes the user-level integer instructions. These consist of the following:

- Integer arithmetic instructions
- Integer compare instructions
- Integer logical instructions
- Integer rotate and shift instructions

Integer instructions use the content of the GPRs as source operands and place results into GPRs and the XER and CR fields.

4.6.1.1.1 Integer Arithmetic Instructions

This table lists the integer arithmetic instructions.

Table 4-6. Integer Arithmetic Instructions

Name	Mnemonic	Syntax
Add	**add** (**add. addo addo.**)	**r**D,**r**A,**r**B
Add Carrying	**addc** (**addc. addco addco.**)	**r**D,**r**A,**r**B
Add Extended	**adde** (**adde. addeo addeo.**)	**r**D,**r**A,**r**B
Add Immediate	**addi**	**r**D,**r**A,SIMM
Add Immediate Carrying	**addic**	**r**D,**r**A,SIMM
Add Immediate Carrying and Record	**addic.**	**r**D,**r**A,SIMM
Add Immediate Shifted	**addis**	**r**D,**r**A,SIMM
Add to Minus One Extended	**addme** (**addme. addmeo addmeo.**)	**r**D,**r**A
Add to Zero Extended	**addze** (**addze. addzeo addzeo.**)	**r**D,**r**A
Divide Doubleword <64>	**divd** (**divd. divdo divdo.**)	**r**D,**r**A,**r**B
Divide Doubleword Unsigned <64>	**divdu divdu. divduo divduo.**	**r**D,**r**A,**r**B
Divide Word	**divw** (**divw. divwo divwo.**)	**r**D,**r**A,**r**B
Divide Word Unsigned	**divwu divwu. divwuo divwuo.**	**r**D,**r**A,**r**B
Multiply High Doubleword <64>	**mulhd** (**mulhd.**)	**r**D,**r**A,**r**B
Multiply High Doubleword Unsigned <64>	**mulhdu** (**mulhdu.**)	**r**D,**r**A,**r**B
Multiply High Word	**mulhw** (**mulhw.**)	**r**D,**r**A,**r**B
Multiply High Word Unsigned	**mulhwu** (**mulhwu.**)	**r**D,**r**A,**r**B
Multiply Low Immediate	**mulli**	**r**D,**r**A,SIMM
Multiply Low Doubleword <64>	**mulld** (**mulld. mulldo mulldo.**)	**r**D,**r**A,**r**B
Multiply Low Word	**mullw** (**mullw. mullwo mullwo.**)	**r**D,**r**A,**r**B
Negate	**neg** (**neg. nego nego.**)	**r**D,**r**A
Subtract From	**subf** (**subf. subfo subfo.**)	**r**D,**r**A,**r**B
Subtract from Carrying	**subfc** (**subfc. subfco subfco.**)	**r**D,**r**A,**r**B
Subtract from Extended	**subfe** (**subfe. subfeo subfeo.**)	**r**D,**r**A,**r**B
Subtract from Immediate Carrying	**subfic**	**r**D,**r**A,SIMM
Subtract from Minus One Extended	**subfme** (**subfme. subfmeo subfmeo.**)	**r**D,**r**A
Subtract from Zero Extended	**subfze** (**subfze. subfzeo subfzeo.**)	**r**D,**r**A

Although there is no subtract immediate instruction, its effect can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation. Subtract instructions subtract the second operand (**r**A) from the third operand (**r**B). Simplified mnemonics are provided in which the third operand is subtracted from the second. For examples, see Section C.2, “Subtract Simplified Mnemonics.”

An implementation that executes instructions with the overflow exception enable bit (OE) set or that sets the carry bit (CA) can either execute these instructions slowly or prevent execution of the subsequent instruction until the operation completes. The summary overflow (SO) and overflow (OV) bits in the XER are set to reflect an overflow condition of a 32-bit result or a 64-bit result <64> based on computation mode only if the instruction's OE bit is set.

4.6.1.1.2 Integer Compare Instructions

Integer compare instructions algebraically or logically compare the contents of register **r**A with either the zero-extended value of the UIMM operand, the sign-extended value of the SIMM operand, or the contents of **r**B. The comparison is signed for **cmpi** and **cmp** and unsigned for **cmpli** and **cmpl**.

This table lists integer compare instructions. The L bit can be either 0 (for a 32-bit compare) or 1 (for a 64-bit compare) regardless of the computation mode. Note that the L bit must be 0 for 32-bit implementations.

Table 4-7. Integer Compare Instructions

Name	Mnemonic	Syntax
Compare	**cmp**	**cr**D,L,**r**A,**r**B
Compare Immediate	**cmpi**	**cr**D,L,**r**A,SIMM
Compare Logical	**cmpl**	**cr**D,L,**r**A,**r**B
Compare Logical Immediate	**cmpli**	**cr**D,L,**r**A,UIMM

The **cr**D operand can be omitted if the result of the comparison is to be placed in CR0; otherwise, the target CR field must be specified in **cr**D by using an explicit field number.

For information on simplified mnemonics for the integer compare instructions, see Section C.5, "Compare Word Simplified Mnemonics."

4.6.1.1.3 Integer Logical Instructions

The logical instructions shown in the following table perform bit-parallel operations on the specified operands. Logical instructions with the CR updating enabled (uses dot suffix) and instructions **andi.** and **andis.** set CR field CR0 to characterize the result of the logical operation by comparison of the result to 0. Logical instructions do not affect XER[SO], XER[OV], or XER[CA].

Table 4-8. Integer Logical Instructions

Name	Mnemonic	Syntax
AND	**and** (**and.**)	**r**A,**r**S,**r**B
AND Immediate	**andi.**	**r**A,**r**S,UIMM
AND Immediate Shifted	**andis.**	**r**A,**r**S,UIMM
AND with Complement	**andc** (**andc.**)	**r**A,**r**S,**r**B
Bit Permute Doubleword <64>	**bpermd**	**r**A,**r**S,**r**B
Compare Bytes	**cmpb**	**r**A,**r**S,**r**B

Table 4-8. Integer Logical Instructions (continued)

Name	Mnemonic	Syntax
Count Leading Zeros Word	**cntlzw** (**cntlzw.**)	rA,rS
Count Leading Zeros Doubleword <64>	**cntlzd** (**cntlzd.**)	rA,rS
Equivalent	**eqv** (**eqv.**)	rA,rS,rB
Extend Sign Byte	**extsb** (**extsb.**)	rA,rS
Extend Sign Half Word	**extsh** (**extsh.**)	rA,rS
Extend Sign Word <64>	**extsw** (**extsw.**)	rA,rS
NAND	**nand** (**nand.**)	rA,rS,rB
NOR	**nor** (**nor.**)	rA,rS,rB
OR	**or** (**or.**)	rA,rS,rB
OR Immediate	**ori**	rA,rS,UIMM
OR Immediate Shifted	**oris**	rA,rS,UIMM
OR with Complement	**orc** (**orc.**)	rA,rS,rB
Parity Doubleword <64>	**prtyd**	rA,rS
Parity Word	**prtyw**	rA,rS
Population Count Byte	**popcntb**	rA,rS
Population Count Doubleword <64>	**popcntd**	rA,rS
Population Count Word	**popcntw**	rA,rS
XOR	**xor** (**xor.**)	rA,rS,rB
XOR Immediate	**xori**	rA,rS,UIMM
XOR Immediate Shifted	**xoris**	rA,rS,UIMM

The **ori r0,r0,0** instruction is the preferred form for a no-op.

The **or r**_x_**,r**_x_**,r**_x_ instruction is used by the architecture to provide hints to the implementation. See page 5-217.

4.6.1.1.4 Integer Rotate and Shift Instructions

Rotation operations are performed on data from a GPR, and the result, or a portion of the result, is returned to a GPR. Integer rotate instructions, summarized in the following table, rotate the contents of a register. The result is either inserted into the target register under control of a mask (if a mask bit is set the associated bit of the rotated data is placed into the target register, and if the mask bit is cleared the associated bit in the target register is unchanged) or ANDed with a mask before being placed into the target register. Section C.3, “Rotate and Shift Simplified Mnemonics,” lists simplified mnemonics that allow simpler coding of often used functions such as clearing the left- or right-most bits of a register, left or right justifying an arbitrary field, and simple rotates and shifts.

Table 4-9. Integer Rotate Instructions

Name	Mnemonic	Syntax
Rotate Left Doubleword then Clear Left <64>	**rldcl** (**rldcl.**)	**r**A,**r**S,**r**B,MB
Rotate Left Doubleword then Clear Right <64>	**rldcr** (**rldcr.**)	**r**A,**r**S,**r**B,ME
Rotate Left Doubleword Immediate then Clear <64>	**rldic** (**rldic.**)	**r**A,**r**S,SH,MB
Rotate Left Doubleword Immediate then Clear Left <64>	**rldicl** (**rldicl.**)	**r**A,**r**S,SH,MB
Rotate Left Doubleword Immediate then Clear Right <64>	**rldicr** (**rldicr.**)	**r**A,**r**S,SH,ME
Rotate Left Doubleword Immediate then Mask Insert <64>	**rldimi**(**rldimi.**)	**r**A,**r**S,SH,MB
Rotate Left Word then AND with Mask	**rlwnm** (**rlwnm.**)	**r**A,**r**S,**r**B,MB,ME
Rotate Left Word Immediate then Mask Insert	**rlwimi** (**rlwimi.**)	**r**A,**r**S,SH,MB,ME
Rotate Left Word Immediate then AND with Mask	**rlwinm** (**rlwinm.**)	**r**A,**r**S,SH,MB,ME

The integer shift instructions (Table 4-10) perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Simplified mnemonics (shown in Section C.3, “Rotate and Shift Simplified Mnemonics”) are provided to simplify coding of such shifts.

Multiple-precision shifts can be programmed as shown in Section D.2, “Multiple-Precision Shifts.” The integer shift instructions are summarized in this table.

Table 4-10. Integer Shift Instructions

Name	Mnemonic	Syntax
Shift Left Doubleword <64>	**sld** (**sld.**)	**r**A,**r**S,**r**B
Shift Left Word	**slw** (**slw.**)	**r**A,**r**S,**r**B
Shift Right Doubleword <64>	**srd** (**srd.**)	**r**A,**r**S,**r**B
Shift Right Word	**srw** (**srw.**)	**r**A,**r**S,**r**B
Shift Right Algebraic Doubleword Immediate <64>	**sradi** (**sradi.**)	**r**A,**r**S,SH
Shift Right Algebraic Word Immediate	**srawi** (**srawi.**)	**r**A,**r**S,SH
Shift Right Algebraic Doubleword <64>	**srad**(**srad.**)	**r**A,**r**S,**r**B
Shift Right Algebraic Word	**sraw** (**sraw.**)	**r**A,**r**S,**r**B

4.6.1.2 Load and Store Instructions

Load and store instructions are issued and translated in program order; however, the accesses can occur out of order. See Section 6.4.8.1, “Memory Access Ordering.” The following load and store instructions are defined for user-level:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte-reverse instructions
- Integer load and store multiple instructions
- Floating-point load and store instructions <FP>
- Load and reserve and store conditional instructions
- Decorated storage load and store instructions <DS>
- SPE load and store instructions. See the SPE PEM. <SP>
- AltiVec load and store instructions. See the AltiVec PEM. <V>

4.6.1.2.1 Load and Store Address Generation

Load and store operations generate EAs using register indirect with immediate index mode, register indirect with index mode, or register indirect mode. Only the low order 32 bits generated EA are used in 32-bit mode. The generate EA is then translated to a real address and the real address is used to perform the memory access. The different EA calculation modes are described as follows:

- Register indirect with immediate index addressing for loads and stores. Instructions using this addressing mode contain a signed 16-bit immediate index (d operand), which is sign extended and added to the contents of a general-purpose register specified in the instruction (**r**A operand), to generate the EA. If **r0** is specified, a value of zero is added to the immediate index (d operand) in place of the contents of **r0**. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0). The following figure shows how an EA is generated using this mode.

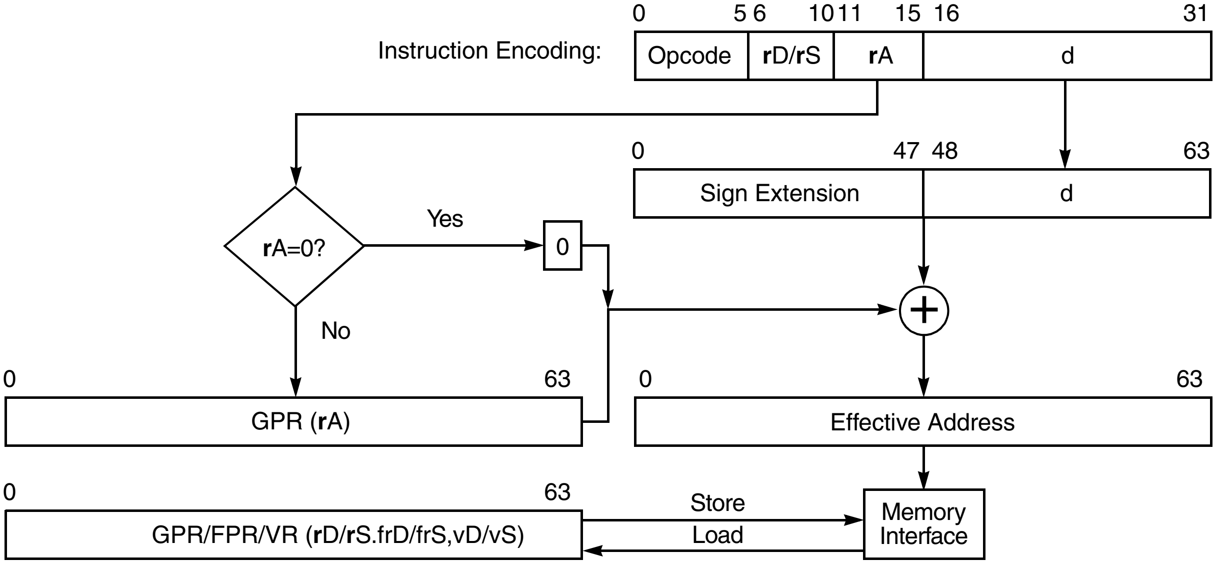


Figure 4-1. Register Indirect with Immediate Index Addressing for Loads/Stores

- Register indirect with index addressing for loads and stores. Instructions using this mode cause the contents of two GPRs (specified as operands **r**A and **r**B) to be added in the EA generation. A zero in place of the **r**A operand causes a zero to be added to the GPR contents specified in operand **r**B. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0). The following figure shows how an EA is generated using this mode.

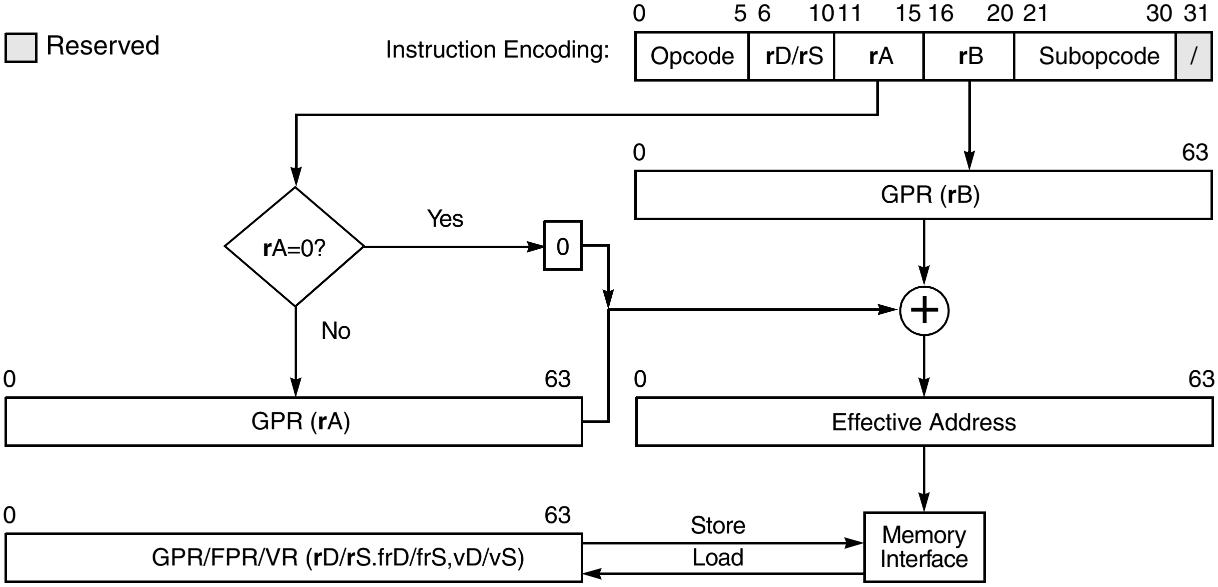


Figure 4-2. Register Indirect with Index Addressing for Loads/Stores

- Register indirect addressing for loads and stores. Decorated storage instructions using this addressing mode use the contents of the GPR specified by the **r**B operand as the EA. The rA operand contains a decoration value to be used by the targeted device. The following figure shows how an EA is generated using this mode.

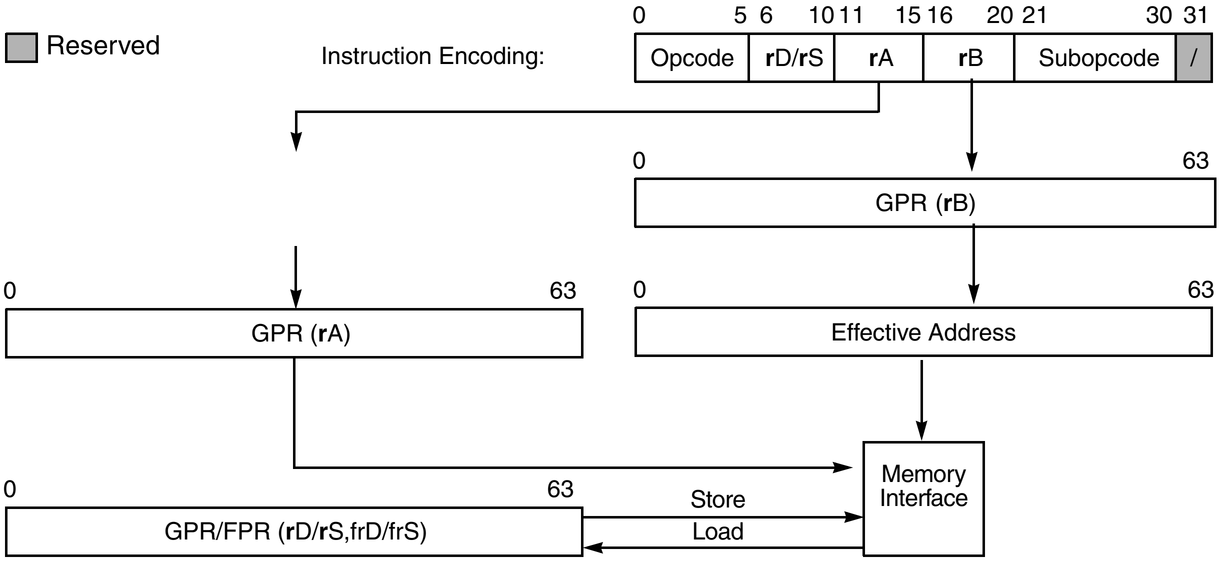


Figure 4-3. Register Indirect Addressing for Integer Loads/Stores

For information about calculating EAs, see Section 4.5.3.3, “Effective Address Calculation.”

NOTE

In some implementations, operations that are not naturally aligned can suffer performance degradation. See Section 7.8.6, “Alignment Interrupt,” for additional information about load and store address alignment interrupts.

4.6.1.2.2 Update Forms of Load and Store Instructions

Many load and store instructions have an update form, in which **r**A is updated with the generated EA. For these forms, the EA is placed into **r**A and the memory element (byte, half word, or word) addressed by the EA is loaded into **r**D (load instructions) or stored from **r**S (store instructions).

For load instructions using the update form if the register specified by **r**A = 0 or **r**A = **r**D, the form is invalid and the result is boundedly undefined.

For store instructions using the update form if the register specified by **r**A = 0, the form is invalid and the result is boundedly undefined.

Not all variations of load and store instructions contain an update form.

4.6.1.2.3 Integer Load Instructions

For integer load instructions, the byte, half-word, word, or double-word addressed by EA is loaded into the register specified by **r**D. If the load instruction is an update form, **r**A is updated with the EA.

This table summarizes the integer load instructions.

Table 4-11. Integer Load Instructions

Name	Mnemonic	Syntax
Load Byte and Zero	**lbz**	**r**D,**d**(**r**A)
Load Byte and Zero Indexed	**lbzx**	**r**D,**r**A,**r**B
Load Byte and Zero with Update	**lbzu**	**r**D,**d**(**r**A)
Load Byte and Zero with Update Indexed	**lbzux**	**r**D,**r**A,**r**B
Load Doubleword **<64>**	**ld**	**r**D,**d**(**r**A)
Load Doubleword Indexed **<64>**	**ldx**	**r**D,**r**A,**r**B
Load Doubleword with Update **<64>**	**ldu**	**r**D,**d**(**r**A)
Load Doubleword with Update Indexed **<64>**	**ldux**	**r**D,**r**A,**r**B
Load Half Word and Zero	**lhz**	**r**D,**d**(**r**A)
Load Half Word and Zero Indexed	**lhzx**	**r**D,**r**A,**r**B
Load Half Word and Zero with Update	**lhzu**	**r**D,**d**(**r**A)
Load Half Word and Zero with Update Indexed	**lhzux**	**r**D,**r**A,**r**B
Load Half Word Algebraic	**lha**	**r**D,**d**(**r**A)
Load Half Word Algebraic Indexed	**lhax**	**r**D,**r**A,**r**B
Load Half Word Algebraic with Update	**lhau**	**r**D,**d**(**r**A)
Load Half Word Algebraic with Update Indexed	**lhaux**	**r**D,**r**A,**r**B

Table 4-11. Integer Load Instructions (continued)

Name	Mnemonic	Syntax
Load Word and Zero	**lwz**	**r**D,**d(r**A)
Load Word and Zero Indexed	**lwzx**	**r**D,**r**A,**r**B
Load Word and Zero with Update	**lwzu**	**r**D,**d(r**A)
Load Word and Zero with Update Indexed	**lwzux**	**r**D,**r**A,**r**B
Load Word Algebraic <64>	**lwa**	**r**D,**d(r**A)
Load Word Algebraic Indexed <64>	**lwax**	**r**D,**r**A,**r**B
Load Word Algebraic with Update Indexed <64>	**lwaux**	**r**D,**r**A,**r**B

NOTE: Software Considerations

Some implementations may execute the load algebraic (**lha**, **lhax**, **lhau**, **lhaux**, **lwa** <64>, **lwax** <64>, **lwaux** <64>) instructions with greater latency than other types of load instructions.

4.6.1.2.4 Integer Store Instructions

For integer store instructions, the **r**S contents are stored into the byte, half word, word or double-word in memory addressed by the EA. If the store instruction is an update form, **r**A is updated with the EA.

This table summarizes integer store instructions.

Table 4-12. Integer Store Instructions

Name	Mnemonic	Syntax
Store Byte	**stb**	**r**S,**d(r**A)
Store Byte Indexed	**stbx**	**r**S,**r**A,**r**B
Store Byte with Update	**stbu**	**r**S,**d(r**A)
Store Byte with Update Indexed	**stbux**	**r**S,**r**A,**r**B
Store Doubleword <64>	**std**	**r**S,**d(r**A)
Store Doubleword Indexed <64>	**stdx**	**r**S,**r**A,**r**B
Store Doubleword with Update <64>	**stdu**	**r**S,**d(r**A)
Store Doubleword with Update Indexed <64>	**stdux**	**r**S,**r**A,**r**B
Store Half Word	**sth**	**r**S,**d(r**A)
Store Half Word Indexed	**sthx**	**r**S,**r**A,**r**B
Store Half Word with Update	**sthu**	**r**S,**d(r**A)
Store Half Word with Update Indexed	**sthux**	**r**S,**r**A,**r**B
Store Word	**stw**	**r**S,**d(r**A)
Store Word Indexed	**stwx**	**r**S,**r**A,**r**B

Table 4-12. Integer Store Instructions (continued)

Name	Mnemonic	Syntax
Store Word with Update	**stwu**	**r**S,**d(r**A**)**
Store Word with Update Indexed	**stwux**	**r**S,**r**A,**r**B

4.6.1.2.5 Integer Load and Store with Byte-Reverse Instructions

The following table describes integer load and store with byte-reverse instructions. For integer load with byte-reverse instructions, the byte, half-word, word, or double-word addressed by EA is byte-reversed and loaded into the register specified by **r**D. For integer store with byte-reverse instructions, the **r**S contents are byte-reversed then stored into the byte, half word, word or double-word in memory addressed by the EA.

These instructions were defined in part to support the original PowerPC definition of little-endian byte ordering. Note that the EIS supports true little endian on a per-page basis. For more information, see Section 4.5.3.4, "Byte Ordering."

Table 4-13. Integer Load and Store with Byte-Reverse Instructions

Name	Mnemonic	Syntax
Load Doubleword Byte-Reverse Indexed <64>	**ldbrx**	**r**D,**r**A,**r**B
Load Half Word Byte-Reverse Indexed	**lhbrx**	**r**D,**r**A,**r**B
Load Word Byte-Reverse Indexed	**lwbrx**	**r**D,**r**A,**r**B
Store Doubleword Byte-Reverse Indexed <64>	**stdbrx**	**r**D,**r**A,**r**B
Store Half Word Byte-Reverse Indexed	**sthbrx**	**r**S,**r**A,**r**B
Store Word Byte-Reverse Indexed	**stwbrx**	**r**S,**r**A,**r**B

4.6.1.2.6 Integer Load and Store Multiple Instructions

The load/store multiple instructions, listed in Table 4-14, move blocks of data to and from the GPRs. If their operands require memory accesses crossing a page boundary, these instructions may require a data storage interrupt or a data TLB error interrupt to translate the second page. Also, if one of these instructions is interrupted, it will be restarted, requiring multiple memory accesses.

The architecture defines the **lmw** instruction with **r**A in the range of registers to be loaded as an invalid form.

Load and store multiple accesses must be word aligned; otherwise, they cause an alignment exception.

Table 4-14. Integer Load and Store Multiple Instructions

Name	Mnemonic	Syntax
Load Multiple Word	**lmw**	**r**D,**d(r**A**)**
Store Multiple Word	**stmw**	**r**S,**d(r**A**)**

4.6.1.2.7 Floating-Point Load and Store Instructions <FP>

There are two forms of floating-point load and store instructions—single-precision (word) and double-precision (double-word) operand formats. FPRs are 64-bits and operate on 64-bit floating point values. When a single precision (word) value is loaded into a FPR, the single-precision value is converted to a double-precision (64-bit) format. Conversely, when a single precision (word) value is stored from an FPR to memory, the double-precision value is converted to a single-precision (32-bit) format then stored to memory.

For floating-point load single instructions, the word addressed by EA is interpreted as a floating-point single precision operand and is converted to floating-point double format then loaded into the register specified by **fr**D. The conversion process is as follows, where $WORD_{0:31}$ is the floating-point single-precision operand accessed from memory:

```
Normalized Operand
 if $WORD_{1:8} > 0$ and $WORD_{1:8} < 255$ then
    $FPR(frD)_{0:1} \leftarrow WORD_{0:1}$

    $FPR(frD)_{2} \leftarrow \neg WORD_{1}$
    $FPR(frD)_{3} \leftarrow \neg WORD_{1}$
    $FPR(frD)_{4} \leftarrow \neg WORD_{1}$
    $FPR(frD)_{5:63} \leftarrow WORD_{2:31} \| {}^{29}0$

Denormalized Operand
if $WORD_{1:8} = 0$ and $WORD_{9:31} \neq 0$ then
    sign $\leftarrow WORD_{0}$
    exp $\leftarrow$ -126
    $frac_{0:52} \leftarrow 0b0 \| WORD_{9:31} \| {}^{29}0$
    normalize the operand
        do while $frac_{0} = 0$
            frac $\leftarrow frac_{1:52} \| 0b0$
            exp $\leftarrow$ exp - 1
    $FPR(frD)_{0} \leftarrow$ sign
    $FPR(frD)_{1:11} \leftarrow$ exp + 1023
    $FPR(frD)_{12:63} \leftarrow frac_{1:52}$

Zero/Infinity/NaN
if $WORD_{1:8} = 255$ or $WORD_{1:31} = 0$ then
    $FPR(frD)_{0:1} \leftarrow WORD_{0:1}$
    $FPR(frD)_{2} \leftarrow WORD_{1}$
    $FPR(frD)_{3} \leftarrow WORD_{1}$
    $FPR(frD)_{4} \leftarrow WORD_{1}$
    $FPR(frD)_{5:63} \leftarrow WORD_{2:31} \| {}^{29}0$
```

For floating-point store single instructions (except **stfiwx**), the **fr**S contents are converted to a single precision format then stored into the word in memory addressed by EA. The conversion steps are as follows, where $WORD_{0:31}$ is the word in memory written to:

```
No Denormalization Required (includes Zero / Infinity / NaN)
if $FPR(FRS)_{1:11} > 896$ or $FPR(FRS)_{1:63} = 0$ then
    $WORD_{0:1} \leftarrow FPR(FRS)_{0:1}$
    $WORD_{2:31} \leftarrow FPR(FRS)_{5:34}$

Denormalization Required
if $874 \leq FRS_{1:11} \leq 896$ then
    sign $\leftarrow FPR(FRS)_{0}$
    exp $\leftarrow FPR(FRS)_{1:11}$ - 1023
    frac $\leftarrow 0b1 \| FPR(FRS)_{12:63}$
    denormalize operand
        do while exp < –126
```

```
            frac ← 0b0 || frac$_{0:62}$
            exp ← exp + 1
    WORD$_0$ ← sign
    WORD$_{1:8}$ ← 0x00
    WORD$_{9:31}$ ← frac$_{1:23}$
else WORD ← undefined
```

When a floating-point store single operation occurs, the value in the FPR is converted to single precision and may require denormalization. If the value in the FPR is smaller in precision than is representable in single-precision format a signed zero value is stored. If the value in the FPR is larger in magnitude than is representable, the value stored is computed as follows (assuming that FPR is a double-precision representation of the value in **fr**S):

```
WORD ← FPR$_{0:1}$ || FPR$_{5:34}$
```

In this case, the value stored in memory is not numerically equal to the value in the FPR that was stored.

The **stfiwx** instruction stores the low-order 32 bits of frS without conversion into the word in memory addressed by EA.

NOTE: Software Considerations

The **stfiwx** instruction is useful only for storing the contents of the FPSCR after it has been moved to an FPR.

For floating-point load double instructions, the double-word addressed by EA is loaded into the register specified by **fr**D. For floating-point store double instructions, the **fr**S contents are stored into the double-word in memory addressed by EA. No conversion is required when loading or storing floating-point doubles as the data is copied directly to or from the FPR.

If the load or store instruction is an update form, **r**A is updated with the EA.

Attempted execution of a floating-point load or store instruction causes a floating-point unavailable interrupt if MSR[FP] = 0.

This table summarizes the floating-point load and store instructions.

Table 4-15. Floating-Point Load and Store Instructions

Name	Mnemonic	Syntax
Load Floating-Point Single	**lfs**	**fr**D,**d(r**A**)**
Load Floating-Point Single Indexed	**lfsx**	**fr**D,**r**A,**r**B
Load Floating-Point Single with Update	**lfsu**	**fr**D,**d(r**A**)**
Load Floating-Point Single with Update Indexed	**lfsux**	**fr**D,**r**A,**r**B
Load Floating-Point Double	**lfd**	**fr**D,**d(r**A**)**
Load Floating-Point Double Indexed	**lfdx**	**fr**D,**r**A,**r**B
Load Floating-Point Double with Update	**lfdu**	**fr**D,**d(r**A**)**
Load Floating-Point Double with Update Indexed	**lfdux**	**fr**D,**r**A,**r**B
Store Floating-Point Single	**stfs**	**fr**S,**d(r**A**)**
Store Floating-Point Single Indexed	**stfsx**	**fr**S,**r**A,**r**B

Table 4-15. Floating-Point Load and Store Instructions (continued)

Name	Mnemonic	Syntax
Store Floating-Point Single with Update	**stfsu**	**fr**S,**d(**rA**)**
Store Floating-Point Single with Update Indexed	**stfsux**	**fr**S,**r**A,**r**B
Store Floating-Point Double	**stfd**	**fr**S,**d(**rA**)**
Store Floating-Point Double Indexed	**stfdx**	**fr**S,**r**A,**r**B
Store Floating-Point Double with Update	**stfdu**	**fr**S,**d(**rA**)**
Store Floating-Point Double with Update Indexed	**stfdux**	**fr**S,**r**A,**r**B
Store Floating-Point as Integer Word Indexed	**stfiwx**	**fr**S,**r**A,**r**B

NOTE: Software Considerations

Because no conversion occurs on loading and storing floating-point double precision memory locations, load floating double and store floating double can be used to copy memory contents efficiently using 64-bit accesses on a 32-bit implementation.

4.6.1.2.8 Load and Reserve and Store Conditional Instructions

Load and reserve and store conditional instructions are used to perform atomic operations on memory, particularly in a multiprocessor shared memory environment. Load and reserve instructions load data into a GPR and establish a reservation held by the processor. Store conditional instructions store to memory only if a previous reservation exists and is still valid. CR0 is updated based on whether the store was performed or not.

Reservations are based on real addresses, and can only be established by load and reserve instructions. Reservations are lost (become invalid) if another load and reservation is performed by the same processor to a different address (establishes a new reservation), or another processor or device modifies (perform a store) the address previously established by the load and reserve instruction. Reservations can be lost for other reasons as well.

Load and reserve and store conditional instructions can be used to emulate semaphore operations such as test and set, compare and swap, atomic update, exchange memory, and so on, along with simple lock semantics. For a more in depth discussion see Section 6.4.8, “Shared Memory.”

Although reservations are based on the coherency granule, load and reserve and store conditional sizes and addresses must match for a store conditional to be guaranteed to succeed. Mixing sizes (for example, **lwarx** with **stdcx.** <64>) or mixing addresses within an implementation’s coherency granule (for example, **ldarx** <64> to 0x100000 and **stwcx.** to 0x100024 on a processor with a coherency granule of x040), may succeed on one implementation but fail on another.

Execution of a load and reserve or store conditional instruction to a misaligned address causes an alignment interrupt.

The memory attributes allowed for load and reserve and store conditional instructions are implementation dependent. Execution of a load and reserve or store conditional instruction to an address that is not

Memory Coherence Required, or is Write Through Required or Caching Inhibited, may cause a data storage interrupt. See the core reference manual.

This table lists the load and reserve and store conditional instructions.

Table 4-16. Load and Reserve and Store Conditional Instructions

Name	Mnemonic	Syntax	Notes
Load Byte and Reserve Indexed <ER>	**lbarx**	**r**D,**r**A,**r**B	Should be paired with a later **stbcx.** to the same real address.
Load Doubleword and Reserve Indexed <64>	**ldarx**	**r**D,**r**A,**r**B	Should be paired with a later **stdcx.** to the same real address.
Load Halfword and Reserve Indexed <ER>	**lharx**	**r**D,**r**A,**r**B	Should be paired with a later **sthcx.** to the same real address.
Load Word and Reserve Indexed	**lwarx**	**r**D,**r**A,**r**B	Should be paired with a later **stwcx.** to the same real address.
Store Byte Conditional Indexed <ER>	**stbcx.**	**r**S,**r**A,**r**B	Should be paired with a previous **lbarx** to the same real address.
Store Doubleword Conditional Indexed <64>	**stdcx.**	**r**S,**r**A,**r**B	Should be paired with a previous **ldarx** to the same real address.
Store Halfword Conditional Indexed <ER>	**sthcx.**	**r**S,**r**A,**r**B	Should be paired with a previous **lharx** to the same real address.
Store Word Conditional Indexed	**stwcx.**	**r**S,**r**A,**r**B	Should be paired with a previous **lwarx** to the same real address.

4.6.1.2.9 Decorated Load and Store Instructions <DS>

Decorated load and store instructions provide load and store operations to memory addresses (integrated device-specific) that have additional semantics available other than the customary load (read) and store (write). A “decoration” is additional semantic information applied to the decorated storage operation. The contents of **r**A specify the decoration. The contents of **r**B specify the address. The decoration is delivered with the address and data size to the device associated with the memory address. For decorated store instructions the contents of **r**S is also sent. The device performs device-specific semantics using the decoration, address, size, and store data (for decorated store instructions) and returns load data that is placed in **r**D (for decorated load instructions). The number of bits of decoration that are delivered to the target address is implementation specific.

Decorated storage also provides a “notify” instruction, **dsn**. A notify is a type of access that is neither load or store because it does not provide a data value (store) and does not receive a data value (load). A notify sends the decoration with the address to the device associated with the memory address. The device uses the decoration to determine the semantic operation.

Decorated load and store operations behave the same as loads and stores with respect to all other aspects of load and store operations provided by the processor for translation, access control, debug events, storage attributes, alignment, and memory access ordering. A decorated notify operation is treated as a store with respect to these same attributes.

Additionally, some implementations require write permissions for decorated load instructions performed to memory that is Caching Inhibited.

Decorated storage operations to memory that are Caching Inhibited are treated as Guarded regardless of the setting of the Guarded memory attribute.

Decorated storage operations to memory, other than **dsn**, that are not Caching Inhibited are treated the same as the corresponding non-decorated storage operations, except that an implementation-specific performance hint may be provided by the decoration. A **dsn** to memory that is not Caching Inhibited should be treated as a no-op.

A decorated load or store to a device that does not support decorations is boundedly undefined. A non decorated load or store to a device that requires decorations is boundedly undefined.

Decorated load and store instructions allow efficient, device-specific operations targeted by storage address, such as packet-counting statistics. The target device defines specific semantics understood by a customized resource that requires them. As such, EIS only defines the instruction semantics as defined by the processor. To determine the full semantic of a decorated storage operation, consult the reference manual for the integrated device.

NOTE: Software Considerations

- Software must be acutely aware of the effects of operations defined by the decorations supplied and must ensure that the processor only issues operations appropriately. This will generally require the addresses to be marked as Guarded and Caching Inhibited and appropriate memory barriers are used to ensure order.
- Software should ensure that accesses are aligned, otherwise atomic accesses are not guaranteed and may require multiple accesses to perform the operation which is likely not to produce the intended result.

This table lists the decorated storage load and store instructions.

Table 4-17. Decorated Load and Store Instructions

Instruction	Mnemonic	Syntax
Load Byte with Decoration Indexed	**lbdx**	**r**D,**r**A,**r**B
Load Half Word with Decoration Indexed	**lhdx**	**r**D,**r**A,**r**B
Load Word with Decoration Indexed	**lwdx**	**r**D,**r**A,**r**B
Load Doubleword with Decoration Indexed <64>	**lddx**	**r**D,**r**A,**r**B
Load Floating-Point Doubleword with Decoration Indexed <FP>	**lfddx**	**fr**D,**r**A,**r**B
Store Byte with Decoration Indexed	**stbdx**	**r**S,**r**A,**r**B
Store Half Word with Decoration Indexed	**sthdx**	**r**S,**r**A,**r**B
Store Word with Decoration Indexed	**stwdx**	**r**S,**r**A,**r**B
Store Doubleword with Decoration Indexed <64>	**stddx**	**r**S,**r**A,**r**B
Store Floating-Point Doubleword with Decoration Indexed <FP>	**stfddx**	**fr**S,**r**A,**r**B
Decorated Storage Notify	**dsn**	**r**A,**r**B

4.6.1.3 Floating-Point Instructions <FP>

This section describes the floating-point instructions other than the floating-point load and store instructions, which are defined in Section 4.6.1.2.7, “Floating-Point Load and Store Instructions <FP>.”

The EIS provides for hardware implementation of a floating-point system as defined in ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating Point Arithmetic*. For detailed information about the floating-point data formats, see Section 3.4.3, “Floating-Point Data.”

The IEEE Std. 754 includes 64- and 32-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands.

The EIS follows these guidelines:

- Double-precision arithmetic instructions may have single-precision operands but always produce double-precision results.
- Single-precision arithmetic instructions require all operands to be single-precision and always produce single-precision results.

For arithmetic instructions, conversions from double- to single-precision must be done explicitly by software, while conversions from single- to double-precision are done implicitly.

To ensure that identical results are obtained, all PowerISA implementations provide the equivalent of the execution models described in this chapter. The definition of the arithmetic instructions for infinities, denormalized numbers, and NaNs follow conventions described in Section 3.4.3, “Floating-Point Data” and the following sections.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic uses two additional bit positions to avoid potential transient overflow conditions. An extra bit is required when denormalized double-precision numbers are prenormalized. A second bit is required to permit computation of the adjusted exponent value in the following examples when the corresponding exception enable bit is one:

- Underflow during multiplication using a denormalized factor
- Overflow during division using a denormalized divisor

4.6.1.3.1 Floating-Point Move Instructions

Described in the following table, floating-point move instructions copy data from one FPR to another, altering the sign bit (bit 0) as described below for **fneg**, **fabs**, and **fnabs**. These instructions treat NaNs just like any other kind of value (for example, the sign bit of a NaN may be altered by **fneg**, **fabs**, fcpsign, and **fnabs**). These instructions do not alter the FPSCR.

Table 4-18. Floating-Point Move Instructions

Instruction	Mnemonic	Syntax
Floating Absolute Value	**fabs**[.]	**fr**D,**fr**B
Floating Absolute Value	**fcpsign**[.]	**fr**D,**fr**B

Table 4-18. Floating-Point Move Instructions (continued)

Instruction	Mnemonic	Syntax
Floating Move Register	**fmr**[.]	**fr**D,**fr**B
Floating Negative Absolute Value	**fnabs**[.]	**fr**D,**fr**B
Floating Negate	**fneg**[.]	**fr**D,**fr**B

NOTE: Software Considerations

Many processors that implement category Floating-point, do not implement the **fcpsign** instruction. This is because this instruction was not implemented in early PowerPC processors. Software should consult their core reference manual and be wary that this instruction may not be implemented on other cores.

4.6.1.3.2 Floating-Point Arithmetic Instructions

This table lists mnemonics and syntax of floating-point arithmetic instructions.

Table 4-19. Floating-Point Arithmetic Instructions

Instruction	Mnemonic	Syntax
Floating Add	**fadd**[.]	**fr**D,**fr**A,**fr**B
Floating Add Single	**fadds**[.]	**fr**D,**fr**A,**fr**B
Floating Divide	**fdiv**[.]	**fr**D,**fr**A,**fr**B
Floating Divide Single	**fdivs**[.]	**fr**D,**fr**A,**fr**B
Floating Multiply	**fmul**[.]	**fr**D,**fr**A,**fr**C
Floating Multiply Single	**fmuls**[.]	**fr**D,**fr**A,**fr**C
Floating Reciprocal Estimate	**fre**[.]	**fr**D,**fr**B
Floating Reciprocal Estimate Single	**fres**[.]	**fr**D,**fr**B
Floating Reciprocal Square Root Estimate	**frsqrte**[.]	**fr**D,**fr**B
Floating Reciprocal Square Root Estimate Single	**frsqrtes**[.]	**fr**D,**fr**B
Floating Square Root	**fsqrt**[.]	**fr**D,**fr**B
Floating Square Root Single	**fsqrts**[.]	**fr**D,**fr**B
Floating Test for software Divide	**ftdiv**[.]	**cr**D,**fr**A,**fr**B
Floating Test for software Square Root	**ftdiv**[.]	**cr**D,**fr**B
Floating Subtract	**fsub**[.]	**fr**D,**fr**A,**fr**B
Floating Subtract Single	**fsubs**[.]	**fr**D,**fr**A,**fr**B

NOTE: Software Considerations

Many processors that implement category Floating-point, do not implement the **fre**, **fsqrt**, **fsqrts**, **ftdiv**, **ftsqrt**, or **frsqrtes** instructions. This is because such instructions were not implemented in early PowerPC processors. Software should consult their core reference manual and be wary that all or some of these may not be implemented on other cores.

4.6.1.3.3 Floating-Point Multiply-Add Instructions

These instructions combine a multiply and an add operation without an intermediate rounding operation. FPSCR status bits, described in Table 4-20 are set as follows:

- Overflow, underflow, and inexact exception bits, the FR, FI, and FPRF fields are set based on the final result of the operation, not on the result of the multiplication.
- Invalid operation exception bits are set as if the multiplication and the addition were performed using two separate instructions (**fmul**[**s**], followed by **fadd**[**s**] or **fsub**[**s**]). That is, any of the following actions will cause appropriate exception bits to be set:
  - — Multiplication of infinity by 0
  - — Multiplication of anything by an SNaN
  - — Addition of anything with an SNaN

Table 4-20. Floating-Point Multiply-Add Instructions

Instruction	Mnemonic	Instruction
Floating Multiply-Add	**fmadd**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Multiply-Add Single	**fmadds**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Multiply-Subtract	**fmsub**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Multiply-Subtract Single	**fmsubs**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Negative Multiply-Add	**fnmadd**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Negative Multiply-Add Single	**fnmadds**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Negative Multiply-Subtract	**fnmsub**[.]	**fr**D,**fr**A,**fr**B,**fr**C
Floating Negative Multiply-Subtract Single	**fnmsubs**[.]	**fr**D,**fr**A,**fr**B,**fr**C

4.6.1.4 Floating-Point Rounding and Conversion Instructions

Examples of uses of these instructions to perform various conversions can be found in Section D.3, “Floating-Point Conversions <FP>.”

This table lists floating-point rounding and conversion instructions.

Table 4-21. Floating-Point Rounding and Conversion Instructions

Instruction	Mnemonic	Syntax
Floating Convert from Integer Double Word	**fcfid**[.]	**fr**D,**fr**B
Floating Convert from Integer Double Word Single	**fcfids**[.]	**fr**D,**fr**B
Floating Convert from Integer Double Word Unsigned	**fcfidu**[.]	**fr**D,**fr**B
Floating Convert from Integer Double Word Unsigned Single	**fcfidus**[.]	**fr**D,**fr**B
Floating Convert to Integer Double Word	**fctid**[.]	**fr**D,**fr**B
Floating Convert to Integer Double Word Unsigned	**fctidu**[.]	**fr**D,**fr**B
Floating Convert to Integer Double Word Unsigned and round toward Zero	**fctiduz**[.]	**fr**D,**fr**B

Table 4-21. Floating-Point Rounding and Conversion Instructions (continued)

Instruction	Mnemonic	Syntax
Floating Convert to Integer Double word and round toward Zero	**fctidz**[.]	**fr**D,**fr**B
Floating Convert to Integer Word	**fctiw**[.]	**fr**D,**fr**B
Floating Convert to Integer Word Unsigned	**fctiwu**[.]	**fr**D,**fr**B
Floating Convert to Integer Word Unsigned and Round to Zero	**fctiwuz**[.]	**fr**D,**fr**B
Floating Convert to Integer Word and Round to Zero	**fctiwz**[.]	**fr**D,**fr**B
Floating Round to Integer Minus	**frim**[.]	**fr**D,**fr**B
Floating Round to Integer Nearest	**frin**[.]	**fr**D,**fr**B
Floating Round to Integer Plus	**frip**[.]	**fr**D,**fr**B
Floating Round to Integer Toward Zero	**friz**[.]	**fr**D,**fr**B
Floating Round to Single-Precision	**frsp**[.]	**fr**D,**fr**B

NOTE: Software Considerations

Many processors that implement category Floating-point, do not implement the **fcfid**, **fcfids**, **fcfidu**, **fcfidus**, **fctid**, **fctidu**, **fctiduz**, **fctidz**, **fctiwu**, **fctiwuz**, **frim**, **frin**, **frip**, or **friz** instructions. This is because such instructions were not implemented in early PowerPC processors. Software should consult their core reference manual and be wary that all or some of these may not be implemented on other cores.

4.6.1.5 Floating-Point Compare Instructions

The floating-point compare instructions compare the contents of two FPRs. Comparison ignores the sign of zero (that is, regards +0 as equal to –0). The comparison result can be ordered or unordered. The comparison sets one bit in the designated CR field and clears the other three. The floating-point condition code, FPSCR[FPCC], is set in the same way.

The CR field and the FPCC are set as described in this table.

Table 4-22. CR Field Settings

Bit	Name	Description
0	FL	(**fr**A) < (**fr**B)
1	FG	(**fr**A) > (**fr**B)
2	FE	(**fr**A) = (**fr**B)
3	FU	(**fr**A) ? (**fr**B) (unordered)

The floating-point compare and select instruction set is shown in this table.

Table 4-23. Floating-Point Compare and Select Instructions

Instruction	Mnemonic	Syntax
Floating Compare Ordered	**fcmpo**	**cr**D,**fr**A,**fr**B
Floating Compare Unordered	**fcmpu**	**cr**D,**fr**A,**fr**B
Floating Select	**fsel**[.]	**fr**D,**fr**A,**fr**B,**fr**C

4.6.1.6 Floating-Point Status and Control Register Instructions

Every FPSCR instruction synchronizes the effects of all floating-point instructions executed by a given processor. Executing an FPSCR instruction ensures that all floating-point instructions previously initiated by the processor complete before the FPSCR instruction is initiated and that no subsequent floating-point instructions are initiated by the processor until the FPSCR instruction completes.

- All exceptions caused by the previously initiated instructions are recorded in the FPSCR before the FPSCR instruction is initiated.
- All invocations of floating-point enabled exception-type program interrupt that is caused by the previously initiated instructions have occurred before the FPSCR instruction is initiated.
- No subsequent floating-point instruction that depends on or alters FPSCR bits begins until the FPSCR instruction completes.

Floating-point load and floating-point store instructions are not affected.

Table 4-24. Floating-Point Status and Control Register Instructions

Instruction	Mnemonic	Syntax
Move from FPSCR	**mffs**, **mffs.**	**fr**D
Move to FPSCR Bit 0	**mtfsb0**, **mtfsb0.**	**crb**D
Move to FPSCR Bit 1	**mtfsb1**, **mtfsb1.**	**crb**D
Move to FPSCR Fields	**mtfsf**, **mtfsf.**	FM,**fr**B
Move to FPSCR Field Immediate	**mtfsfi**, **mtfsfi.**	**cr**D,IMM

4.6.1.7 Floating-Point Exceptions

This architecture defines the following floating-point exceptions:

- Invalid operation exception
  - SNaN
  - Infinity - infinity
  - Infinity÷Infinity
  - Zero÷Zero
  - Infinity×Zero
  - Invalid compare

  - — Software-defined condition
  - — Invalid square root
  - — Invalid integer convert
- Zero-divide exception
- Overflow exception
- Underflow exception
- Inexact exception

These exceptions, other than invalid operation exceptions due to a software-defined condition, may occur during execution of computational instructions. An invalid operation exception due to a software-defined condition occurs when a Move to FPSCR instruction sets FPSCR[VXSOFT].

Each floating-point exception, and each category of invalid operation exception, has an exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with the FE0 and FE1 bits (see Table 4-25), whether and how the system floating-point enabled exception error handler is invoked. (In general, the enabling specified by the enable bit is of invoking the system error handler, not of permitting the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any control bits. The only deviation from this general rule is that the occurrence of an underflow exception may depend on the setting of the enable bit.)

A single instruction, other than **mtfsfi** or **mtfsf**, may set multiple exception bits only in the following cases:

- Inexact exceptions may be set with overflow exception.
- Inexact exceptions may be set with underflow exception.
- Invalid operation exceptions (SNaN) are set with invalid operation exception (∞×0) for multiply-add instructions for which the values being multiplied are infinity and zero and the value being added is an SNaN.
- Invalid operation exceptions (SNaN) may be set with invalid operation exception (invalid compare) for compare ordered instructions.
- Invalid operation exceptions (SNaN) may be set with invalid operation exception (invalid integer convert) for convert to integer instructions.

When an exception occurs, the writing of a result to the target register may be suppressed or a result may be delivered, depending on the exception.

The writing of a result to the target register is suppressed for the following kinds of exception, so that there is no possibility that one of the operands is lost:

- Enabled invalid operation
- Enabled zero divide

For the remaining kinds of exception, a result is generated and written to the destination specified by the instruction causing the exception. The result may be a different value for the enabled and disabled conditions for some of these exceptions. The kinds of exception that deliver a result are the following:

- Disabled invalid operation

- Disabled zero divide
- Disabled overflow
- Disabled underflow
- Disabled inexact
- Enabled overflow
- Enabled underflow
- Enabled inexact

Subsequent sections define each of the floating-point exceptions and specify the action that is taken when they are detected.

IEEE Std. 754 specifies the handling of exceptional conditions in terms of “traps” and “trap handlers.” In this architecture, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE Std. 754 for the trap-enabled case; software is expected to detect the exception and revise the result. An FPSCR exception enable bit of 0 causes generation of the default-result value specified for the trap-disabled (or no-trap-occurs or trap-not-implemented) case; the expectation is that the exception is not detected by software, which uses the default result. The result delivered in each case, for each exception, is described in the following sections.

When an exception occurs, the IEEE Std. 754 default behavior is to generate a default value without notifying software. In this architecture, this behavior is desired for all exceptions; all FPSCR exception enable bits should be cleared and ignore exceptions mode (see below) should be used. In this case, the floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur; software can inspect the FPSCR exception bits to determine whether exceptions occurred.

In this architecture, if software is to be notified that a given kind of exception occurred, the corresponding FPSCR enable bit must be set and a mode other than ignore exceptions mode must be used. In this case, the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. The system floating-point enabled exception error handler is also invoked if a move to FPSCR instruction causes an exception bit and the corresponding enable bit both to be 1; the move to FPSCR instruction is considered to cause the enabled exception.

MSR[FE0,FE1] control whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs. In EIS, only precise mode is supported and if MSR[FE0,FE1] are non-zero, precise mode is used. (The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception.) PowerISA defines settings for MSR[FE0,FE1] that include imprecise modes which are not implemented by EIS.

The effects of the four possible settings of these bits in Power ISA are described in this table.

Table 4-25. Power ISA FE0 and FE1 Configuration Descriptions

FE0	FE1	Description
0	0	Ignore exceptions mode. Floating-point exceptions do not cause the system floating-point enabled exception error handler to be invoked.
0	1	Imprecise nonrecoverable mode. The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results produced by the excepting instruction may have been used by or may have affected subsequent instructions that are executed before the error handler is invoked. EIS treats this setting to be the same as precise mode.
1	0	Imprecise recoverable mode. The system floating-point enabled exception error handler is invoked at some point at or beyond the instruction that caused the enabled exception. Sufficient information is provided to the error handler that it can identify the excepting instruction and the operands, and correct the result. No results produced by the excepting instruction have been used by or have affected subsequent instructions that are executed before the error handler is invoked. EIS treats this setting to be the same as precise mode.
1	1	Precise mode. The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

In all cases, the question of whether a floating-point result is stored, and what value is stored, is governed by the FPSCR exception enable bits, as described in subsequent sections, and is not affected by the value of the FE0 and FE1 bits.

In all cases in which the system floating-point enabled exception error handler is invoked, all instructions before the instruction at which the system floating-point enabled exception error handler is invoked have completed, and no instruction after the instruction at which the system floating-point enabled exception error handler is invoked has begun execution. The instruction at which the floating-point enabled program exception error handler is invoked has completed if it is the excepting instruction and there is only one such instruction. Otherwise it has not begun execution (or may have been partially executed in some cases, as described in Section 7.8.7, “Program Interrupt”).

NOTE: Software Considerations

- In any nonprecise mode, an FPSCR instruction can be used to force any exceptions, due to instructions initiated before the FPSCR instruction, to be recorded in the FPSCR. (This forcing is superfluous for precise mode.)
- In either imprecise mode, an FPSCR instruction can be used to force any invocations of the system floating-point enabled exception error handler, due to previously initiated instructions, to occur. (This has no effect in ignore exceptions mode and is superfluous for precise mode.)
- The last sentence of the paragraph preceding this note applies only for imprecise modes or if the mode has been changed from ignore exceptions mode. (It always applies in the latter case.)

For best performance across the widest range of implementations, follow these guidelines:

- If the IEEE-754 default results are acceptable to the application, use ignore exceptions mode with all FPSCR exception enable bits cleared.
- If the IEEE-754 default results are not acceptable, use imprecise nonrecoverable mode (or imprecise recoverable mode if recoverability is needed) with FPSCR exception enable bits set for those exceptions that would invoke the system floating-point enabled exception error handler.
- Ignore exceptions mode should not, in general, be used if any FPSCR exception enable bits are set.
- Precise mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

NOTE: Software Considerations

> In general, most processors implement a precise recoverable mode for FE0, FE1 regardless of whether it is set to use an imprecise mode. Consult the core reference manual.

4.6.1.7.1 Invalid Operation Exception

An invalid operation exception occurs when an operand is invalid for one the following operations:

- Any floating-point operation on a Signaling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities ($\infty - \infty$)
- Division of infinity by infinity ($\infty \div \infty$)
- Division of zero by zero ($0 \div 0$)
- Multiplication of infinity by zero ($\infty \times 0$)
- Ordered comparison involving a NaN (invalid compare)
- Square root or reciprocal square root of a negative (and nonzero) number (invalid square root)
- Integer convert involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN (invalid integer convert)

An invalid operation exception also occurs when an **mtfsfi**, **mtfsf**, or **mtfsb1** executes that sets FPSCR[VXSOFT] (software-defined condition).

The action to be taken depends on the setting of the invalid operation exception enable bit of the FPSCR.

When an invalid operation exception is enabled (FPSCR[VE]=1) and an invalid operation exception occurs, the following actions are taken:

1. One or two invalid operation exceptions are set

FPSCR[VXSNAN]	(if SNaN)
FPSCR[VXISI]	(if $\infty - \infty$)
FPSCR[VXIDI]	(if $\infty \div \infty$)
FPSCR[VXZDZ]	(if $0 \div 0$)
FPSCR[VXIMZ]	(if $\infty \times 0$)
FPSCR[VXVC]	(if invalid comparison)
FPSCR[VXSOFT]	(if software-defined condition)

FPSCR[VXSQRT]	(if invalid square root)
FPSCR[VXCVI]	(if invalid integer convert)

2. If the operation is an arithmetic, floating round to single-precision, floating round to integer, or convert to integer operation, the following occurs:
   - — The target FPR is unchanged
   - — FPSCR[FR,FI] are cleared
   - — FPSCR[FPRF] is unchanged
3. If the operation is a compare, the following occurs:
   - — FPSCR[FR,FI,C] are unchanged
   - — FPSCR[FPCC] is set to reflect unordered
4. If an **mtfsfi**, **mtfsf**, or **mtfsb1** instruction is executed that sets FPSCR[VXSOFT], the FPSCR is set as specified in the instruction description.

When an invalid operation exception is disabled (FPSCR[VE]=0) and an invalid operation exception occurs, the following actions are taken:

1. One or two invalid operation exceptions are set:

FPSCR[VXSNAN]	(if SNaN)
FPSCR[VXISI]	(if ∞ - ∞)
FPSCR[VXIDI]	(if ∞ ÷ ∞)
FPSCR[VXZDZ]	(if 0 ÷ 0)
FPSCR[VXIMZ]	(if ∞ × 0)
FPSCR[VXVC]	(if invalid comparison)
FPSCR[VXSOFT]	(if software-defined condition)
FPSCR[VXSQRT]	(if invalid square root)
FPSCR[VXCVI]	(if invalid integer convert)

2. If the operation is an arithmetic or floating round to single-precision operation,
   - — the target FPR is set to a Quiet NaN.
   - — FPSCR[FR, FI] are cleared.
   - — FPSCR[FPRF] is set to indicate the class of the result (Quiet NaN).
3. If the operation is a convert to 64-bit integer operation, the target FPR is set as follows:
   - — **fr**D is set to the most positive 64-bit integer if the operand in frB is a positive number or **+** ∞, and to the most negative 64-bit integer if the operand in frB is a negative number, - ∞, or NaN.
   - — FPSCR[FR, FI] are cleared.
   - — FPSCR[FPRF] is undefined.
4. If the operation is a convert to 32-bit integer operation, the target FPR is set as follows:
   - — **fr**$D_{0:31}$ ← undefined
     fr$D_{32:63}$ are set to the most positive 32-bit integer if the operand in **fr**B is a positive number or +infinity, and to the most negative 32-bit integer if the operand in **fr**B is a negative number, -infinity, or NaN

- — FPSCR[FR, FI] are cleared.
- — FPSCR[FPRF] is undefined.

5. If the operation is a compare, then
   - — FPSCR[FR, FI, C] are unchanged.
   - — FPSCR[FPCC] is set to reflect unordered.
6. If an **mtfsfi**, **mtfsf**, or **mtfsb1** instruction is executed that sets FPSCR[VXSOFT], FPSCR is set as specified in the instruction description.

4.6.1.7.2 Zero Divide Exception

A zero divide exception occurs when a divide instruction is executed with a zero divisor value and a finite nonzero dividend value. It also occurs when a reciprocal estimate instruction (**fre**[**s**] or **frsqrte**[**s**]) is executed with an operand value of zero.

The action to be taken depends on the setting of the zero divide exception enable bit, FPSCR[ZE]. If FPSCR[ZE]=1 and a zero divide exception occurs, the following actions are taken:

1. Zero divide exception is set: FPSCR[ZX] = 1.
2. The target FPR is unchanged; FPSCR[FR, FI] are cleared.
3. FPSCR[FPRF] is unchanged.

When zero divide exception is disabled (FPSCR[ZE]=0) and a zero divide exception occurs, the following actions are taken:

1. Zero divide exception is set; FPSCR[ZX] ⤴ 1.
2. The target FPR is set to ± Infinity, where the sign is determined by the XOR of the signs of the operands.
3. FPSCR[FR,FI] are cleared.
4. FPSCR[FPRF] is set to indicate the class and sign of the result (± Infinity).

4.6.1.7.3 Overflow Exception

An overflow exception occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

The action taken depends on the setting of the overflow exception enable bit of the FPSCR.

When overflow exception is enabled (FPSCR[OE]=1) and an overflow exception occurs, the following actions are taken:

1. Overflow exception is set: FPSCR[OX] =1.
2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536.
3. For single-precision arithmetic instructions and the floating round to single-precision instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192.
4. The adjusted rounded result is placed into the target FPR.
5. FPSCR[FPRF] is set to indicate the class and sign of the result (± Normal Number).

When overflow exception is disabled (FPSCR[OE]=0) and an overflow exception occurs, the following actions are taken:

1. Overflow exception is set: FPSCR[OX] = 1.
2. Inexact exception is set: FPSCR[XX] = 1.
3. The result is determined by the rounding mode (FPSCR[RN]) and the sign of the intermediate result as follows:
   - Round to Nearest
     Store ± Infinity, where the sign is the sign of the intermediate result.
   - Round toward Zero
     Store the format's largest finite number with the sign of the intermediate result.
   - Round toward **+** Infinity
     For negative overflow, store the format's most negative finite number; for positive overflow, store +Infinity.
   - Round toward - Infinity
     For negative overflow, store -Infinity; for positive overflow, store the format's largest finite number.
4. The result is placed into the target FPR.
5. FPSCR[FR] is undefined.
6. FPSCR[FI] is set.
7. FPSCR[FPRF] is set to indicate the class and sign of the result (± Infinity or ± Normal Number).

4.6.1.7.4 Underflow Exception

Underflow exception is defined separately for the enabled and disabled states:

- Enabled: Underflow occurs when the intermediate result is "Tiny."
- Disabled: Underflow occurs when the intermediate result is "Tiny" and there is "Loss of Accuracy."

A "Tiny" result is detected before rounding, when a nonzero intermediate result computed as though both the precision and the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is "Tiny" and the underflow exception is disabled (FPSCR[UE]=0) then the intermediate result is denormalized (see Section 3.4.3.4, "Normalization and Denormalization") and rounded (see Section 3.4.3.6, "Rounding") before being placed into the target FPR.

"Loss of Accuracy" is detected when the delivered result value differs from what would have been computed were both the precision and the exponent range unbounded.

The action to be taken depends on the setting of the underflow exception enable bit of the FPSCR.

When underflow exception is enabled (FPSCR[UE]=1) and an underflow exception occurs, the following actions are taken:

1. Underflow exception is set: FPSCR[UX] = 1.

2. For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by adding 1536.
3. For single-precision arithmetic instructions and the floating round to single-precision instruction, the exponent of the normalized intermediate result is adjusted by adding 192.
4. The adjusted rounded result is placed into the target FPR.
5. FPSCR[FPRF] is set to indicate the class and sign of the result (± Normalized Number).

NOTE: Software Considerations

> The FR and FI bits are provided to allow the system floating-point enabled exception error handler, when invoked because of an underflow exception, to simulate a “trap disabled” environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

When underflow exception is disabled (FPSCR[UE]=0) and an underflow exception occurs, the following actions are taken:

1. Underflow exception is set. FPSCR[UX] = 1.
2. The rounded result is placed into the target FPR.
3. FPSCR[FPRF] is set to indicate the class and sign of the result (± Normalized Number, ± Denormalized Number, or ± Zero).

4.6.1.7.5 Inexact Exception

An inexact exception occurs when one of two conditions occur during rounding:

- The rounded result differs from the intermediate result assuming both the precision and the exponent range of the intermediate result to be unbounded. In this case the result is said to be inexact. (If the rounding causes an enabled overflow exception or an enabled underflow exception, an inexact exception also occurs only if the significands of the rounded result and the intermediate result differ.)
- The rounded result overflows and overflow exception is disabled.

The action to be taken does not depend on the setting of the inexact exception enable bit of the FPSCR.

When an inexact exception occurs, the following actions are taken:

1. Inexact exception is set: FPSCR[XX] = 1.
2. The rounded or overflowed result is placed into the target FPR.
3. FPSCR[FPRF] is set to indicate the class and sign of the result.

NOTE: Software Considerations

> In some implementations, enabling inexact exceptions may degrade performance more than does enabling other types of floating-point exception.

4.6.1.8 Branch and Flow Control Instructions

Some branch instructions can redirect instruction execution conditionally, based on the value of bits in the CR.

4.6.1.8.1 Branch Instruction Address Calculation

Branch instructions can alter the sequence of instruction execution. Instruction addresses are always assumed to be word-aligned; processors ignore the two low-order bits of the generated branch target address.

<Category VLE>:
Instruction addresses are assumed to be half-word aligned; processors ignore the low-order bit of the generated branch target address.

Branch instructions compute the EA of the next instruction address using the following addressing modes:

- Branch relative
- Branch conditional to relative address
- Branch to absolute address
- Branch conditional to absolute address
- Branch conditional to link register (LR)
- Branch conditional to count register (CTR)

4.6.1.8.2 Branch Relative Addressing Mode

Instructions that use branch relative addressing generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand LI, and adding the resultant value to the current instruction address. Branches using this mode have the absolute addressing option disabled (AA field, bit 30, in the instruction encoding = 0). The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This causes the EA of the instruction following the branch instruction to be placed in the LR.

This figure shows how the branch target address is generated using this mode.

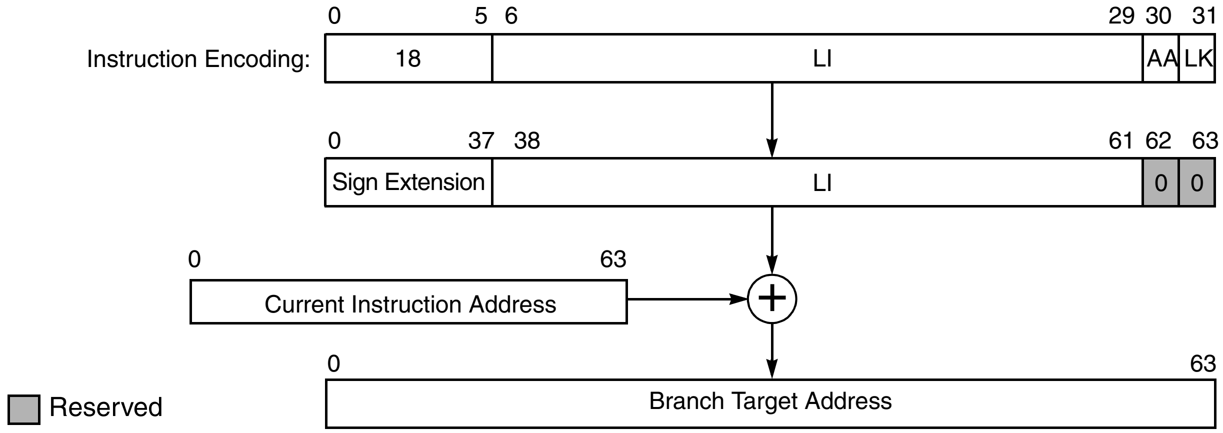


Figure 4-4. Branch Relative Addressing

4.6.1.8.3 Branch Conditional to Relative Addressing Mode

If branch conditions are met, instructions that use the branch conditional to relative addressing mode generate the next instruction address by sign extending and appending results to the immediate displacement operand (BD) and adding the resultant value to the current instruction address. Branches using this mode have the absolute addressing option disabled (AA field, bit 30, in the instruction encoding = 0). The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR.

This figure shows how the branch target address is generated using this mode.

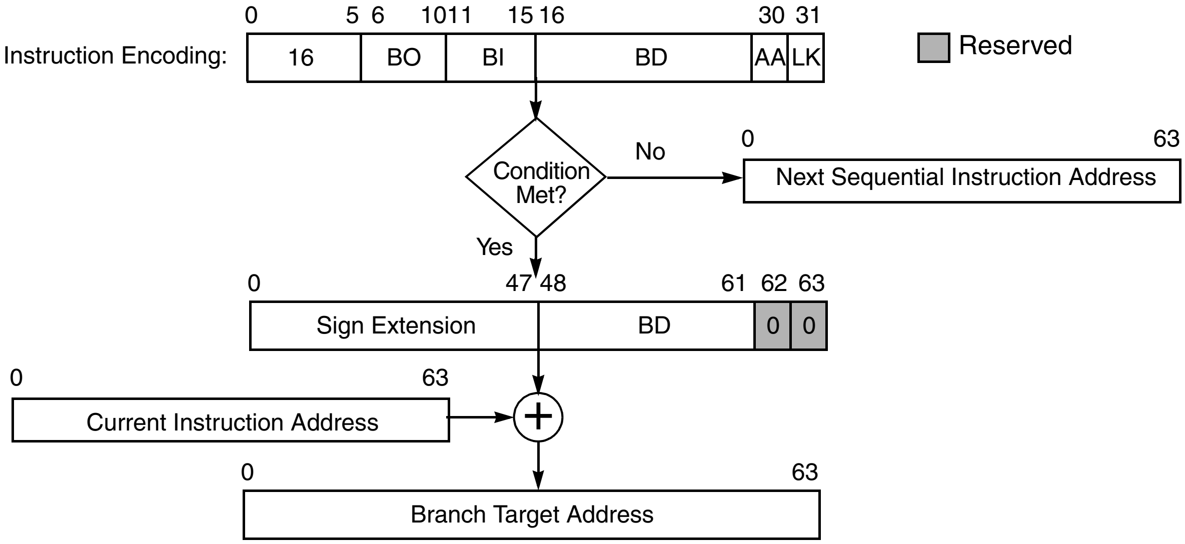


Figure 4-5. Branch Conditional Relative Addressing

4.6.1.8.4 Branch to Absolute Addressing Mode

Instructions that use branch to absolute addressing mode generate the next instruction address by sign extending and appending 0b00 to the LI operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit 30, in the instruction encoding = 1). The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR.

This figure shows how the branch target address is generated using this mode.

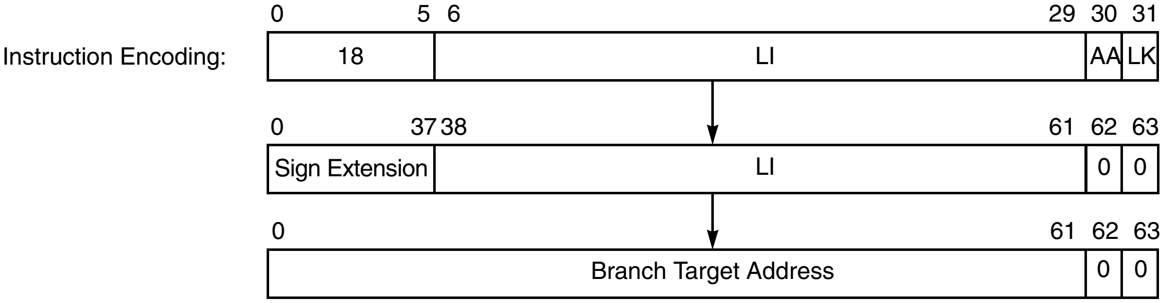


Figure 4-6. Branch to Absolute Addressing

4.6.1.8.5 Branch Conditional to Absolute Addressing Mode

If the branch conditions are met, instructions that use the branch conditional to absolute addressing mode generate the next instruction address by sign extending and appending 0b00 to the BD operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit 30, in the instruction encoding = 1). The LR update option can be enabled (bit 31 (LK) in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR.

This figure shows how the branch target address is generated using this mode.

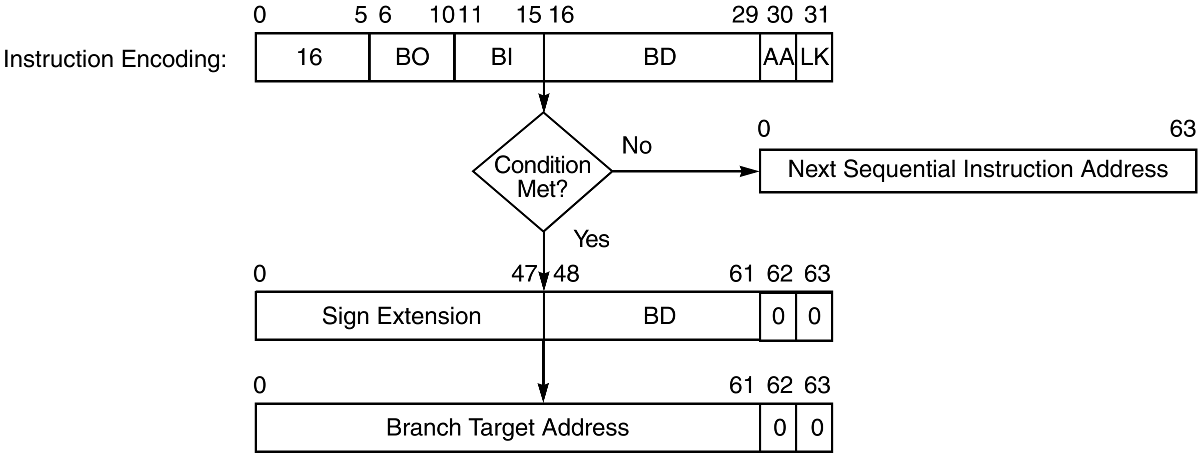


Figure 4-7. Branch Conditional to Absolute Addressing

4.6.1.8.6 Branch Conditional to Link Register Addressing Mode

If the branch conditions are met, the branch conditional to LR instruction generates the next instruction address by fetching the contents of the LR and clearing the two low-order bits to zero. The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR.

This figure shows how the branch target address is generated using this mode.

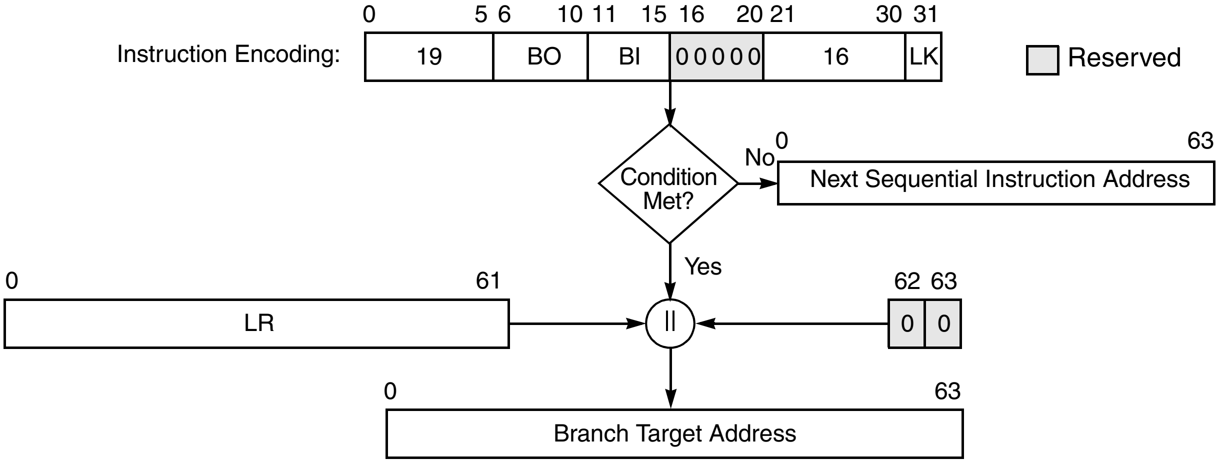


Figure 4-8. Branch Conditional to Link Register Addressing

4.6.1.8.7 Branch Conditional to Count Register Addressing Mode

If the branch conditions are met, the branch conditional to count register instruction generates the next instruction address by fetching the contents of the count register (CTR) and clearing the two low-order bits to zero. The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the EA of the instruction following the branch instruction to be placed in the LR.

This figure shows how the branch target address is generated when using this mode.

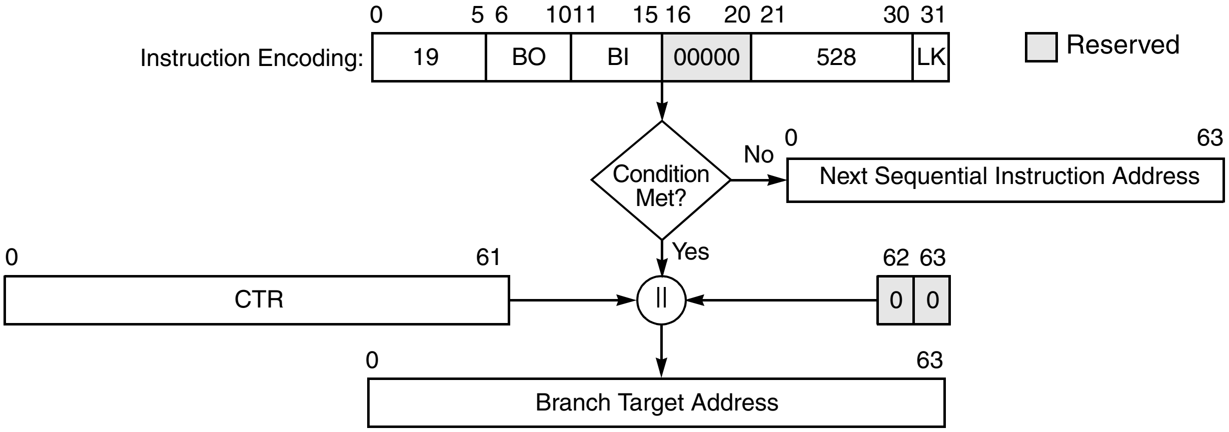


Figure 4-9. Branch Conditional to Count Register Addressing

4.6.1.9 Conditional Branch Control

NOTE: Software Considerations

> Some processors do not implement the static branch prediction defined in the architecture and described here. Consult the core reference manual.

For branch conditional instructions, the BO operand specifies the conditions under which the branch is taken. The first four bits of the BO operand specify how the branch is affected by or affects the condition and count registers. The fifth bit, shown in Table 4-26 as having the value *t*, is used by some implementations for branch prediction as described below.

Figure 4-10. BO Bit Descriptions

BO Bits	Description
0	Setting this bit causes the CR bit to be ignored.
1	Bit value to test against
2	Setting this causes the decrement to not be decremented.
3	Setting this bit reverses the sense of the CTR test.
4	Used for the *t* bit, which provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

The encodings for the BO operands are shown in this table.

Table 4-26. BO Operand Encodings

BO	Description
0000*z*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is FALSE.
0001*z*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001*at*	Branch if the condition is FALSE.
0100*z*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is TRUE.
0101*z*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011*at*	Branch if the condition is TRUE.
1*a*00*t*	Decrement the CTR, then branch if the decremented CTR ≠ 0.
1*a*01*t*	Decrement the CTR, then branch if the decremented CTR = 0.
1*z*1*zz*	Branch always.

In this table, *z* indicates a bit that is ignored. Note that the *z* bits should be cleared, as they may be assigned a meaning in some future version of the architecture.
The *a* and *t* bits provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

The *a* and *t* bits of the BO field can be used by software to provide a hint about whether the branch is likely to be taken or is likely not to be taken as shown in this table:

Table 4-27. *at* Bit Encodings

at	Hint
00	No hint is given
01	Reserved
10	The branch is very likely not to be taken
11	The branch is very likely to be taken

For **bclr**[**l**] and **bcctr**[**l**] instructions, the BH field provides a hint about the use of the instruction as shown in this table:

Table 4-28. BH Field Encodings

BH	Instruction	Hint
00	**bclr**[**l**]	The instruction is a subroutine return
	bcctr[**l**]	The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken

Table 4-28. BH Field Encodings (continued)

BH	Instruction	Hint
01	**bclr**[**l**]	The instruction is not a subroutine return; the target address is likely to be the same as the target address used the preceding time the branch was taken
	bcctr[**l**]	Reserved
10	Reserved	
11	**bclr**[**l**]	The target address is not predictable
11	**bcctr**[**l**]	The target address is not predictable

The hint provided by the BH field is independent of the hint provided by the *at* bits.

NOTE: Software Considerations

Many processors have dynamic methods of predicting whether a branch will be taken or not. Some processors maintain a "link stack" to dynamically predict subroutine calls. Since the dynamic prediction is likely to be very accurate, such implementations will generally ignore static branch prediction hints.

NOTE: Architecture

Previous versions of the architecture (going back to the original PowerPC architecture), used a "y" bit as the 5th bit of the BO field for static branch prediction. The "at" method is backward compatible with software that used the old method.

The 5-bit BI operand in branch conditional instructions specifies which CR bit represents the condition to test. The CR bit selected is BI +32, as shown in Table 3-13.

If the branch instructions contain immediate addressing operands, the target addresses can be computed sufficiently ahead of the branch instruction that instructions can be fetched along the target path. If the branch instructions use the link and count registers, instructions along the target path can be fetched if the link or count register is loaded sufficiently ahead of the branch instruction.

Branching can be conditional or unconditional, and optionally a branch return address is created by storing the EA of the instruction following the branch instruction in the LR after the branch target address has been computed. This is done regardless of whether the branch is taken.

4.6.1.9.1 Using a Link Register Stack

Some processors may keep a stack of the LR values most recently set by branch and link instructions, with the possible exception of the form shown below for obtaining the address of the next instruction. To benefit from this stack, the following programming conventions should be used.

In the following examples, let A, B, and Glue represent subroutine labels:

- Obtaining the address of the next instruction—use the following form of branch and link: bcl **20,31,$+4**
- Loop counts: Keep them in the count register, and use one of the branch conditional instructions to decrement the count and to control branching (for example, branching back to the start of a loop if the decremented counter value is nonzero).
- Computed GOTOs, case statements, and so on: Use the count register to hold the address to branch to, and use the **bcctr** instruction with the link register option disabled (LK = 0) and BH = 0b11 if appropriate, to branch to the selected address.
- Direct subroutine linkage—where A calls B and B returns to A. The two branches should be as follows:
  - — A calls B: use a branch instruction that enables the link register (LK = 1).
  - — B returns to A: use the **bclr** instruction with the link register option disabled (LK = 0) (the return address is in, or can be restored to, the link register).
- Indirect subroutine linkage: Where A calls Glue, Glue calls B, and B returns to A rather than to Glue. (Such a calling sequence is common in linkage code used when the subroutine that the programmer wants to call, here B, is in a different module from the caller: the binder inserts "glue" code to mediate the branch.) The three branches should be as follows:
  - — A calls Glue: use a branch instruction that sets the link register with the link register option enabled (LK = 1).
  - — Glue calls B: place the address of B in the count register, and use the **bcctr** instruction with the link register option disabled (LK = 0).
  - — B returns to A: use the **bclr** instruction with the link register option disabled (LK = 0) (the return address is in, or can be restored to, the link register).

4.6.1.9.2 Branch Instructions

This table lists branch instructions. A set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions. See Section C.4, "Branch Instruction Simplified Mnemonics."

Table 4-29. Branch Instructions

Name	Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI

NOTE

The Integer Select instruction, **isel**, can be used to handle sequences with multiple conditional branches more efficiently. A detailed description including syntax and an example of how **isel** can be used can be found Section 4.6.1.9.5, “Integer Select Instruction.”

4.6.1.9.3 Condition Register (CR) Logical Instructions

CR logical instructions, shown in the following table, and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Table 4-30. Condition Register Logical Instructions

Name	Mnemonic	Syntax
Condition Register AND	**crand**	**crb**D,**crb**A,**crb**B
Condition Register OR	**cror**	**crb**D,**crb**A,**crb**B
Condition Register XOR	**crxor**	**crb**D,**crb**A,**crb**B
Condition Register NAND	**crnand**	**crb**D,**crb**A,**crb**B
Condition Register NOR	**crnor**	**crb**D,**crb**A,**crb**B
Condition Register Equivalent	**creqv**	**crb**D,**crb**A,**crb**B
Condition Register AND with Complement	**crandc**	**crb**D,**crb**A,**crb**B
Condition Register OR with Complement	**crorc**	**crb**D,**crb**A,**crb**B
Move Condition Register Field	**mcrf**	**crD**,**crf**S

4.6.1.9.4 Trap Instructions

The trap instructions, shown in the following table, test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, a trap type program interrupt is taken. For more information, see Section 7.8.7, “Program Interrupt.” If the tested conditions are not met, instruction execution continues normally. See Appendix C, “Simplified Mnemonics.”

Table 4-31. Trap Instructions

Name	Mnemonic	Syntax
Trap Doubleword Immediate <64>	**tdi**	TO,**r**A,SIMM
Trap Doubleword <64>	**td**	TO,**r**A,**r**B
Trap Word Immediate	**twi**	TO,**r**A,SIMM
Trap Word	**tw**	TO,**r**A,**r**B

4.6.1.9.5 Integer Select Instruction

The Integer Select instruction, **isel**, shown in the following table, is a conditional register move that helps eliminate short branches often found in control code, which is common in embedded applications, is characterized by unpredictable short branches. When mispredicted, such branches cause long pipeline delays.

Table 4-32. Integer Select Instruction

Name	Mnemonic	Syntax
Integer Select	**isel**	**r**D,**r**A,**r**B,**cr**B

The **isel** instruction can be used to handle short conditional branch segments more efficiently. It uses two source registers and one destination register. Under the control of a specified condition code bit, it copies one or the other source operand to the destination, as follows:

```
if crB then
        rD = rA
else
        rD = rB
```

The **isel** instruction allows more efficient implementation of a condition sequence such as the one in the following generic example:

```
int16 global1,…, global37,...;
....
void procedure17(int16 parm) {
   if  (global1 == 27) {
       global37 = parm + 17;
   }
   else {
       global37 = parm - 17;
   }
}
```

This table shows a coding example with and without the **isel** instruction.

Table 4-33. Recoding with isel

Code Sequence without isel	Code Sequence with isel

`cmpi cr3, r17, 27;`
`bne cr3, NotEqual;`
`addi r15, r17, 17;`
`jmp Assign;`
`NotEqual:`
`     addi r15, r17, -17;`
`Assign:`
`     stw r15, (rGlobals + g37);`
	`cmpi cr3, r17, 27;`
	`addi r15, r17, 17;`
	`addi r16, r17, -17;`
	`isel r15, r15, r16, cr3.eq`
	`stw r15, (rGlobals + g37);`

The sequence without **isel** turns conditional branches into a code sequence that sets a condition code according to the results of a computation. It uses a conditional branch to choose a target sequence, but needs an unconditional branch for the IF clause. The conditional branch is often hard to predict, the code sequences are generally small, and the resulting throughput is typically low.

The sequence using **isel** does the following:

- Sets a condition code according to the results of a comparison
- Has code that executes both the IF and the ELSE segments
- Copies the results of one segment to the desired destination register

- Works well for small code segments and for unpredictable branches
- Can reduce code size

4.6.1.10 System Linkage Instruction

The System Call (**sc**) instruction permits a program to call on the system to perform a service or an operating system to call on the hypervisor to perform a service. Table 4-34 lists the system linkage instruction. Executing **sc** with LEV = 0, produces a normal system call to the operating system. Executing **sc** with LEV = 1 should be used by operating systems to call on the hypervisor <E.HV>.

Table 4-34. System Linkage Instruction

Name	Mnemonic	Syntax
System Call	**sc**	LEV

Executing **sc** invokes the system call interrupt handler or the hypervisor system call interrupt handler depending on the value of the LEV. For more information, see Section 7.8.9, “System Call Interrupt,” and Section 7.8.26, “Embedded Hypervisor System Call Interrupt <E.HV>.”

An **sc** instruction without the LEV field is treated by the assembler as an **sc** with LEV = 0 <E.HV>.

NOTE: Software Considerations

A user-level program should not invoke **sc** with LEV=1, unless the user-level program is explicitly intended to execute directly under the control of the hypervisor.

NOTE: Software Considerations

A hypervisor should treat the execution of an **sc** with LEV=1 from a user-level program as an illegal instruction unless the partition which executed the instruction is expected by the hypervisor to perform system calls directly to the hypervisor.

4.6.1.11 Embedded Hypervisor Privilege Instruction <E.HV>

The Embedded Hypervisor Privilege (**ehpriv**) instruction causes an embedded hypervisor privilege exception and subsequent interrupt. ehpriv is used by the hypervisor to provide an emulation capability for virtual instructions defined by the hypervisor. The OC field is not interpreted by hardware but is for the use of the hypervisor to provide specific emulation.

This table lists the embedded hypervisor privilege instruction.

Table 4-35. Embedded Hypervisor Privilege Instruction

Name	Mnemonic	Syntax
Embedded Hypervisor Privilege	**ehpriv**	OC

NOTE: Software Considerations

If the hypervisor creates virtual instructions, it should defined extended mnemonics for such instructions in the assembler that encode into the OC field.

4.6.1.12 Debug Instruction

The Debugger Notify Halt (**dnh**) instruction (see Table 4-36) is a user-level instruction that provides the means for the transfer of information between the processor and an implementation-dependent external debug facility. If the processor is in external debug mode and is configured to halt on the execution of **dnh**, **dnh** causes the processor to stop fetching and executing instructions, entering an external debug halt state. If the processor is not in external debug mode, executing dnh causes an illegal instruction exception.

Table 4-36. Debug Instruction

Name	Mnemonic	Syntax
Debugger Notify Halt	**dnh**	DUI,DUIS

4.6.1.13 Processor Control Instructions

User-level processor control instructions are used to read from and write to the CR and special-purpose registers (SPRs), as well as the **wait** instruction.

4.6.1.13.1 Move to/from Condition Register Instructions

This table summarizes the instructions for reading from or writing to the CR.

Table 4-37. Move to/from Condition Register Instructions

Name	Mnemonic	Syntax
Move to Condition Register Fields	**mtcrf**	CRM,**r**S
Move to Condition Register from XER	**mcrxr**	**cr**D
Move from Condition Register	**mfcr**	**r**D
Move from One Condition Register Field	**mfocrf**	**r**D,FXM
Move to One Condition Register Field	**mtocrf**	FXM,**r**S

4.6.1.13.2 Move to/from Special-Purpose Register Instructions

This table lists the **mtspr** and **mfspr** instructions.

Table 4-38. Move to/from Special-Purpose Register Instructions

Name	Mnemonic	Syntax
Move to Special-Purpose Register	**mtspr**	SPR,**r**S
Move from Special-Purpose Register	**mfspr**	**r**D,SPR
Move from Time Base	**mftb**	**r**D,TBRN **r**D,SPR

See Section 3.2.2, “Special-Purpose Registers (SPRs)” for a complete list of SPRs defined by EIS and their access levels.

4.6.1.13.3 Wait for Interrupt Instruction

The **wait** instruction stops synchronous processor activity including the fetching of instructions until an asynchronous interrupt, a debug post-completion (ICMP) interrupt occurs, or the event specified by the WC operand occurs. A processor typically uses this to reduce power consumption.

Table 4-39. Wait for Interrupt Instruction

Name	Mnemonic	Syntax
Wait for Interrupt	**wait**	WC,WH

Not all processors implement events defined by the WC operand or allow the hint from the WH operand.

4.6.1.14 Performance Monitor Instructions

The performance monitor provides read-only, user-level access to some performance monitor resources. Instructions are listed in this table.

Table 4-40. Performance Monitor Instructions

Name	Mnemonic	Syntax
Move from Performance Monitor Register	**mfpmr**	**r**D,PMRN

The user-level PMRs listed in Section 3.15.4, “Performance Monitor Counter Registers (PMCn/UPMCn)” are accessed with **mfpmr**. Attempting to write user-level PMRs in either mode causes an illegal instruction exception.

4.6.1.15 Memory Synchronization Instructions

Memory synchronization instructions control the order in which memory operations complete with respect to asynchronous events and the order in which memory operations are seen by other mechanisms that access memory. Further information about memory barriers and storage ordering is described in Section 6.4.8, “Shared Memory.”

This table lists the memory synchronization instructions.

Table 4-41. Memory Synchronization Instructions

Name	Mnemonic	Syntax	Notes
Instruction Synchronize	**isync**	—	**isync** is used to perform context serialization for the instruction stream. **isync** waits for previous instructions (including any interrupts they generate) to complete before **isync** executes, which purges all instructions from the processor and refetches the next instruction. **isync** does not wait for pending stores to complete.
Memory Barrier	**mbar**	MO	**mbar** provides a pipelined memory barrier. (Note that **mbar** uses the same opcode as **eieio** from the PowerPC architecture.) The behavior of **mbar** depends on the MO operand. MO = 0—**mbar** behaves identically to **sync 0**. MO = 1—**mbar** is a weaker, faster memory barrier that guarantees the same ordering as defined by the **eieio** instruction from the PowerPC architecture.
Synchronize	**sync**	L	**sync** provides an ordering function for the effects of all instructions executed by the processor executing the **sync**. Executing **sync** ensures that all previous instructions complete before it completes and that no subsequent instructions are initiated until after it completes. It also creates a memory barrier, which orders the storage accesses associated with these instructions. The behavior of **sync** depends on the L operand: L = 0—**sync** behaves as a heavyweight sync (extended mnemonic **hwsync** or **sync** with no operand) in which a memory barrier is created which orders all memory accesses. L =1—**sync** behaves as a lightweight sync (extended mnemonic **lwsync**) in which a lighter weight memory barrier is created which only orders storage accesses for a specific set of cacheable memory accesses. Previous versions of the architecture used the mnemonic **msync** for **sync**. **msync** is now an extended mnemonic for **sync 0**.

4.6.1.16 Atomic Update Primitives Using Load and Reserve and Store Conditional instructions

The load and reserve and store conditional instructions together permit atomic update of a memory location. The architecture provides byte, half word, word, and double word forms of these instructions. For the purposes of this section, the operation of the word forms **lwarx** and **stwcx.** are described here, although any of the other memory sizes may be used.

A specified memory location that may be modified by other processors or mechanisms requires memory coherence. If the location is in write-through required or caching-inhibited memory, the implementation determines whether these instructions function correctly or cause the system data storage error handler to be invoked. Consult the user documentation.

NOTE:

> The memory coherence required attribute on other processors and mechanisms ensures that their stores to the specified location will cause the reservation created by the **lwarx** to be cancelled.

A **lwarx** instruction is a load from a word-aligned location with the following side effects:

- A reservation for a subsequent **stwcx.** instruction is created.
- The memory coherence mechanism is notified that a reservation exists for the location accessed by the **lwarx**.

The **stwcx.** is a store to a word-aligned location that is conditioned on the existence of the reservation created by the **lwarx** and on whether both instructions specify the same location. To emulate an atomic operation, both **lwarx** and **stwcx.** must access the same location. The **lwarx** and **stwcx.** are ordered by a dependence on the reservation, and the program is not required to insert other instructions to maintain the order of memory accesses caused by these two instructions.

A **stwcx.** performs a store to the target location only if the location accessed by the **lwarx** that established the reservation has not been stored into by another processor or mechanism between supplying a value for the **lwarx** and storing the value supplied by the **stwcx.**. If the instructions specify different locations, the store is not necessarily performed. CR0 is modified to indicate whether the store was performed, as follows:

- CR0[LT,GT,EQ,SO] = 0b00 || store_performed || XER[SO]

If a **stwcx.** completes but does not perform the store because a reservation no longer exists, CR0 is modified to indicate that the **stwcx.** completed without altering memory.

A **stwcx.** that performs its store is said to succeed.

Examples using **lwarx** and **stwcx.** are given in Appendix D, “Programming Examples.”

A successful **stwcx.** to a given location may complete before its store has been performed with respect to other processors and mechanisms. As a result, a subsequent load or **lwarx** from the given location on another processor may return a stale value. However, a subsequent **lwarx** from the given location on the other processor followed by a successful **stwcx.** on that processor is guaranteed to have returned the value stored by the first processor’s **stwcx.** (in the absence of other stores to the given location).

4.6.1.16.1 Reservations

The ability to emulate an atomic operation using **lwarx** and **stwcx.** is based on the conditional behavior of **stwcx.**, the reservation set by **lwarx**, and the clearing of that reservation if the target location is modified by another processor or mechanism before the **stwcx.** performs its store.

A reservation is held on an aligned unit of real memory called a reservation granule. The size of the reservation granule is implementation-dependent, but is a multiple of four bytes for **lwarx**. The reservation granule associated with EA contains the real address to which the EA maps. (In the RTL for the load and reserve and store conditional instructions, 'real_addr(EA)' stands for 'real address to which EA maps.') When one processor holds a reservation and another processor performs a store, the first processor's reservation is cleared when the store affects any bytes in the reservation granule.

NOTE: Software Considerations

> One use of **lwarx** and **stwcx.** is to emulate a compare and swap primitive like that provided by the IBM System/370 compare and swap instruction, which checks only that the old and current values of the word being tested are equal, with the result that programs that use such a compare and swap to control a shared resource can err if the word has been modified and the old value is subsequently restored. The use of **lwarx** and **stwcx.** improves on such a compare and swap, because the reservation reliably binds **lwarx** and **stwcx.** together. The reservation is always lost if the word is modified by another processor or mechanism between the **lwarx** and **stwcx.**, so the **stwcx.** never succeeds unless the word has not been stored into (by another processor or mechanism) since the **lwarx**.

A processor has at most one reservation at any time. The architecture states that a reservation is established by executing a **lwarx** and is lost (or may be lost, in the case of the fourth and fifth bullets) if any of the following occurs:

- The processor holding the reservation executes another **lwarx**; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes any **stwcx.**, regardless of whether the specified address matches that of the **lwarx**.
- Another processor executes a store or **dcbz**, **dcbzl** <DEO>, **dcbzep** <E.PD>, or **dcbzlep** <E.PD,DEO> to the same reservation granule.
- Another processor executes a **dcbtst**, **dcbst**, or **dcbf** to the same reservation granule; whether the reservation is lost is undefined.
- Another processor executes a **dcba** or **dcbal** <DEO> to the reservation granule. The reservation is lost if the instruction causes the target block to be newly established in the data cache or to be modified; otherwise, whether the reservation is lost is undefined.
- Some other mechanism modifies a location in the same reservation granule.
- An asynchronous interrupt may cause a reservation to be lost.
- Other implementation-specific conditions may also cause the reservation to be cleared, See the core reference manual.

Interrupts are not guaranteed to clear reservations. (However, system software invoked by interrupts may clear reservations.)

In general, programming conventions must ensure that **lwarx** and **stwcx.** specify addresses that match; a **stwcx.** should be paired with a specific **lwarx** to the same location. Situations in which a **stwcx.** may erroneously be issued after some **lwarx** other than that with which it is intended to be paired must be scrupulously avoided. For example, there must not be a context switch in which the processor holds a reservation in behalf of the old context, and the new context resumes after a **lwarx** and before the paired **stwcx.**. The **stwcx.** in the new context might succeed, which is not what was intended by the programmer.

Such a situation must be prevented by issuing a **stwcx.** to a dummy writable word-aligned location as part of the context switch, thereby clearing any reservation established by the old context. Executing **stwcx.** to a word-aligned location is enough to clear the reservation, regardless of whether it was set by **lwarx**.

4.6.1.16.2 Forward Progress

Forward progress in loops that use **lwarx** and **stwcx.** is achieved by a cooperative effort among hardware, operating system software, and application software.

One of the following is guaranteed when a processor executes a **lwarx** to obtain a reservation for location X and then a **stwcx.** to store a value to location X:

1. The **stwcx.** succeeds and the value is written to location X.
2. The **stwcx.** fails because some other processor or mechanism modified location X.
3. The **stwcx.** fails because the processor’s reservation was lost for some other reason.

In cases 1 and 2, the system as a whole makes progress in the sense that some processor successfully modifies location X. Case 3 covers reservation loss required for correct operation of the rest of the system. This includes cancellation caused by some other processor writing elsewhere in the reservation granule for X, as well as cancellation caused by the operating system in managing certain limited resources such as real memory or context switches. It may also include implementation-dependent causes of reservation loss.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of reservation loss in case 3. Although the architecture alone cannot provide such a guarantee, the conditions in cases 1 and 2 are necessary for a guarantee. An implementation and operating system can build on them to provide such a guarantee.

NOTE

The architecture does not guarantee fairness. In competing for a reservation, two processors can indefinitely lock out a third.

4.6.1.16.3 Reservation Loss Due to Granularity

Lock words should be allocated such that contention for the locks and updates to nearby data structures do not cause excessive reservation losses due to false indications of sharing that can occur due to the reservation granularity.

A processor holding a reservation on any word in a reservation granule loses its reservation if some other processor stores anywhere in that granule. Such problems can be avoided only by ensuring that few such stores occur. This can most easily be accomplished by allocating an entire granule for a lock and wasting all but one word.

Reservation granularity may vary for each implementation. There are no architectural restrictions bounding the granularity implementations must support, so reasonably portable code must dynamically allocate aligned and padded memory for locks to guarantee absence of granularity-induced reservation loss.

4.6.1.17 Cache Management Instructions

This section briefly describes the user-level cache management instructions. See Section 4.6.2.2, “Supervisor-Level Memory Control Instructions,” for supervisor-level cache management instructions.

Cache management instructions help software more explicitly manage on-chip caches. They allow software to perform functions such as touch, flush, invalidate, zero, allocate, and lock to addressed cache lines (blocks). Because cache blocks have an implementation-dependent size, such instructions should be used with care or used only in a library where the library executes such instructions with knowledge of the cache block size.

As with other memory-related instructions, the effects of cache management instructions on memory are weakly ordered. If the programmer must ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, an **sync** must be placed after those instructions.

It is permissible for an implementation to treat any or all of the cache touch instructions (**icbt**, **dcbt**, or **dcbtst**) as no-ops, even if a cache is implemented.

4.6.1.17.1 Cache Target Identifiers

Some cache management instructions, such as cache locking instructions, provide an immediate operand CT (or TH for **dcbt** and **dcbtst** instructions) that explicitly designate a particular cache in a hierarchy of caches. Such values are cache target identifiers or “CT values.” EIS defines CT values to be within the range of 0–7. Other CT values are boundedly undefined and should not be used.

CT values are said to be internal if they specify a cache that is specific to the processor core, or the processor core complex and the processor core that issues the instruction defines which cache the operation applies to. CT values are said to be external if they specify a cache explicitly regardless of which processor core the request is issued from. By definition, external CT values must be mapped one to one to specific caches within a coherence domain.

The following CT values are defined:

- CT = 0 indicates the L1 (or primary) cache or the processor that executes the instruction. When the primary cache is a separated instruction and data cache (harvard architecture), the type of instruction determines whether the operation is performed to the instruction cache or the data cache. CT = 0 is an internal CT value.
- CT = 1 indicates the I/O or platform cache, if one is implemented on the integrated device. Note that some documentation refers to this cache as a frontside L2 cache. CT = 1 is an external CT

value.

Additional external CT values may be defined by the integrated device.

- CT = 2 indicates the L2 cache. The L2 cache may be private to a processor or may be shared among several processors in a processor complex. CT = 2 is an internal CT value.
- The CT values of 4 and 6 are reserved for internal CT values. Executing any cache management instruction with a CT value of 4 or 6 is boundedly undefined.
- The CT values 3, 5, and 7 are reserved for implementation defined external CT values.

Any cache management instruction that contains a valid internal CT value, but no such cache is implemented, is treated as a no-op. Any cache management instruction which contains an external CT value is always treated as valid. If no such external CT value is implemented on the integrated device, the results are implementation specific.

4.6.1.17.2 User-Level Cache Management Instructions

This table summarizes the cache management instructions.

Table 4-42. User-Level Cache Instructions

Name	Mnemonic	Syntax	Description
Data Cache Block Allocate	**dcba**	**r**A,**r**B	**dcba** is a performance hint that allocates a block specified by EA in the data cache without first reading the contents from the memory subsystem. The result leaves the contents of the block as undefined. It is expected that the processor executing the **dcba** will subsequently perform stores to write the block to define its contents. **dcba** can be used when the entire block is to be written and the current contents of the block are no longer required. Many implementations will treat **dcba** as **dcbz** and zero the block when it is allocated. L1CSR0[DCBZ32] = 1, **dcba** operates on 32 byte aligned granules (32-byte operation). This may perform slower on some implementations. This instruction is treated as a store with respect to translation, protection, and debug address comparisons. The **dcba** is treated as a no-op if the block cannot be allocated in the cache, if the page is marked write-through or cache-inhibited, or if a TLB protection violation occurs. Other implementation specific conditions can cause the **dcba** to be treated as a no-op. See the core reference manual.
Data Cache Block Allocate by Line <DEO>	**dcbal**	**r**A,**r**B	**dcbal** behaves the same as **dcba** except it always operates on all bytes in the cache line regardless of the setting of L1CSR0[DCBZ32] .

Table 4-42. User-Level Cache Instructions (continued)

Name	Mnemonic	Syntax	Description
Data Cache Block Flush	**dcbf**	**r**A,**r**B	**dcbf** flushes the block specified by EA from the data cache. If the block exists and is modified in the cache, the block is first written back to memory. The cache block is then invalidated. If the block does not exist in the cache, no action in the cache is taken. Only local caches (primary data cache and L2 cache) are affected when the address is not in memory that is Memory Coherence Required, otherwise the operation applies to all caches in the coherence domain. This instruction is treated as a load with respect to translation, protection, and debug address comparisons.
Data Cache Block Zero	**dcbz**	**r**A,**r**B	**dcbz** allocates the block specified by EA in the cache (if it does not already exist) and sets all bytes in the block to zero.If the block is allocated in the cache, the block is allocated without first reading its contents from the memory subsystem. If the address specified by EA is caching inhibited or write-through, **dcbz** will either take an alignment interrupt or set all the bytes specified by the cache block to zero. If the block cannot be established in the cache (because the cache may be locked or disabled), it is implementation dependent whether an alignment interrupt occurs or the **dcbz** sets all the bytes in the cache line to zero. L1CSR0[DCBZ32] = 1, **dcbz** operates on 32 byte aligned granules (32-byte operation). This may perform slower on some implementations. This instruction is treated as a store with respect to memory barriers, synchronization, translation and protection, and debug address comparisons.
Data Cache Block Zero by Line <DEO>	**dcbzl**	**r**A,**r**B	**dcbzl** behaves the same as **dcbz** except it always operates on all bytes in the cache line regardless of the setting of L1CSR0[DCBZ32] .
Data Cache Block Store	**dcbst**	**r**A,**r**B	**dcbst** flushes the block specified by EA from the data cache. If the block exists and is modified in the cache, the block is written back to memory. It is implementation dependent whether the block remains in the cache after being flushed. If it remains in the cache, it is not in a modified state. If the block does not exist in the cache, no action in the cache is taken. Only local caches (primary data cache and L2 cache) are affected when the address is not in memory that is memory coherence required, otherwise the operation applies to all caches in the coherence domain. This instruction is treated as a load with respect to translation, protection, and debug address comparisons.

Table 4-42. User-Level Cache Instructions (continued)

Name	Mnemonic	Syntax	Description
Data Cache Block Touch [1]	**dcbt**	TH,**r**A,**r**B	**dcbt** is a performance hint that causes the block specified by EA to be read from memory and established in the cache specified by TH. As a hint, the processor may treat the operation as a no-op. The **dcbt** is treated as a no-op if translation would cause any exception (including DTLB and DSI exceptions), the line cannot be allocated in the cache, or if the address is write-through or cache-inhibited. Other implementation specific conditions can cause the **dcbt** to be treated as a no-op. See the core reference manual. This instruction is treated as a load with respect to translation, protection, and debug address comparisons. It is implementation dependent whether a **dcbt** that is treated as a no-op will cause debug events.
Data Cache Block Touch for Store [1]	**dcbtst**	TH,**r**A,**r**B	**dcbtst** behaves the same as **dcbt**, except the hint assumes that software will soon write to the block. Some implementations may also treat this instruction as a store with respect to translation and protection.
Instruction Cache Block Invalidate	**icbi**	**r**A,**r**B	**icbi** invalidates the block specified by EA from the instruction caches and L2 caches which implement harvard architecture semantics. **icbi** should always be followed by a **sync** and an **isync** to make sure its effects are seen by instruction fetches and instruction execution following the **icbi** itself. This instruction is treated as a load with respect to translation, protection, and debug address comparisons.
Instruction Cache Block Touch	**icbt**	CT,**r**A,**r**B	**icbt** is a performance hint that causes the block specified by EA to be read from memory and established in the instruction cache specified by CT. As a hint, the processor may treat the operation as a no-op. The **icbt** is treated as a no-op if translation would cause any exception (including DTLB and DSI exceptions), the line cannot be allocated in the cache, or if the address is write-through or cache-inhibited. Other implementation specific conditions can cause the **icbt** to be treated as a no-op. See the core reference manual. It is implementation dependent whether **icbt** requires read or execute permissions to perform the hint. This instruction is treated as a load with respect to translation, protection, and debug address comparisons. It is implementation dependent whether an **icbt** that is treated as a no-op will cause debug events.

[1] A program that uses **dcbt** and **dcbtst** improperly is less efficient. To improve performance, HID0[NOPTI] can be set, which causes **dcbt** and **dcbtst** to be no-oped at the cache. The default state of this bit is zero, which enables the use of these instructions.

4.6.1.17.3 Cache Locking Instructions <E.CL>

Cache locking instructions provide the capability to lock cache blocks into specific caches. Cache locking instructions require supervisor privilege to execute unless MSR[UCLE] is set. A complete description of cache locking behaviors is described in Section 6.3.1, “Cache Line Locking <E.CL>.”

This table lists the cache locking instructions.

Table 4-43. Cache Locking Instructions

Name	Mnemonic	Syntax	Description
Data Cache Block Lock Clear	**dcblc**	CT,rA,rB	The line in the specified cache is unlocked, making it eligible for replacement. This instruction is treated as a load with respect to translation, protection, and debug address comparisons.
Data Cache Block Lock Query	**dcblq.**	CT,rA,rB	The line in the specified cache is queried. The status of the lock is placed into CR0[EQ]. This instruction is treated as a load with respect to translation, protection, and debug address comparisons.
Data Cache Block Touch and Lock Set	**dcbtls**	CT,rA,rB	The line is loaded and locked into the specified cache. This instruction is treated as a load with respect to translation, protection, and debug address comparisons.
Data Cache Block Touch for Store and Lock Set	**dcbtstls**	CT,rA,rB	The line is loaded and locked into the specified cache. The line is marked as modified. It is implementation dependent whether this instruction is treated as a load or a store with respect to translation, protection, and debug address comparisons.
Instruction Cache Block Lock Clear	**icblc**	CT,rA,rB	The line in the specified cache is unlocked, making it eligible for replacement. This instruction is treated as a load with respect to translation, protection, and debug address comparisons.
Instruction Cache Block Lock Query	**icblq.**	CT,rA,rB	The line in the specified cache is queried. The status of the lock is placed into CR0[EQ]. This instruction is treated as a load with respect to translation, protection, and debug address comparisons.
Instruction Cache Block Touch and Lock Set	**icbtls**	CT,rA,rB	The line is loaded and locked into the specified cache. This instruction is treated as a load with respect to translation, protection, and debug address comparisons.

4.6.2 Hypervisor- and Supervisor-Level Instructions

This section describes the hypervisor- and supervisor-level instructions provided by the architecture. Supervisor instructions are used to perform TLB operations, access privileged registers, and handle interrupts.

Instructions described here have an associated privilege and actions as described in this table.

Table 4-44. Instruction Execution Based on Privilege Level

Privilege Level of Instruction	User Mode (MSR[GS,PR]=0bx1)	Guest Supervisor Mode (MSR[GS,PR]=0b10)	Hypervisor Mode (MSR[GS,PR]=0b00)
User	execute normally	execute normally	execute normally
Guest Supervisor	privileged instruction exception	execute normally	execute normally
Hypervisor	privileged instruction exception	embedded hypervisor privilege exception	execute normally

4.6.2.1 System Linkage and MSR Access Instructions

Table 4-45 lists system linkage instructions defined by the architecture. The user-level **sc** (LEV = 0) instruction lets a user program call on the system to perform a service and causes the processor to take a system call interrupt. The **sc** (LEV = 1) <E.HV> instruction is also used for the supervisor to invoke the hypervisor to perform a service and causes the processor to take an embedded hypervisor system call interrupt. The supervisor-level **rfi** and **rfgi** <E.HV> instructions are used for returning from an interrupt handler. The **rfci** instruction is used for critical interrupts; **rfdi** is used for debug interrupts;<E.ED> **rfmci** is used for machine check interrupts.

Return from interrupt instructions are context-synchronizing.

<Embedded.Hypervisor>:
In guest supervisor state, **rfi** (**rfgi**) cannot modify any MSR bits protected by MSRP or MSR[GS]. Guest supervisor software should use **rfi**, **rfci**, **rfdi** <E.ED>, and **rfmci** when returning from their associated interrupts. When a guest operating system executes **rfi**, the processor maps the instruction to **rfgi** assuring that the appropriate guest save/restore registers are used for the return. For **rfci**, **rfdi** <E.ED>, and **rfmci**, the hypervisor should emulate these instructions as it will emulate the taking of these interrupts in guest supervisor state.

Privileges are as follows:

- **sc** is user privileged.
- **rfi** (**rfgi** <E.HV>), **mfmsr**, **mtmsr**, **wrtee**, **wrteei** are guest–supervisor privileged.
- **rfci**, **rfdi** <E.ED>, **rfmci** are hypervisor privileged.

Table 4-45. System Linkage Instructions—Supervisor-Level

Name	Mnemonic	Syntax	Description
Return from Interrupt	**rfi**	—	**rfi** is used to return from a base level interrupt. SRR0/SRR1 are used to provide the return address and MSR value respectively. **rfgi** is executed in place of **rfi** when executed in guest supervisor state.
Return from Guest Interrupt <E.HV>	**rfgi**	—	**rfgi** is used to return from a base level interrupt. GSRR0/GSRR1 are used to provide the return address and MSR value respectively.
Return from Critical Interrupt	**rfci**	—	**rfci** is used to return from a critical level interrupt. CSRR0/SCRR1 are used to provide the return address and MSR value respectively.
Return from Debug Interrupt <E.ED>	**rfdi**	—	**rfdi** is used to return from a debug level interrupt. SDRR0/DSRR1 are used to provide the return address and MSR value respectively.
Return from Machine Check Interrupt	**rfmci**	—	**rfmci** is used to return from a machine check level interrupt. MCSRR0/MCSRR1 are used to provide the return address and MSR value respectively.
System Call	**sc**	LEV	sc invokes a system service depending on the value of LEV: LEV = 0 - system call interrupt, LEV = 1 - embedded hypervisor system call interrupt <E.HV>

This table lists instructions for accessing the MSR.

Table 4-46. Move to/from Machine State Register Instructions

Name	Mnemonic	Syntax	Description
Move from Machine State Register	**mfmsr**	**r**D	**mfmsr** copies the contents of the MSR to **r**D.
Move to Machine State Register	**mtmsr**	**r**S	**mtmsr** copies the contents of **r**S to the MSR. In guest supervisor state **mtmsr** cannot alter MSR[GS] or any bits protected by MSRP. <E.HV>
Write MSR External Enable	**wrtee**	**r**S	Bit 48 of the contents of **r**S is placed into MSR[EE]. Other MSR bits are unaffected.
Write MSR External Enable Immediate	**wrteei**	E	The value of E is placed into MSR[EE]. Other MSR bits are unaffected.

4.6.2.2 Supervisor-Level Memory Control Instructions

Memory control instructions include the following:

- Cache management instructions (supervisor-level and user-level)
- Translation lookaside buffer management instructions

This section describes supervisor-level memory control instructions. Section 4.6.1.17, “Cache Management Instructions,” describes user-level memory control instructions.

4.6.2.2.1 Supervisor-Level Cache Management Instructions

This table lists the supervisor-level cache management instructions. **dcbstep**, **dcbtep**, **dcbtstep**, **dcbfep**, **icbiep**, **dcbzep**, and **dcbzlep** are cache management instructions as well. Because they use external PID semantics for translation, they are listed in Section 4.6.2.3, “External PID Instructions <E.PD>.”

Table 4-47. Supervisor-Level Cache Management Instruction

Name	Mnemonic	Syntax	Description
Data Cache Block Invalidate	**dcbi**	**r**A,**r**B	**dcbi** invalidates the block specified by EA from the data cache. If the block exists, is modified in the cache, and is marked as Memory Coherence Required, it is implementation dependent whether the block is first written back to memory. The cache block is then invalidated. If the block does not exist in the cache, no action in the cache is taken. Only local caches (primary data cache and L2 cache) are affected when the address is not in memory that is Memory Coherence Required, otherwise the operation applies to all caches in the coherence domain. If the instruction causes modified data to be invalidated before writing the block back to memory, then the instruction is treated as a store with respect to memory barriers, synchronization, translation and protection, and debug address comparisons. Otherwise, the instruction is treated as a load with respect to memory barriers, synchronization, translation and protection, and debug address comparisons **NOTE: Software Considerations** Invalidation without flushing can cause modified data to be lost. Software should carefully consider the implications of using **dcbi**.

See Section 4.6.1.17.2, “User-Level Cache Management Instructions,” for cache instructions that provide user-level programs the ability to manage the on-chip caches.

4.6.2.2.2 TLB Management Instructions

The address translation mechanism is defined in terms of Translation Lookaside Buffer (TLB) entries used to translate the effective-to-physical address mapping for a particular access. See Section 6.5, “MMU Architecture,” for more information about TLB operations.

This table summarizes the operation of the TLB instructions.

Table 4-48. TLB Management Instructions

Name	Mnemonic	Syntax	Description
TLB Invalidate Local <E.HV>	**tlbilx**	T, **r**A, **r**B	**tlbilx** performs invalidations of TLB entries on the processor that executes the **tlbilx** instruction. Any entry that has the IPROT attribute set is not invalidated. **tlbilx** can be used to invalidate all entries corresponding to a LPID value, all entries corresponding to a PID value, or a single entry. See Section 6.5, “MMU Architecture.” **tlbilx** is guest supervisor privileged, however it will cause an embedded hypervisor privilege exception if EPCR[DGTMI] is set. **NOTE: Software Considerations** **tlbilx** requires the same local-processor synchronization as **tlbivax**, but not the cross-processor synchronization (that is, it does not require **tlbsync**).
TLB Invalidate Virtual Address Indexed	**tlbivax**	**r**A, **r**B	**tlbivax** performs invalidations of TLB entries. **tlbivax** invalidates any TLB entry on any processor in the coherence domain that corresponds to the virtual address calculated by this instruction. Any entry that has the IPROT attribute set is not invalidated. Thus an invalidate operation is broadcast throughout the coherence domain of the processor executing **tlbivax**. See Section 6.5, “MMU Architecture.” **tlbivax** is hypervisor privileged. **Note:** : The preferred form of **tlbivax** contains the entire EA in **r**B and zero in **r**A. Some implementations may take an illegal instruction exception if **r**A is nonzero. **Note:** : The preferred method of performing invalidations on processors that implement category E.HV is to execute **tlbilx** on all processors that need to perform the invalidation.
TLB Read Entry	**tlbre**	—	**tlbre** causes the contents of a single TLB entry to be extracted from the MMU and be placed in the corresponding fields of the MAS registers. The entry extracted is specified by the TLBSEL, ESEL, and EPN fields of MAS0 and MAS2. The contents extracted from the MMU are placed in MAS0–MAS3, MAS7, and MAS8. See Section 6.5.3.3, “Reading and Writing TLB Entries.” **tlbre** is hypervisor privileged.
TLB Search Indexed	**tlbsx**	**r**A, **r**B	**tlbsx** searches the MMU for a particular entry based on the computed EA and the search values in MAS5 and MAS6 .If a match is found, MAS1[V] is set and the found entry is read into the MAS0–MAS3, MAS7, and MAS8. If the entry is not found MAS1[V] is set to 0.See Section 6.5.3.3, “Reading and Writing TLB Entries.” **tlbsx** is hypervisor privileged. **Note:** : **r**A=0 is a preferred form for **tlbsx** and that some Freescale implementations take an illegal instruction exception if **r**A != 0.

Table 4-48. TLB Management Instructions (continued)

Name	Mnemonic	Syntax	Description
TLB Synchronize	**tlbsync**	—	**tlbsync** provides ordering for the effects of all **tlbivax** instructions executed by the processor executing **tlbsync**, with respect to the memory barrier created when that processor executes a subsequent **sync**. Executing **tlbsync** ensures that all of the following occurs: • All TLB invalidations caused by **tlbivax** instructions before the **tlbsync** complete on any other processor before any storage accesses associated with data accesses caused by instructions following the **sync** are performed with respect to that processor. • All storage accesses by other processors for which the address was translated using the translations being invalidated, will have been performed with respect to the processor executing the **sync**, to the extent required by the associated memory coherence required attributes, before the **mbar** or **sync** instruction's memory barrier is created. See Section 6.5, "MMU Architecture." **tlbsync** is hypervisor privileged. Note that only one **tlbsync** can be in process at any given time on all processors of a coherence domain. The hypervisor or operating system should ensure this by doing the appropriate mutual exclusion.
TLB Write Entry	**tlbwe**	—	**tlbwe** causes the contents of certain fields of MAS0–MAS3, MAS7, and MAS8 to be written into a single TLB entry in the MMU. The entry written is specified by the TLBSEL, ESEL, and EPN fields of MAS0 and MAS2. See Section 6.5.3.3, "Reading and Writing TLB Entries." **tlbwe** is always hypervisor privileged.

4.6.2.3 External PID Instructions <E.PD>

External PID load and store instructions are used by the operating system and hypervisor to perform load, store, and cache management instructions to a separate address space while still fetching and executing in the normal supervisor or hypervisor context. The operating system or hypervisor selects the address space to target by altering the contents of the EPLC and EPSC registers for loads and stores respectively. When the effective address specified by the external PID load or store instruction is translated, the translation mechanism uses ELPID, EPID, EAS, EPR, and EGS fields from the EPLC or EPSC register instead of LPIDR, PID, MSR[DS], MSR[PR], and MSR[GS] values. Such instructions are useful for an operating system to manipulate a processes virtual memory using the context and credentials of the process.

External PID instructions have analogous non-external PID load, store, or cache management instructions. The instruction behavior is the same (except how the address space is translated and how permissions are applied) for the external PID instruction and its analogous non-external PID instruction.

All external PID instructions are guest supervisor privileged.

This table lists external PID load and store instructions.

Table 4-49. External PID Load Store Instructions

Instruction	Mnemonic	Syntax	Non External PID Analogous Instruction
Data Cache Block Flush by External PID Indexed	**dcbfep**	**r**A,**r**B	**dcbf**
Data Cache Block Store by External PID Indexed	**dcbstep**	**r**A,**r**B	**dcbst**
Data Cache Block Touch by External PID Indexed	**dcbtep**	TH,**r**A,**r**B	**dcbt**
Data Cache Block Touch for Store by External PID Indexed	**dcbtstep**	TH,**r**A,**r**B	**dcbtst**
Data Cache Block Zero by External PID Indexed	**dcbzep**	**r**A,**r**B	**dcbz**
Data Cache Block Zero Line by External PID Indexed <DEO>	**dcbzlep**	**r**A,**r**B	**dcbzl**
Instruction Cache Block Invalidate by External PID Indexed	**icbiep**	**r**A,**r**B	**icbi**
Load Byte by External PID Indexed	**lbepx**	**r**D,**r**A,**r**B	**lbzx**
Load Doubleword by External PID Indexed <64>	**ldepx**	**r**D,**r**A,**r**B	**ldx**
Load Floating-Point Double Word by External PID Indexed <FP>	**lfdepx**	**fr**D,**r**A,**r**B	**lfdx**
Load Half Word by External PID Indexed	**lhepx**	**r**D,**r**A,**r**B	**lhzx**
Load Word by External PID Indexed	**lwepx**	**r**D,**r**A,**r**B	**lwzx**
Store Byte by External PID Indexed	**stbepx**	**r**S,**r**A,**r**B	**stbx**
Store Doubleword by External PID Indexed <64>	**stdepx**	**r**S,**r**A,**r**B	**stdx**
Store Floating-Point Double Word by External PID Indexed <FP>	**stfdepx**	**fr**S,**r**A,**r**B	**stfdx**
Store Half Word by External PID Indexed	**sthepx**	**r**S,**r**A,**r**B	**sthx**
Store Word by External PID Indexed	**stwepx**	**r**S,**r**A,**r**B	**stwx**
Vector Load Doubleword into Doubleword by External PID Indexed <SP>	**evlddepx**	**r**D,**r**A,**r**B	**evlddx**
Vector Store Doubleword into Doubleword by External PID Indexed <SP>	**evstddepx**	**r**S,**r**A,**r**B	**evstddx**
Load Vector by External PID Indexed <V>	**lvepx**	**v**D,**r**A,**r**B	**lvx**
Load Vector by External PID Indexed LRU <V>	**lvepxl**	**v**D,**r**A,**r**B	**lvxl**
Store Vector by External PID Indexed <V>	**stvepx**	**v**S,**r**A,**r**B	**stvx**
Store Vector by External PID Indexed LRU <V>	**stvepxl**	**v**S,**r**A,**r**B	**stvxl**

4.6.2.4 Hypervisor-Level Messaging Instructions <E.PC>

Messaging instructions provide facilities for processors within a coherence domain to send messages to other devices in the coherence domain. The facility provides a mechanism for sending interrupts to other processors that are not dependent on the interrupt controller and allow message filtering by the processors that receive the message.

Messaging initiated by processors to processors is topology independent, and when category E.HV is implemented is also fully partitioned.

Some message types, particularly the Guest Processor Doorbell types, are used by the hypervisor to reflect (or synthesize) interrupts to a guest operating system when the operating system has set the appropriate asynchronous interrupt enables.

Messaging instructions are also useful for sending messages to a device to provide specialized services such as secure boot operations controlled by a security device. EIS does not define any such messages at this time. EIS does define processor-to-processor messaging and what actions processors take on the receipt of a message. The actions taken by devices other than processors is left to the architectural definition of that particular device.

4.6.2.4.1 Sending and Receiving Messages

Processors initiate a message by executing the **msgsnd** instruction and specifying a message type and message payload in a general purpose register. Sending a message causes the message to be sent to all the processors, including the sending processor, in the coherence domain in a reliable manner. The message is broadcast on the interconnect mechanism that connects all devices in the coherence domain and has a unique transaction type that cannot be generated using any other processor operations. The uniqueness of the transaction type insures that processors can only generate a message transaction using the defined instructions that send such messages.

Each device receives all messages that are sent. The actions that a device takes are dependent on the message type and payload. There are no restrictions on what messages a processor can send.

When a device or processor receives a message, the processor examines the message type and payload to determine whether the device or processor should accept the message. This is called message “filtering.” If, after examining the payload, the device or processor decides that the message should be processed (that is, has met all the appropriate criteria specified in the message type and payload), the device or processor *accepts* the message and processes it accordingly.

Processors that implement the category E.PC filter and accept messages are defined in Table 4-50. Processors that also implement category E.HV and filter and accept messages are defined in Table 4-51. Processors ignore (that is, filter and do not accept) any messages with other message types.

Table 4-50. Processor Message Types

Message Type (bits 32:36)	Message Type Name	Message Type Description	Interrupt Enabling Condition
0	DBELL	Doorbell. A processor doorbell interrupt (see Section 7.8.21, “Processor Doorbell Interrupt <E.PC>”) is generated or pended on the processor when the processor has filtered the message based on the payload and has determined that it should accept the message. A processor doorbell interrupt is gated by MSR[EE].	MSR[EE]
1	DBELL_CRIT	Doorbell Critical. A processor doorbell critical interrupt (see Section 7.8.22, “Processor Doorbell Critical Interrupt <E.PC>”) is generated or pended on the processor when the processor has filtered the message based on the payload and has determined that it should accept the message. A processor doorbell critical interrupt is gated by MSR[CE].	MSR[CE]

Table 4-51. Processor Message Types <E.HV>

Message Type (bits 32:36)	Message Type Name	Message Type Description	Interrupt Enabling Condition
2	G_DBELL	Guest Doorbell. A guest processor doorbell interrupt (see Section 7.8.23, "Guest Processor Doorbell Interrupt <E.HV>") is generated or pended on the processor when the processor has filtered the message based on the payload and has determined that it should accept the message. A guest processor doorbell interrupt is gated by MSR[EE] & MSR[GS]. Note: *Guest processor doorbell interrupts are directed to the hypervisor and will only interrupt when the guest is in execution. This message is used by hypervisor software to reflect or emulate an MSR[EE] based asynchronous interrupt to the guest operating system*.	MSR[EE] & MSR[GS]
3	G_DBELL_CRIT	Guest Doorbell Critical. A guest processor doorbell critical interrupt (see Section 7.8.24, "Guest Processor Doorbell Critical Interrupt <E.HV>") is generated or pended on the processor when the processor has filtered the message based on the payload and has determined that it should accept the message. A guest processor doorbell critical interrupt is gated by MSR[CE] & MSR[GS]. Note: Guest processor doorbell critical interrupts are directed to the hypervisor and will only interrupt when the guest is in execution. This message is used by hypervisor software to reflect or emulate an MSR[CE] based asynchronous interrupt to the guest operating system.	MSR[CE] & MSR[GS]
4	G_DBELL_MC	Guest Doorbell Machine Check. A guest processor doorbell machine check interrupt (see Section 7.8.25, "Guest Processor Doorbell Machine Check Interrupt <E.HV>") is generated or pended on the processor when the processor has filtered the message based on the payload and has determined that it should accept the message. A guest processor doorbell machine check interrupt is gated by MSR[ME] & MSR[GS]. Note: Guest processor doorbell machine check interrupts are directed to the hypervisor and will only interrupt when the guest is in execution. This message is used by hypervisor software to reflect or emulate an MSR[ME] based asynchronous interrupt to the guest operating system.	MSR[ME] & MSR[GS]

If the message is accepted, the message may be pended if the interrupt gating conditions are not met. For example, a DBELL_CRIT message is received on a processor and is accepted; however, MSR[CE] = 0 at the time the message is accepted. The interrupt is pended until MSR[CE] changes to 1.

Messages of the same type are not cumulative; that is, if a message of type *n* is accepted, and the interrupt is pended, further messages of type *n* accepted by the processor are ignored until the associated pending condition is cleared by taking the interrupt or executing a **msgclr** instruction on the accepting processor. In general, software will be required to use memory to perform higher level messaging, using **msgsnd** to notify other processors that higher level messages are waiting for the accepting processor to process. For this reason, a **sync 0** will order any previous stores and any subsequent **msgsnd** operations. This ensures that the stores to write higher level message information are performed before the message is sent. The timing relationship between when a message is sent and when it is received is not defined.

4.6.2.5 Performance Monitor Instructions

The performance monitor instructions provide read-write access to performance monitor resources. Instructions are listed in this table.

Table 4-52. Supervisor Performance Monitor Instructions

Name	Mnemonic	Syntax
Move from Performance Monitor Register	**mfpmr**	**r**D,PMRN
Move to Performance Monitor Register	**mtpmr**	PMRN,**r**S

The PMRs listed in Section 3.15, “Performance Monitor Registers (PMRs) <E.PM>” are accessed with **mfpmr** and **mtpmr**.

The hypervisor can restrict guest supervisor access to PMRs by setting MSRP[PMMP]. <E.HV>

4.6.2.6 Supervisor Level Device Control Register Instructions <E.DC>

Device control register instructions are used to read and write Device Control Registers (DCRs). The definition of DCRs is implementation dependent. Device control instructions are hypervisor privileged.

This table lists the device control instructions.

Table 4-53. Device Control Register Instructions

Instruction	Mnemonic	Syntax
Move From Device Control Register	**mfdcr**	**r**D,DCRN
Move To Device Control Register	**mtdcr**	DCRN,**r**S

4.6.3 Recommended Simplified Mnemonics

To simplify assembly language programming, simplified mnemonics and symbols are provided for some frequently used instructions. These are listed in Section C.10, “Recommended Simplified Mnemonics.” Programs written to be portable across the various assemblers for the architecture should not assume the existence of mnemonics not described in this document.

4.7 Instruction Listing

This table lists the instructions defined by EIS except for instructions defined by category VLE.

Table 4-54. Instruction Set

Mnemonic	Syntax	Category	Comments
add	**r**D,**r**A,**r**B	Base	—
add.	**r**D,**r**A,**r**B	Base	—
addc	**r**D,**r**A,**r**B	Base	—
addc.	**r**D,**r**A,**r**B	Base	—
addco	**r**D,**r**A,**r**B	Base	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
addco.	rD,rA,rB	Base	—
adde	rD,rA,rB	Base	—
adde.	rD,rA,rB	Base	—
addeo	rD,rA,rB	Base	—
addeo.	rD,rA,rB	Base	—
addi	rD,rA,SIMM	Base	—
addic	rD,rA,SIMM	Base	—
addic.	rD,rA,SIMM	Base	—
addis	rD,rA,SIMM	Base	—
addme	rD,rA	Base	—
addme.	rD,rA	Base	—
addmeo	rD,rA	Base	—
addmeo.	rD,rA	Base	—
addo	rD,rA,rB	Base	—
addo.	rD,rA,rB	Base	—
addze	rD,rA	Base	—
addze.	rD,rA	Base	—
addzeo	rD,rA	Base	—
addzeo.	rD,rA	Base	—
and	rA,rS,rB	Base	—
and.	rA,rS,rB	Base	—
andc	rA,rS,rB	Base	—
andc.	rA,rS,rB	Base	—
andi.	rA,rS,UIMM	Base	—
andis.	rA,rS,UIMM	Base	—
b	LI	Base	—
ba	LI	Base	—
bc	BO,BI,BD	Base	—
bca	BO,BI,BD	Base	—
bcctr	BO,BI	Base	—
bcctrl	BO,BI	Base	—
bcl	BO,BI,BD	Base	—
bcla	BO,BI,BD	Base	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
bclr	BO,BI	Base	—
bclrl	BO,BI	Base	—
bl	LI	Base	—
bla	LI	Base	—
bpermd	**r**A,**r**S,**r**B	64	—
brinc	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
cmp	**crD**,L,**r**A,**r**B	Base	—
cmpb	**r**A,**r**S,**r**B	Base	—
cmpi	**crD**,L,**r**A,SIMM	Base	—
cmpl	**crD**,**r**A,**r**B	Base	—
cmpli	**crD**,L,**r**A,UIMM	Base	—
cntlzd	**r**A,**r**S	64	—
cntlzd.	**r**A,**r**S	64	—
cntlzw	**r**A,**r**S	Base	—
cntlzw.	**r**A,**r**S	Base	—
crand	**crb**D,**crb**A,**crb**B	Base	—
crandc	**crb**D,**crb**A,**crb**B	Base	—
creqv	**crb**D,**crb**A,**crb**B	Base	—
crnand	**crb**D,**crb**A,**crb**B	Base	—
crnor	**crb**D,**crb**A,**crb**B	Base	—
cror	**crb**D,**crb**A,**crb**B	Base	—
crorc	**crb**D,**crb**A,**crb**B	Base	—
crxor	**crb**D,**crb**A,**crb**B	Base	—
dcba	**r**A,**r**B	Base	—
dcbal	**r**A,**r**B	DEO	—
dcbf	**r**A,**r**B	Base	—
dcbfep	**r**A,**r**B	E.PD	—
dcbi	**r**A,**r**B	Embedded	—
dcblc	CT,**r**A,**r**B	E.CL	—
dcblq.	CT,**r**A,**r**B	E.CL	—
dcbst	**r**A,**r**B	Base	—
dcbstep	**r**A,**r**B	E.PD	—
dcbt	TH,**r**A,**r**B	Base	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
dcbtep	TH,**r**A,**r**B	E.PD	—
dcbtls	CT,**r**A,**r**B	E.CL	—
dcbtst	CT,**r**A,**r**B	Base	—
dcbtstep	TH,**r**A,**r**B	E.PD	—
dcbtstls	CT,**r**A,**r**B	E.CL	—
dcbz	**r**A,**r**B	Base	—
dcbzep	**r**A,**r**B	E.PD	—
dcbzl	**r**A,**r**B	DEO	—
dcbzlep	**r**A,**r**B	E.PD, DEO	External PID instruction when DEO implemented.
divd	**r**D,**r**A,**r**B	64	—
divd.	**r**D,**r**A,**r**B	64	—
divdo	**r**D,**r**A,**r**B	64	—
divdo.	**r**D,**r**A,**r**B	64	—
divdu	**r**D,**r**A,**r**B	64	—
divdu.	**r**D,**r**A,**r**B	64	—
divduo	**r**D,**r**A,**r**B	64	—
divduo.	**r**D,**r**A,**r**B	64	—
divw	**r**D,**r**A,**r**B	Base	—
divw.	**r**D,**r**A,**r**B	Base	—
divwo	**r**D,**r**A,**r**B	Base	—
divwo.	**r**D,**r**A,**r**B	Base	—
divwu	**r**D,**r**A,**r**B	Base	—
divwu.	**r**D,**r**A,**r**B	Base	—
divwuo	**r**D,**r**A,**r**B	Base	—
divwuo.	**r**D,**r**A,**r**B	Base	—
dnh	DUI,DUIS	E.ED	—
dsn	**r**A,**r**B	DS	—
efdabs	**r**D,**r**A	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdadd	**r**D,**r**A,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdcfs	**r**D,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdcfsf	**r**D,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdcfsi	**r**D,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdcfuf	**r**D,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
efdcfui	**r**D,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdcmpeq	**crD**,**r**A,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdcmpgt	**crD**,**r**A,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdcmplt	**crD**,**r**A,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdctsf	**r**D,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdctsi	**r**D,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdctsiz	**r**D,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdctuf	**r**D,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdctui	**r**D,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdctuiz	**r**D,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efddiv	**r**D,**r**A,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdmul	**r**D,**r**A,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdnabs	**r**D,**r**A	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdneg	**r**D,**r**A	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdsub	**r**D,**r**A,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdtsteq	**crD**,**r**A,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdtstgt	**crD**,**r**A,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efdtstlt	**crD**,**r**A,**r**B	SP.FD	SPE embedded scalar double-precision floating-point. See the SPE PEM.
efsabs	**r**D,**r**A	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efsadd	**r**D,**r**A,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efscfsf	**r**D,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efscfsi	**r**D,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efscfuf	**r**D,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efscfui	**r**D,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efscmpeq	**crD**,**r**A,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efscmpgt	**crD**,**r**A,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efscmplt	**crD**,**r**A,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efsctsf	**r**D,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efsctsi	**r**D,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efsctsiz	**r**D,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efsctuf	**r**D,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efsctui	**r**D,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efsctuiz	**r**D,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
efsdiv	**r**D,**r**A,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efsmul	**r**D,**r**A,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efsnabs	**r**D,**r**A	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efsneg	**r**D,**r**A	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efssub	**r**D,**r**A,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efststeq	**crD**,**r**A,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efststgt	**crD**,**r**A,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
efststlt	**crD**,**r**A,**r**B	SP.FS	SPE embedded scalar single-precision floating-point. See the SPE PEM.
ehpriv	OC	E.HV	—
eqv	**r**A,**r**S,**r**B	Base	—
eqv.	**r**A,**r**S,**r**B	Base	—
evabs	**r**D,**r**A	SP	SPE. See the SPE PEM.
evaddiw	**r**D,**r**B,UIMM	SP	SPE. See the SPE PEM.
evaddsmiaaw	**r**D,**r**A	SP	SPE. See the SPE PEM.
evaddssiaaw	**r**D,**r**A	SP	SPE. See the SPE PEM.
evaddumiaaw	**r**D,**r**A	SP	SPE. See the SPE PEM.
evaddusiaaw	**r**D,**r**A	SP	SPE. See the SPE PEM.
evaddw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evand	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evandc	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evcmpeq	**crD**,**r**A,**r**B	SP	SPE. See the SPE PEM.
evcmpgts	**crD**,**r**A,**r**B	SP	SPE. See the SPE PEM.
evcmpgtu	**crD**,**r**A,**r**B	SP	SPE. See the SPE PEM.
evcmplts	**crD**,**r**A,**r**B	SP	SPE. See the SPE PEM.
evcmpltu	**crD**,**r**A,**r**B	SP	SPE. See the SPE PEM.
evcntlsw	**r**D,**r**A	SP	SPE. See the SPE PEM.
evcntlzw	**r**D,**r**A	SP	SPE. See the SPE PEM.
evdivws	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evdivwu	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
eveqv	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evextsb	**r**D,**r**A	SP	SPE. See the SPE PEM.
evextsh	**r**D,**r**A	SP	SPE. See the SPE PEM.
evfsabs	**r**D,**r**A	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
evfsadd	**r**D,**r**A,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfscfsf	**r**D,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfscfsi	**r**D,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfscfuf	**r**D,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfscfui	**r**D,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfscmpeq	**crD**,**r**A,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfscmpgt	**crD**,**r**A,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfscmplt	**crD**,**r**A,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfsctsf	**r**D,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfsctsi	**r**D,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfsctsiz	**r**D,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfsctuf	**r**D,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfsctui	**r**D,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfsctuiz	**r**D,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfsdiv	**r**D,**r**A,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfsmul	**r**D,**r**A,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfsnabs	**r**D,**r**A	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfsneg	**r**D,**r**A	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfssub	**r**D,**r**A,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfststeq	**crD**,**r**A,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfststgt	**crD**,**r**A,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evfststlt	**crD**,**r**A,**r**B	SP.FV	SPE embedded vector single-precision floating-point. See the SPE PEM.
evldd	**r**D,d(**r**A)	SP	SPE. See the SPE PEM.
evlddepx	**r**D,**r**A,**r**B	SP, E.PD	External PID instruction when SPE implemented.
evlddx	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evldh	**r**D,d(**r**A)	SP	SPE. See the SPE PEM.
evldhx	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evldw	**r**D,d(**r**A)	SP	SPE. See the SPE PEM.
evldwx	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evlhhesplat	**r**D,d(**r**A)	SP	SPE. See the SPE PEM.
evlhhesplatx	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evlhhossplat	**r**D,d(**r**A)	SP	SPE. See the SPE PEM.
evlhhossplatx	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
evlhhousplat	**r**D,d(**r**A)	SP	SPE. See the SPE PEM.
evlhhousplatx	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evlwhe	**r**D,d(**r**A)	SP	SPE. See the SPE PEM.
evlwhex	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evlwhos	**r**D,d(**r**A)	SP	SPE. See the SPE PEM.
evlwhosx	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evlwhou	**r**D,d(**r**A)	SP	SPE. See the SPE PEM.
evlwhoux	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evlwhsplat	**r**D,d(**r**A)	SP	SPE. See the SPE PEM.
evlwhsplatx	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evlwwsplat	**r**D,d(**r**A)	SP	SPE. See the SPE PEM.
evlwwsplatx	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmergehi	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmergehilo	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmergelo	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmergelohi	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhegsmfaa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhegsmfan	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhegsmiaa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhegsmian	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhegumiaa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhegumian	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhesmf	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhesmfa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhesmfaaw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhesmfanw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhesmi	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhesmia	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhesmiaaw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhesmianw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhessf	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhessfa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhessfaaw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
evmhessfanw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhessiaaw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhessianw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmheumi	rD,rA,rB	SP	SPE. See the SPE PEM.
evmheumia	rD,rA,rB	SP	SPE. See the SPE PEM.
evmheumiaaw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmheumianw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmheusiaaw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmheusianw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhogsmfaa	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhogsmfan	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhogsmiaa	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhogsmian	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhogumiaa	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhogumian	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhosmf	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhosmfa	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhosmfaaw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhosmfanw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhosmi	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhosmia	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhosmiaaw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhosmianw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhossf	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhossfa	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhossfaaw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhossfanw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhossiaaw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhossianw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhoumi	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhoumia	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhoumiaaw	rD,rA,rB	SP	SPE. See the SPE PEM.
evmhoumianw	rD,rA,rB	SP	SPE. See the SPE PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
evmhousiaaw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmhousianw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmra	**r**D,**r**A	SP	SPE. See the SPE PEM.
evmwhsmf	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwhsmfa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwhsmi	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwhsmia	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwhssf	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwhssfa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwhumi	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwhumia	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwlsmiaaw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwlsmianw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwlssiaaw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwlssianw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwlumi	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwlumia	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwlumiaaw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwlumianw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwlusiaaw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwlusianw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwsmf	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwsmfa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwsmfaa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwsmfan	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwsmi	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwsmia	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwsmiaa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwsmian	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwssf	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwssfa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwssfaa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwssfan	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
evmwumi	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwumia	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwumiaa	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evmwumian	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evnand	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evneg	**r**D,**r**A	SP	SPE. See the SPE PEM.
evnor	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evor	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evorc	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evrlw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evrlwi	**r**D,**r**A,UIMM	SP	SPE. See the SPE PEM.
evrndw	**r**D,**r**A	SP	SPE. See the SPE PEM.
evsel	**r**D,**r**A,**r**B,**crf**S	SP	SPE. See the SPE PEM.
evslw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evslwi	**r**D,**r**A,UIMM	SP	SPE. See the SPE PEM.
evsplatfi	**r**D,SIMM	SP	SPE. See the SPE PEM.
evsplati	**r**D,SIMM	SP	SPE. See the SPE PEM.
evsrwis	**r**D,**r**A,UIMM	SP	SPE. See the SPE PEM.
evsrwiu	**r**D,**r**A,UIMM	SP	SPE. See the SPE PEM.
evsrws	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evsrwu	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evstdd	**r**S,d(**r**A)	SP	SPE. See the SPE PEM.
evstddepx	**r**S,**r**A,**r**B	SP, E.PD	External PID instruction when SPE implemented.
evstddx	**r**S,**r**A,**r**B	SP	SPE. See the SPE PEM.
evstdh	**r**S,d(**r**A)	SP	SPE. See the SPE PEM.
evstdhx	**r**S,**r**A,**r**B	SP	SPE. See the SPE PEM.
evstdw	**r**S,d(**r**A)	SP	SPE. See the SPE PEM.
evstdwx	**r**S,**r**A,**r**B	SP	SPE. See the SPE PEM.
evstwhe	**r**S,d(**r**A)	SP	SPE. See the SPE PEM.
evstwhex	**r**S,**r**A,**r**B	SP	SPE. See the SPE PEM.
evstwho	**r**S,d(**r**A)	SP	SPE. See the SPE PEM.
evstwhox	**r**S,**r**A,**r**B	SP	SPE. See the SPE PEM.
evstwwe	**r**S,d(**r**A)	SP	SPE. See the SPE PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
evstwwex	**r**S,**r**A,**r**B	SP	SPE. See the SPE PEM.
evstwwo	**r**S,d(**r**A)	SP	SPE. See the SPE PEM.
evstwwox	**r**S,**r**A,**r**B	SP	SPE. See the SPE PEM.
evsubfsmiaaw	**r**D,**r**A	SP	SPE. See the SPE PEM.
evsubfssiaaw	**r**D,**r**A	SP	SPE. See the SPE PEM.
evsubfumiaaw	**r**D,**r**A	SP	SPE. See the SPE PEM.
evsubfusiaaw	**r**D,**r**A	SP	SPE. See the SPE PEM.
evsubfw	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
evsubifw	**r**D,**r**B,UIMM	SP	SPE. See the SPE PEM.
evxor	**r**D,**r**A,**r**B	SP	SPE. See the SPE PEM.
extsb	**r**A,**r**S	Base	—
extsb.	**r**A,**r**S	Base	—
extsh	**r**A,**r**S	Base	—
extsh.	**r**A,**r**S	Base	—
extsw	**r**A,**r**S	64	—
extsw.	**r**A,**r**S	64	—
fabs	**fr**D,**fr**B	Floating-point	—
fabs.	**fr**D,**fr**B	FP.R	—
fadd	**fr**D,**fr**A,**fr**B	Floating-point	—
fadd.	**fr**D,**fr**A,**fr**B	FP.R	—
fadds	**fr**D,**fr**A,**fr**B	Floating-point	—
fadds.	**fr**D,**fr**A,**fr**B	FP.R	—
fcfid	**fr**D,**fr**B	Floating-point	—
fcfid.	**fr**D,**fr**B	FP.R	—
fcmpo	**crD**,**fr**A,**fr**B	Floating-point	—
fcmpu	**crD**,**fr**A,**fr**B	Floating-point	—
fctid	**fr**D,**fr**B	Floating-point	—
fctid.	**fr**D,**fr**B	FP.R	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
fctidz	**fr**D,**fr**B	Floating-point	—
fctidz.	**fr**D,**fr**B	FP.R	—
fctiw	**fr**D,**fr**B	Floating-point	—
fctiw.	**fr**D,**fr**B	FP.R	—
fctiwz	**fr**D,**fr**B	Floating-point	—
fctiwz.	**fr**D,**fr**B	FP.R	—
fdiv	**fr**D,**fr**A,**fr**B	Floating-point	—
fdiv.	**fr**D,**fr**A,**fr**B	FP.R	—
fdivs	**fr**D,**fr**A,**fr**B	Floating-point	—
fdivs.	**fr**D,**fr**A,**fr**B	FP.R	—
fmadd	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	—
fmadd.	**fr**D,**fr**A,**fr**C,**fr**B	FP.R	—
fmadds	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	—
fmadds.	**fr**D,**fr**A,**fr**C,**fr**B	FP.R	—
fmr	**fr**D,**fr**B	Floating-point	—
fmr.	**fr**D,**fr**B	FP.R	—
fmsub	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	—
fmsub.	**fr**D,**fr**A,**fr**C,**fr**B	FP.R	—
fmsubs	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	—
fmsubs.	**fr**D,**fr**A,**fr**C,**fr**B	FP.R	—
fmul	**fr**D,**fr**A,**fr**C	Floating-point	—
fmul.	**fr**D,**fr**A,**fr**C	FP.R	—
fmuls	**fr**D,**fr**A,**fr**C	Floating-point	—
fmuls.	**fr**D,**fr**A,**fr**C	FP.R	—
fnabs	**fr**D,**fr**B	Floating-point	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
fnabs.	**fr**D,**fr**B	FP.R	—
fneg	**fr**D,**fr**B	Floating-point	—
fneg.	**fr**D,**fr**B	FP.R	—
fnmadd	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	—
fnmadd.	**fr**D,**fr**A,**fr**C,**fr**B	FP.R	—
fnmadds	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	—
fnmadds.	**fr**D,**fr**A,**fr**C,**fr**B	FP.R	—
fnmsub	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	—
fnmsub.	**fr**D,**fr**A,**fr**C,**fr**B	FP.R	—
fnmsubs	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	—
fnmsubs.	**fr**D,**fr**A,**fr**C,**fr**B	FP.R	—
fres	**fr**D,**fr**B	Floating-point	—
fres.	**fr**D,**fr**B	FP.R	—
frsp	**fr**D,**fr**B	Floating-point	—
frsp.	**fr**D,**fr**B	FP.R	—
frsqrte	**fr**D,**fr**B	Floating-point	—
frsqrte.	**fr**D,**fr**B	FP.R	—
fsel	**fr**D,**fr**A,**fr**C,**fr**B	Floating-point	—
fsel.	**fr**D,**fr**A,**fr**C,**fr**B	FP.R	—
fsub	**fr**D,**fr**A,**fr**B	Floating-point	—
fsub.	**fr**D,**fr**A,**fr**B	FP.R	—
fsubs	**fr**D,**fr**A,**fr**B	Floating-point	—
fsubs.	**fr**D,**fr**A,**fr**B	FP.R	—
icbi	**fr**A,**fr**B	Base	—
icbiep	**r**A,**r**B	E.PD	—
icblc	CT,**r**A,**r**B	E.CL	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
icblq.	CT,**r**A,**r**B	E.CL	—
icbt	CT,**r**A,**r**B	Base	—
icbtls	CT,**r**A,**r**B	E.CL	—
isel	**r**D,**r**A,**r**B,**crb**C	Base	—
isync	—	Base	—
lbarx	**r**D,**r**A,**r**B	ER	—
lbdx	**r**D,**r**A,**r**B	DS	—
lbepx	**r**D,**r**A,**r**B	E.PD	—
lbz	**r**D,d(**r**A)	Base	—
lbzu	**r**D,d(**r**A)	Base	—
lbzux	**r**D,**r**A,**r**B	Base	—
lbzx	**r**D,**r**A,**r**B	Base	—
ld	**r**D,d(**r**A)	64	—
ldarx	**r**D,**r**A,**r**B	64	—
ldbrx	**r**D,**r**A,**r**B	64	—
lddx	**r**D,**r**A,**r**B	64, DS	Decorated storage instruction when 64 implemented.
ldepx	**r**D,**r**A,**r**B	64, E.PD	External PID instruction when 64 implemented.
ldu	**r**D,d(**r**A)	64	—
ldux	**r**D,**r**A,**r**B	64	—
ldx	**r**D,**r**A,**r**B	64	—
lfd	**fr**D,**d(r**A**)**	Floating-point	—
lfddx	**fr**D,**r**A,**r**B	Floating-point, DS	Decorated storage instruction when Floating-point implemented.
lfdepx	**fr**D,**r**A,**r**B	Floating-point, E.PD	External PID instruction when Floating-point implemented.
lfdu	**fr**D,d(**r**A)	Floating-point	—
lfdux	**fr**D,**r**A,**r**B	Floating-point	—
lfdx	**fr**D,**r**A,**r**B	Floating-point	—
lfs	**fr**D,d(**r**A)	Floating-point	—
lfsu	**fr**D,d(**r**A)	Floating-point	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
lfsux	**fr**D,**r**A,**r**B	Floating-point	—
lfsx	**fr**D,**r**A,**r**B	Floating-point	—
lha	**r**D,d(**r**A)	Base	—
lharx	**r**D,**r**A,**r**B	ER	—
lhau	**r**D,d(**r**A)	Base	—
lhaux	**r**D,**r**A,**r**B	Base	—
lhax	**r**D,**r**A,**r**B	Base	—
lhbrx	**r**D,**r**A,**r**B	Base	—
lhdx	**r**D,**r**A,**r**B	DS	—
lhepx	**r**D,**r**A,**r**B	E.PD	—
lhz	**r**D,d(**r**A)	Base	—
lhzu	**r**D,d(**r**A)	Base	—
lhzux	**r**D,**r**A,**r**B	Base	—
lhzx	**r**D,**r**A,**r**B	Base	—
lmw	**r**D,d(**r**A)	Base	—
lvebx	**v**D,**r**A,**r**B	Vector	AltiVec. See the AltiVec PEM.
lvehx	**v**D,**r**A,**r**B	Vector	AltiVec. See the AltiVec PEM.
lvepx	**v**D,**r**A,**r**B	Vector, E.PD	External PID instruction when AltiVec implemented.
lvepxl	**v**D,**r**A,**r**B	Vector, E.PD	External PID instruction when AltiVec implemented.
lvewx	**v**D,**r**A,**r**B	Vector	AltiVec. See the AltiVec PEM.
lvsl	**v**D,**r**A,**r**B	Vector	AltiVec. See the AltiVec PEM.
lvsr	**v**D,**r**A,**r**B	Vector	AltiVec. See the AltiVec PEM.
lvx	**v**D,**r**A,**r**B	Vector	AltiVec. See the AltiVec PEM.
lvxl	**v**D,**r**A,**r**B	Vector	AltiVec. See the AltiVec PEM.
lwa	**r**D,d(**r**A)	64	—
lwarx	**r**D,**r**A,**r**B	Base	—
lwaux	**r**D,**r**A,**r**B	64	—
lwax	**r**D,**r**A,**r**B	64	—
lwbrx	**r**D,**r**A,**r**B	Base	—
lwdx	**r**D,**r**A,**r**B	DS	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
lwepx	**r**D,**r**A,**r**B	E.PD	—
lwz	**r**D,d(**r**A)	Base	—
lwzu	**r**D,d(**r**A)	Base	—
lwzux	**r**D,**r**A,**r**B	Base	—
lwzx	**r**D,**r**A,**r**B	Base	—
mbar	MO	Embedded	—
mcrf	**crD,crf**S	Base	—
mcrfs	**crD,crf**S_FP	Base	—
mcrxr	**crD**	Base	—
mfcr	**r**D	Base	—
mfdcr	**r**D,DCRN	E.DC	—
mffs	**fr**D	Floating-point	—
mffs.	**fr**D	FP.R	—
mfmsr	**r**D	Base	—
mfocrf	**r**D,CRM	Base	—
mfpmr	**r**D,PMRN	E.PM	—
mfspr	**r**D,SPR	Base	—
mftb	**r**D,SPR	Base	—
mfvscr	**v**D	Vector	AltiVec. See the AltiVec PEM.
miso	—	Base	**miso** is an alias for **or r26,r26,r26**
msgclr	**r**B	E.PC	—
msgsnd	**r**B	E.PC	—
mtcrf	C**R**M,**r**S	Base	—
mtdcr	DCRN,**r**S	E.DC	—
mtfsb0	**crb**D_FP	Floating-point	—
mtfsb0.	**crb**D_FP	FP.R	—
mtfsb1	**crb**D_FP	Floating-point	—
mtfsb1.	**crb**D_FP	FP.R	—
mtfsf	FM,fB	Floating-point	—
mtfsf.	FM,fB	FP.R	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
mtfsfi	**crD**_FP,FP_IMM	Floating-point	—
mtfsfi.	**crD**_FP,FP_IMM	FP.R	—
mtmsr	**r**S	Base	—
mtocrf	CRM,**r**S	Base	—
mtpmr	PMRN,**r**S	E.PM	—
mtspr	SPR,**r**S	Base	—
mtvscr	**v**B	Vector	AltiVec. See the AltiVec PEM.
mulhd	**r**D,**r**A,**r**B	64	—
mulhd.	**r**D,**r**A,**r**B	64	—
mulhdu	**r**D,**r**A,**r**B	64	—
mulhdu.	**r**D,**r**A,**r**B	64	—
mulhw	**r**D,**r**A,**r**B	Base	—
mulhw.	**r**D,**r**A,**r**B	Base	—
mulhwu	**r**D,**r**A,**r**B	Base	—
mulhwu.	**r**D,**r**A,**r**B	Base	—
mulld	**r**D,**r**A,**r**B	64	—
mulld.	**r**D,**r**A,**r**B	64	—
mulldo	**r**D,**r**A,**r**B	64	—
mulldo.	**r**D,**r**A,**r**B	64	—
mulli	**r**D,**r**A,SIMM	Base	—
mullw	**r**D,**r**A,**r**B	Base	—
mullw.	**r**D,**r**A,**r**B	Base	—
mullwo	**r**D,**r**A,**r**B	Base	—
mullwo.	**r**D,**r**A,**r**B	Base	—
nand	**r**A,**r**S,**r**B	Base	—
nand.	**r**A,**r**S,**r**B	Base	—
neg	**r**D,**r**A	Base	—
neg.	**r**D,**r**A	Base	—
nego	**r**D,**r**A	Base	—
nego.	**r**D,**r**A	Base	—
nor	**r**A,**r**S,**r**B	Base	—
nor.	**r**A,**r**S,**r**B	Base	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
or	rA,rS,rB	Base	—
or.	rA,rS,rB	Base	—
orc	rA,rS,rB	Base	—
orc.	rA,rS,rB	Base	—
ori	rA,rS,UIMM	Base	—
oris	rA,rS,UIMM	Base	—
popcntb	rA,rS	Base	—
popcntd	rA,rS	64	—
popcntw	rA,rS	Base	—
prtyd	rA,rS	64	—
prtyw	rA,rS	Base	—
rfci	—	Embedded	—
rfdi	—	E.ED	—
rfgi	—	E.HV	—
rfi	—	Embedded	—
rfmci	—	Embedded	—
rldcl	rA,rS,rB,MB	64	—
rldcl.	rA,rS,rB,MB	64	—
rldcr	rA,rS,rB,ME	64	—
rldcr.	rA,rS,rB,ME	64	—
rldic	rA,rS,SH,MB	64	—
rldic.	rA,rS,SH,MB	64	—
rldicl	rA,rS,SH,MB	64	—
rldicl.	rA,rS,SH,MB	64	—
rldicr	rA,rS,SH,ME	64	—
rldicr.	rA,rS,SH,ME	64	—
rldimi	rA,rS,SH,MB	64	—
rldimi.	rA,rS,SH,MB	64	—
rlwimi	rA,rS,SH,MB,ME	Base	—
rlwimi.	rA,rS,SH,MB,ME	Base	—
rlwinm	rA,rS,SH,MB,ME	Base	—
rlwinm.	rA,rS,SH,MB,ME	Base	—
rlwnm	rA,rS,rB,MB,ME	Base	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
rlwnm.	rA,rS,rB,MB,ME	Base	—
sc	LEV	Base	—
sld	rA,rS,rB	64	—
sld.	rA,rS,rB	64	—
slw	rA,rS,rB	Base	—
slw.	rA,rS,rB	Base	—
srad	rA,rS,rB	64	—
srad.	rA,rS,rB	64	—
sradi	rA,rS,SH	64	—
sradi.	rA,rS,SH	64	—
sraw	rA,rS,rB	Base	—
sraw.	rA,rS,rB	Base	—
srawi	rA,rS,SH	Base	—
srawi.	rA,rS,SH	Base	—
srd	rA,rS,rB	64	—
srd.	rA,rS,rB	64	—
srw	rA,rS,rB	Base	—
srw.	rA,rS,rB	Base	—
stb	rS,d(rA)	Base	—
stbcx.	rS,rA,rB	ER	—
stbdx	rS,rA,rB	DS	—
stbepx	rS,rA,rB	E.PD	—
stbu	rS,d(rA)	Base	—
stbux	rS,rA,rB	Base	—
stbx	rS,rA,rB	Base	—
std	rS,d(rA)	64	—
stdbrx	rS,rA,rB	64	—
stdcx.	rS,rA,rB	64	—
stddx	rS,rA,rB	64, DS	Decorated storage instruction when 64 implemented.
stdepx	rS,rA,rB	64, E.PD	External PID instruction when 64 implemented.
stdu	rS,d(rA)	64	—
stdux	rS,rA,rB	64	—
stdx	rS,rA,rB	64	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
stfd	**fr**S,d(**r**A)	Floating-point	—
stfddx	**fr**S,**r**A,**r**B	Floating-point, DS	Decorated storage instruction when Floating-point implemented.
stfdepx	**fr**S,**r**A,**r**B	Floating-point, E.PD	External PID instruction when Floating-point implemented.
stfdu	**fr**S,d(**r**A)	Floating-point	—
stfdux	**fr**S,**r**A,**r**B	Floating-point	—
stfdx	**fr**S,**r**A,**r**B	Floating-point	—
stfiwx	**fr**S,**r**A,**r**B	Floating-point	—
stfs	**fr**S,d(**r**A)	Floating-point	—
stfsu	**fr**S,d(**r**A)	Floating-point	—
stfsux	**fr**S,**r**A,**r**B	Floating-point	—
stfsx	**fr**S,**r**A,**r**B	Floating-point	—
sth	**r**S,d(**r**A)	Base	—
sthbrx	**r**S,**r**A,**r**B	Base	—
sthcx.	**r**S,**r**A,**r**B	ER	—
sthdx	**r**S,**r**A,**r**B	DS	—
sthepx	**r**S,**r**A,**r**B	E.PD	—
sthu	**r**S,d(**r**A)	Base	—
sthux	**r**S,**r**A,**r**B	Base	—
sthx	**r**S,**r**A,**r**B	Base	—
stmw	**r**S,d(**r**A)	Base	—
stvebx	**v**S,**r**A,**r**B	Vector	AltiVec. See the AltiVec PEM.
stvehx	**v**S,**r**A,**r**B	Vector	AltiVec. See the AltiVec PEM.
stvepx	**v**S,**r**A,**r**B	Vector, E.PD	External PID instruction when AltiVec implemented.
stvepxl	**v**S,**r**A,**r**B	Vector, E.PD	External PID instruction when AltiVec implemented.
stvewx	**v**S,**r**A,**r**B	Vector	AltiVec. See the AltiVec PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
stvx	**v**S,**r**A,**r**B	Vector	AltiVec. See the AltiVec PEM.
stvxl	**v**S,**r**A,**r**B	Vector	AltiVec. See the AltiVec PEM.
stw	**r**S,d(**r**A)	Base	—
stwbrx	**r**S,**r**A,**r**B	Base	—
stwcx.	**r**S,**r**A,**r**B	Base	—
stwdx	**r**S,**r**A,**r**B	DS	—
stwepx	**r**S,**r**A,**r**B	E.PD	—
stwu	**r**S,d(**r**A)	Base	—
stwux	**r**S,**r**A,**r**B	Base	—
stwx	**r**S,**r**A,**r**B	Base	—
subf	**r**D,**r**A,**r**B	Base	—
subf.	**r**D,**r**A,**r**B	Base	—
subfc	**r**D,**r**A,**r**B	Base	—
subfc.	**r**D,**r**A,**r**B	Base	—
subfco	**r**D,**r**A,**r**B	Base	—
subfco.	**r**D,**r**A,**r**B	Base	—
subfe	**r**D,**r**A,**r**B	Base	—
subfe.	**r**D,**r**A,**r**B	Base	—
subfeo	**r**D,**r**A,**r**B	Base	—
subfeo.	**r**D,**r**A,**r**B	Base	—
subfic	**r**D,**r**A,SIMM	Base	—
subfme	**r**D,**r**A	Base	—
subfme.	**r**D,**r**A	Base	—
subfmeo	**r**D,**r**A	Base	—
subfmeo.	**r**D,**r**A	Base	—
subfo	**r**D,**r**A,**r**B	Base	—
subfo.	**r**D,**r**A,**r**B	Base	—
subfze	**r**D,**r**A	Base	—
subfze.	**r**D,**r**A	Base	—
subfzeo	**r**D,**r**A	Base	—
subfzeo.	**r**D,**r**A	Base	—

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
sync **(msync)** **(lwsync)** **(hwsync)**	L	Base	—
td	TO,**r**A,**r**B	64	—
tdi	TO,**r**A,SIMM	64	—
tlbilx	T,**r**A,**r**B	E.HV	—
tlbivax	**r**A,**r**B	Embedded	—
tlbre	—	Embedded	—
tlbsx	**r**A,**r**B	Embedded	—
tlbsync	—	Embedded	—
tlbwe	—	Embedded	—
tw	TO,**r**A,**r**B	Base	—
twi	TO,**r**A,SIMM	Base	—
vaddcuw	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vaddfp	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vaddsbs	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vaddshs	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vaddsws	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vaddubm	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vaddubs	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vadduhm	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vadduhs	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vadduwm	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vadduws	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vand	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vandc	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vavgsb	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vavgsh	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vavgsw	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vavgub	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vavguh	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vavguw	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
vcfsx	vD,vB,UIMM	Vector	AltiVec. See the AltiVec PEM.
vcfux	vD,vB,UIMM	Vector	AltiVec. See the AltiVec PEM.
vcmpbfp*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpbfp*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpeqfp*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpeqfp*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpequb*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpequb*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpequh*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpequh*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpequw*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpequw*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgefp*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgefp*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtfp*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtfp*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtsb*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtsb*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtsh*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtsh*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtsw*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtsw*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtub*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtub*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtuh*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtuh*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtuw*x*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vcmpgtuw*x.*	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vctsxs	vD,vB,UIMM	Vector	AltiVec. See the AltiVec PEM.
vctuxs	vD,vB,UIMM	Vector	AltiVec. See the AltiVec PEM.
vexptefp	vD,vB	Vector	AltiVec. See the AltiVec PEM.
vlogefp	vD,vB	Vector	AltiVec. See the AltiVec PEM.
vmaddfp	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
vmaxfp	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmaxsb	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmaxsh	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmaxsw	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmaxub	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmaxuh	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmaxuw	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmhaddshs	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.
vmhraddshs	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.
vminfp	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vminsb	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vminsh	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vminsw	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vminub	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vminuh	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vminuw	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmladduhm	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.
vmrghb	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmrghh	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmrghw	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmrglb	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmrglh	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmrglw	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmsummbm	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.
vmsumshm	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.
vmsumshs	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.
vmsumubm	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.
vmsumuhm	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.
vmsumuhs	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.
vmulesb	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmulesh	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmuleub	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmuleuh	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
vmulosb	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmulosh	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmuloub	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vmulouh	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vnmsubfp	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.
vnor	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vor	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vperm	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.
vpkpx	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vpkshss	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vpkshus	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vpkswss	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vpkswus	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vpkuhum	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vpkuhus	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vpkuwum	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vpkuwus	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vrefp	vD,vB	Vector	AltiVec. See the AltiVec PEM.
vrfim	vD,vB	Vector	AltiVec. See the AltiVec PEM.
vrfin	vD,vB	Vector	AltiVec. See the AltiVec PEM.
vrfip	vD,vB	Vector	AltiVec. See the AltiVec PEM.
vrfiz	vD,vB	Vector	AltiVec. See the AltiVec PEM.
vrlb	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vrlh	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vrlw	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vrsqrtefp	vD,vB	Vector	AltiVec. See the AltiVec PEM.
vsel	vD,vA,vB,vC	Vector	AltiVec. See the AltiVec PEM.
vsl	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vslb	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vsldoi	vD,vA,vB,SHB	Vector	AltiVec. See the AltiVec PEM.
vslh	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vslo	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.
vslw	vD,vA,vB	Vector	AltiVec. See the AltiVec PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
vspltb	**v**D,**v**B,UIMM	Vector	AltiVec. See the AltiVec PEM.
vsplth	**v**D,**v**B,UIMM	Vector	AltiVec. See the AltiVec PEM.
vspltisb	**v**D,UIMM	Vector	AltiVec. See the AltiVec PEM.
vspltish	**v**D,UIMM	Vector	AltiVec. See the AltiVec PEM.
vspltisw	**v**D,UIMM	Vector	AltiVec. See the AltiVec PEM.
vspltw	**v**D,**v**B,UIMM	Vector	AltiVec. See the AltiVec PEM.
vsr	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsrab	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsrah	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsraw	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsrb	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsrh	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsro	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsrw	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsubcuw	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsubfp	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsubsbs	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsubshs	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsubsws	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsububm	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsububs	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsubuhm	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsubuhs	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsubuwm	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsubuws	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsum2sws	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsum4sbs	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsum4shs	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsum4ubs	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vsumsws	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
vupkhpx	**v**D,**v**B	Vector	AltiVec. See the AltiVec PEM.
vupkhsb	**v**D,**v**B	Vector	AltiVec. See the AltiVec PEM.
vupkhsh	**v**D,**v**B	Vector	AltiVec. See the AltiVec PEM.

Table 4-54. Instruction Set (continued)

Mnemonic	Syntax	Category	Comments
vupklpx	**v**D,**v**B	Vector	AltiVec. See the AltiVec PEM.
vupklsb	**v**D,**v**B	Vector	AltiVec. See the AltiVec PEM.
vupklsh	**v**D,**v**B	Vector	AltiVec. See the AltiVec PEM.
vxor	**v**D,**v**A,**v**B	Vector	AltiVec. See the AltiVec PEM.
wait	—	WT	—
wrtee	**r**S	Embedded	—
wrteei	E	Embedded	—
xor	**r**A,**r**S,**r**B	Base	—
xor.	**r**A,**r**S,**r**B	Base	—
xori	**r**A,**r**S,UIMM	Base	—
xoris	**r**A,**r**S,UIMM	Base	—

Chapter 5
Instruction Set

This chapter describes the following instructions:

- Instructions defined by the EIS, including instructions that may not be implemented in all devices.

This chapter does not describe EIS instructions defined by the AltiVec (category Vector), signal processing engine (category SPE), or variable-length encoding (category VLE) extensions, which are described in their respective programming environments manuals (PEMs). For information about these documents, refer to the Preface.

5.1 Notation

The following definitions and notation are used throughout this chapter in the instruction descriptions.

Table 5-1. Notation Conventions

Symbol	Meaning
X_p	Bit p of register/field X

X_m	Where “m” is used to describe a bit position, “m” is 0 if the processor is in 64-bit mode or 32 if the processor is in 32-bit mode. More specifically:
	`if MSR`$_{CM}$` = 0 then m ← 32`
	`else              m ← 0`

X_{field}	The bits composing a defined field of X. For example, X_{sign}, X_{exp}, and X_{frac} represent the sign, exponent, and fractional value of a floating-point number X.
$X_{p:q}$	Bits p through q of register/field X
$X_{p\ q\ ...}$	Bits p, q,... of register/field X
¬X	The one’s complement of the contents of X
Field i	Bits 4×i through 4×i+3 of a register
.	As the last character of an instruction mnemonic, this character indicates that the instruction records status information in certain fields of the condition register as a side effect of execution, as described in Section 3.5.1, “Condition Register (CR).”
||	Describes the concatenation of two values. For example, 010 || 111 is the same as 010111.
x^n	x raised to the n^{th} power

Table 5-1. Notation Conventions (continued)

Symbol	Meaning
^{n}x	The replication of x, n times (i.e., x concatenated to itself n–1 times). $^{n}0$ and $^{n}1$ are special cases: $^{n}0$ means a field of n bits with each bit equal to 0. Thus $^{5}0$ is equivalent to 0b0_0000. $^{n}1$ means a field of n bits with each bit equal to 1. Thus $^{5}1$ is equivalent to 0b1_1111.
/, //, ///,	A reserved field in an instruction, register, field or string. Each bit and field in instructions, in status and control registers (such as the XER or FPSCR), and in SPRs is either defined or reserved. Some defined fields contain reserved values. In such cases when this document refers to the specific field, it refers only to the defined values, unless otherwise specified.

5.1.1 Instruction Bit Numbering

Instruction encodings, unlike registers defined in this document, use a 32-bit numbering scheme where the most significant bit is bit 0 and the least significant bit is bit 31. If an instruction description refers to a register field, the bit numbering described when referencing that register is considered to be 64-bit numbering except for vector registers which use 128-bit numbering.

5.2 Instruction Fields

This table describes instruction fields.

Table 5-2. Instruction Field Descriptions

Field	Description
AA (30)	Absolute address bit. 0 The immediate field represents an address relative to the current instruction address. For I-form branch instructions, the effective address of the branch target is the value ^{32}0 \|\| (CIA+EXTS(LI\|\|0b00))$_{32–63}$. For B-form branch instructions, the effective address of the branch target is the value ^{32}0 \|\| (CIA+EXTS(BD\|\|0b00))$_{32–63}$. For I-form branch extended instructions, the effective address of the branch target is the value CIA+EXTS(LI||0b00). For B-form branch extended instructions, the effective address of the branch target is the value CIA+EXTS(BD||0b00). 1 The immediate field represents an absolute address. For I-form branch instructions, the effective address of the branch target is the value ^{32}0 \|\| EXTS(LI\|\|0b00)$_{32–63}$. For B-form branch instructions, the effective address of the branch target is the value ^{32}0 \|\| EXTS(BD\|\|0b00)$_{32–63}$. For I-form branch extended instructions, the effective address of the branch target is the value EXTS(LI||0b00). For B-form branch extended instructions, the effective address of the branch target is the value EXTS(BD||0b00).
BD(16–29)	Immediate field specifying a 14-bit signed two's complement integer that is concatenated on the right with 0b00 and sign-extended to 64 bits
BH(19–20)	Used to specify a branch hint
BI (11–15)	Used to specify a CR bit to be used as the condition of a branch conditional instruction
BO (6–10)	Used to specify options for branch conditional instructions. See Section 4.6.1.9.2, "Branch Instructions."
crbA (11–15)	Used to specify a condition register bit to be used as a source
crbB (16–20)	Used to specify a condition register bit to be used as a source
crbD (6–10)	Used to specify a CR or FPSCR bit to be used as a target
crfD (6–8)	Used to specify a CR or FPSCR field to be used as a target
CRM (12–19)	Field mask used to identify the condition register fields to be processed
crS (11–13)	Used to specify a CR or FPSCR field to be used as a source
CT (6–10)	Used by cache instructions to specify the target portion of the cache facility to place the prefetched data or instructions and is implementation-dependent
D (16–31)	Immediate field used to specify a 16-bit signed two's complement integer that is sign-extended to 64 bits
dcrn (16–20 || 11–15)	Used to specify a Device Control Register number
DCRN(16–20||11–15)	Used to specify a device control register for the **mtdcr** and **mfdcr** instructions
DUI (6–10)	Field passed to external debugger when **dnh** instruction halts
DUIS (11–20)	Field in **dnh** instruction available for use by external debugger when **dnh** instruction halts

Table 5-2. Instruction Field Descriptions (continued)

Field	Description
E (15)	Immediate field used to specify a 1-bit value used by **wrteei** to place in MSR[EE] (external input enable bit)
FM (7–14)	Field mask used to identify FPSCR fields that are to be updated by the **mtfsf** instruction
frA (11–15)	Used to specify an FPR to be used as a source
frB (16–20)	Used to specify an FPR to be used as a source
frC (21–25)	Used to specify an FPR to be used as a source
frD (6–10)	Used to specify an FPR to be used as a target
frS (6–10)	Used to specify an FPR to be used as a source
L (10,9–10)	Used to specify whether an integer compare instruction is to compare 64-bit numbers or 32-bit number or the type of **sync** operation to perform
LEV (20–26)	Used to specify the level of a system call
LI (6–29)	Immediate field specifying a 24-bit signed two’s complement integer that is concatenated on the right with 0b00 and sign-extended to 64 bits
LK (31)	LINK bit. Indicates whether the link register (LR) is set. 0 Do not set the LR. 1 Set the LR. The sum of the value 4 and the address of the branch instruction is placed into the LR.
MB (21–25)	Field used in M-form rotate instructions to specify a 64-bit mask consisting of 1 bits from bit MB+32 through bit ME+32 inclusive and 0 bits elsewhere
mb (26 || 21–25)	Used in MD-form and MDS-form rotate instructions to specify the first 1-bit of a 64-bit mask
me (26 || 21–25)	Used in MD-form and MDS-form rotate instructions to specify the last 1-bit of a 64-bit mask
ME (26–30)	Field used in M-form rotate instructions to specify a 64-bit mask consisting of 1 bits from bit MB+32 through bit ME+32 inclusive and 0 bits elsewhere
MO (6–10)	Used to specify the subset of memory accesses ordered by a Memory Barrier instruction (**mbar**).
OC (6–20)	Field in **ehpriv** instruction available for use by hypervisor when **ehpriv** instruction generates embedded hypervisor privilege interrupt
OE (21)	Used to specify whether overflow detection is enabled
OPCD (0–5)	Primary opcode field
P (3)	Used to specify whether the instruction is single-precision or double-precision
pmrn (16–20 || 11–15)	Used to specify a Performance Monitor Register number
rA (11–15)	Used to specify a GPR to be used as a source or as a target
rB (16–20)	Used to specify a GPR to be used as a source
Rc (31)	Record bit. 0 Do not alter the condition register. 1 Set condition register field 0 or field 1.
rD (6–10)	Used to specify a GPR to be used as a target
rS (6–10)	Used to specify a GPR to be used as a source

Table 5-2. Instruction Field Descriptions (continued)

Field	Description
SH (16–20)	Used to specify a shift amount in rotate word immediate and shift word immediate instructions
sh (30 || 16–20)	Used to specify a shift amount in rotate double word immediate and shift double word immediate instructions
SIMM (16–31)	Immediate field used to specify a 16-bit signed integer
sprn (16–20 || 11–15)	Used to specify an SPR for **mtspr** and **mfspr** instructions
T (9–10)	Used to specify the type of local TLB invalidation to perform
tbrn (16–20 || 11–15)	Used to specify a Time Base Register number
TO (6–10)	Used to specify the conditions on which to trap. The encoding is described in Section 4.6.1.9.4, “Trap Instructions.”
U (16–19)	Immediate field used as the data to be placed into a field in the FPSCR
UIMM (16–31)	Immediate field used to specify a 16-bit unsigned integer
vD (6–10)	Used to specify a VR to be used as a target
vS (6–10)	Used to specify a VR to be used as a source
WC (9–10)	Used to specify a wait condition for the **wait** instruction
XO (21–29, 21–30, 22–30, 26–30, 27–29, 27–30, 28–31)	Extended opcode field

5.3 Description of Instruction Operations

The operation of most instructions is described by a series of statements using a semiformal language at the register transfer level (RTL), which uses the general notation given in Table 5-1 and Table 5-2 and the RTL-specific conventions in Table 5-3. See the example in Figure 5-1. Some of this notation is used in the formal descriptions of instructions.

The RTL descriptions cover the normal execution of the instruction, except that ‘standard’ setting of the condition register, integer exception register, and floating-point status and control register are not always shown. (Non-standard setting of these registers, such as the setting of condition register field 0 by the **stwcx.** instruction, is shown.) The RTL descriptions do not cover all cases in which exceptions may occur, or for which the results are boundedly undefined, and may not cover all invalid forms.

Instruction operation is described for 64-bit implementations. For 32-bit implementations, only the low-order 32-bits of GPRs and SPRs should be considered.

RTL descriptions specify the architectural transformation performed by the execution of an instruction. They do not imply any particular implementation.

Table 5-3. RTL Notation

Notation	Meaning
←	Assignment
$\leftarrow_f$	Assignment in which the data may be reformatted in the target location
¬	NOT logical operator (one’s complement)
+	Two’s complement addition
–	Two’s complement subtraction, unary minus
×	Multiplication
÷	Division (yielding quotient)
$+_{dp}$	Floating-point addition, double precision
$-_{dp}$	Floating-point subtraction, double precision
$\times_{dp}$	Floating-point multiplication, double precision
$\div_{dp}$	Floating-point division quotient, double precision
$+_{sp}$	Floating-point addition, single precision
$-_{sp}$	Floating-point subtraction, single precision
$\times_{sf}$	Signed fractional multiplication. Result of multiplying two quantities having bit lengths *x* and *y* taking the least significant x+y–1 bits of the product and concatenating a 0 to the least significant bit forming a signed fractional result of x+y bits.
$\times_{si}$	Signed integer multiplication
$\times_{sp}$	Floating-point multiplication, single precision
$\div_{sp}$	Floating-point division, single precision
$\times_{fp}$	Floating-point multiplication to infinite precision (no rounding)
$\times_{ui}$	Unsigned integer multiplication
⊕, ≡	Exclusive OR, Equivalence logical operators ((a≡b) = (a⊕¬b))
&, |	AND, OR logical operators
?	Unordered comparison relation
<, ≤, >, ≥	Signed comparison relations
$<_u$, $>_u$	Unsigned comparison relations
=, ≠	Equals, Not Equals relations
>>, <<	Shift right or left logical
(x)	The contents of register x when x is a register field in an instruction, otherwise used to specify precedence.
ABS(x)	Absolute value of x
AllocateDataCache Block(x,s)	If the block containing the byte addressed by x does not exist in the data cache, allocate a block in the data cache based on EA. Then set s bytes to 0 in the cache block starting at EA & ¬(s-1)
APID(x)	Returns an implementation-dependent information on the presence and status of the auxiliary processing extensions specified by x

Table 5-3. RTL Notation (continued)

Notation	Meaning
CA	XER_{CA}
CacheBlockSize()	The size of a cache block in bytes
Carry(x+y)	The carry bits produced as a result of the addition of x+y
CEIL(x)	Least integer ≥ x
characterization	Reference to the setting of status bits in a standard way that is explained in the text
CIA	Current instruction address, the address of the instruction being described in RTL. Used by relative branches to set the next instruction address (NIA) and by branch instructions with LK=1 to set the LR. CIA does not correspond to any architected register.
clear_received_ message(x)	Any message of message type x is cleared on this processor if it has been received and accepted
ConvertFloat toIntDouble(x,y)	The floating-point value in FPR(x) is converted to an integer doubleword using rounding mode y.
ConvertIntDouble toFloat(x)	The integer doubleword in FPR(x) is converted to floating-point format.
DataCacheBlock ClearLock(c,x)	The cache block x in the data cache specified by CT has any cache line lock with it removed
DCREG(x)	Device control register x
DECORATED_MEM(x,y,z)	MEM(x,y) with decoration z supplied.
DECORATED_MEM_ ADDR_ONLY(x,0,y)	Address only transaction sent to memory address x with decoration y.
DOUBLE(x)	Result of converting x from floating-point single format to floating-point double format
entry	A TLB entry
EXTS(x)	Result of extending x on the left with signed bits
EXTZ(x)	Result of extending x on the left with zeros
FEX	$FPSCR_{FEX}$
FlushDataCache Block(x)	If the block containing the byte addressed by x exists in the data cache and is dirty, the block is written to main memory and is removed from the data cache
FPR(x)	Floating-point register x
FPReciprocal Estimate(x)	Floating-point estimate of $\frac{1}{x}$
FPReciprocalSquareRoot Estimate(x)	Floating-point estimate of $\frac{1}{\sqrt{x}}$
FPRoundtoSingle(x)	Result of rounding x from floating-point double format to floating-point single format
FPSquareRoot Double(x)	Floating-point $\sqrt{x}$, result rounded to double-precision
FPSquareRoot Single(x)	Floating-point $\sqrt{x}$, result rounded to single-precision

Table 5-3. RTL Notation (continued)

Notation	Meaning
FX	FPSCR_{FX}
GPR(x)	General-purpose register x
hint	A TLB entry number which is an implementation dependent defined hint for the next TLB entry to victimized
InstructionCacheBlock ClearLock(c,x)	The cache block x in the instruction cache specified by CT has any cache line lock with it removed
InvalidateDataCache Block(x)	If the block containing the byte addressed by x exists in the data cache, the block is removed from the data cache.
InvalidateInstruction CacheBlock(x)	If the block containing the byte addressed by x is in the instruction cache, the block is removed from the instruction cache.
m	Used as bit number based on whether the processor is in 32-bit or 64-bit mode. • m = 0 if the processor is in 64-bit mode • m = 32 if the processor is in 32-bit mode (or is a 32-bit implementation)
MASK(x, y)	Mask having 1s in bit positions x through y (wrapping if x>y) and 0s elsewhere
MEM(x,y)	Contents of y bytes of memory starting at address x. If y > 1 and y < 16 then: If big-endian memory, the byte at address x is the MSB and the byte at address x+y–1 is the LSB of the value being accessed. If little-endian memory, the byte at address x is the LSB and the byte at address x+y–1 is the MSB of the value being accessed. If y > 15 then: The access is always big endian and the byte at address x is the MSB and the byte at address x+y–1 is the LSB of the value being accessed.
MOD(x,y)	Modulo y of x (remainder of x divided by y)
NIA	Next instruction address, the address of the next instruction to be executed. For a successful branch, the next instruction address is the branch target address: in RTL, this is indicated by assigning a value to NIA. For other instructions that cause non-sequential instruction fetching, the RTL is similar. For instructions that do not branch, and do not otherwise cause instruction fetching to be non-sequential, the next instruction address is CIA+4.
OV	XER_{OV}
OX	FPSCR_{OX}
PrefetchDataCache Block(x,y)	If the block containing the byte addressed by x does not exist in the portion of the data cache specified by y, the block in memory is copied into the data cache.
PrefetchDataCache BlockLockSet(x,y)	If the block containing the byte addressed by x does not exist in the portion of the data cache specified by y, the block in memory is copied into the cache. The block is then locked into the cache specified by y
PrefetchForStore DataCache-Block(x,y)	If the block containing the byte addressed by x does not exist in the portion of the data cache specified by y, the block in memory is copied into the data cache and made exclusive to the processor executing the instruction.
PrefetchInstruction CacheBlock(x,y)	If the block containing the byte addressed by x does not exist in the portion of the instruction cache specified by y, the block in memory is copied into the instruction cache.

Table 5-3. RTL Notation (continued)

Notation	Meaning
PrefetchInstructionCache BlockLockSet(x,y)	If the block containing the byte addressed by x does not exist in the portion of the instruction cache specified by y, the block in memory is copied into the cache. The block is then locked into the cache specified by y
$ROTL_{32}$(x, y)	Result of rotating the 64-bit value x || x left y positions, where x is 32 bits long
$ROTL_{64}$(x, y)	Result of rotating the 64-bit value x left y positions
SelectTLB(x,y,z)	TLB entry defined by TLB array x, entry select y, and effective page number z
send_msg_to_domain(x,y)	A message of message type x with payload y is sent to all devices in the coherence domain
SINGLE(x)	Result of converting x from floating-point double format to floating-point single format
SO	XER_{SO}
SPREG(x)	Special-purpose register x
StoreDataCache Block(x)	If the block containing the byte addressed by x exists the data cache and is dirty, the block is written to main memory but may remain in the data cache.
TRAP	Invoke a trap-type program interrupt
undefined	An undefined value. The value may vary between implementations and between different executions on the same implementation.
VE	$FPSCR_{VE}$
VSXNAN	$FPSCR_{VXSNAN}$
VX	$FPSCR_{VX}$
VXCVI	$FPSCR_{VXCVI}$
VXVC	$FPSCR_{VXVC}$
ZeroDataCache Block(x,s)	Set s bytes to 0 in the cache block starting at EA & ¬(s-1).

if … then …
else …
	Conditional execution, indenting shows range; else is optional

do	Do loop, indenting shows range. ‘To’ and/or ‘by’ clauses specify incrementing an iteration variable, and a ‘while’ clause gives termination conditions.
leave	Leave innermost do loop, or do loop described in leave statement.
$^{32}0$	32 bits of 0

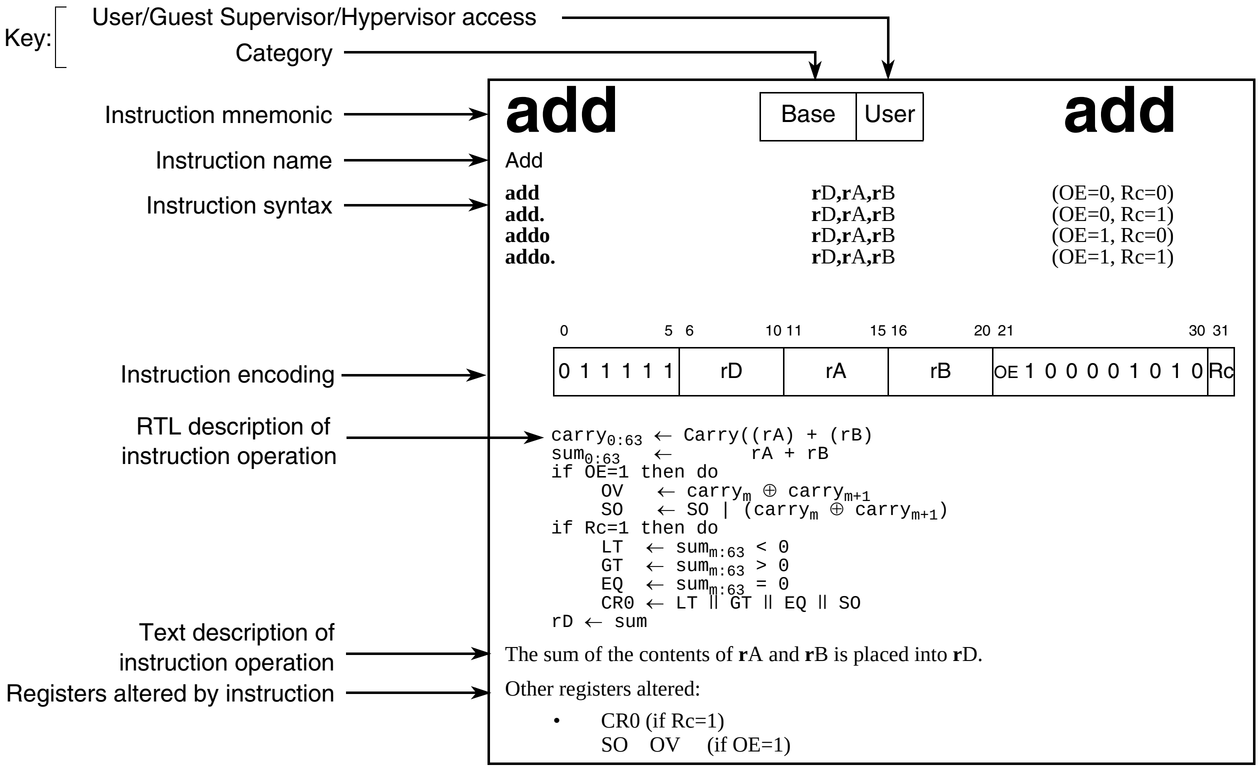
Precedence rules for RTL operators are summarized in Table 5-4. Operators higher in the table are applied before those lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. (For example, – associates from left to right, so a–b–c = (a–b)–c.) Parentheses are used to override the evaluation order implied by the table or to increase clarity; parenthesized expressions are evaluated before serving as operands.

Table 5-4. Operator Precedence

Operators	Associativity
Subscript, function evaluation	Left to right
Pre-superscript (replication), post-superscript (exponentiation)	Right to left
unary –, ¬	Right to left
×, ÷	Left to right
+, –	Left to right
||	Left to right
=, ≠, <, ≤, >, ≥, $<_u$, $>_u$, ?	Left to right
&, ⊕, ≡	Left to right
|	Left to right
: (range)	None
←	None

5.4 Instruction Set

The rest of this chapter describes individual instructions, which are listed in alphabetical order by mnemonic. This figure shows the format for instruction description pages.


Key:
User/Guest Supervisor/Hypervisor access
Category
Instruction mnemonic
Instruction name
Instruction syntax
Instruction encoding
RTL description of instruction operation
Text description of instruction operation
Registers altered by instruction

add | Base | User | **add**

Add

add	**r**D,**r**A,**r**B	(OE=0, Rc=0)
add.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
addo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
addo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)

0–5	6–10	11–15	16–20	21	22–30	31
0 1 1 1 1 1	rD	rA	rB	OE	1 0 0 0 0 1 0 1 0	Rc

```
carry_{0:63} ← Carry((rA) + (rB)
sum_{0:63}   ←       rA + rB
if OE=1 then do
     OV   ← carry_m ⊕ carry_{m+1}
     SO   ← SO | (carry_m ⊕ carry_{m+1})
if Rc=1 then do
     LT  ← sum_{m:63} < 0
     GT  ← sum_{m:63} > 0
     EQ  ← sum_{m:63} = 0
     CR0 ← LT || GT || EQ || SO
rD ← sum
```

The sum of the contents of **r**A and **r**B is placed into **r**D.

Other registers altered:

- CR0 (if Rc=1)
  SO OV (if OE=1)

Figure 5-1. Instruction Description

Note that the execution unit that executes the instruction may not be the same for all processors.

add add

Base	User

Add

add	**r**D,**r**A,**r**B	(OE=0, Rc=0)
add.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
addo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
addo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	OE	1 0 0 0 0 1 0 1 0	Rc

```
$carry_{0:63}$ ← Carry((rA) + (rB))
$sum_{0:63}$   ←       (rA) + (rB)
if OE=1 then do
   OV   ← $carry_m \oplus carry_{m+1}$
   SO   ← SO | ($carry_m \oplus carry_{m+1}$)
if Rc=1 then do
   LT  ← $sum_{m:63} < 0$
   GT  ← $sum_{m:63} > 0$
   EQ  ← $sum_{m:63} = 0$
   CR0 ← LT || GT || EQ || SO
rD ← sum
```

The sum of the contents of **r**A and **r**B is placed into **r**D.

Other registers altered:

- CR0 (if Rc=1)
  SO OV (if OE=1)

addc addc

Base	User

Add Carrying

addc	**r**D,**r**A,**r**B	(OE=0, Rc=0)
addc.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
addco	**r**D,**r**A,**r**B	(OE=1, Rc=0)
addco.	**r**D,**r**A,**r**B	(OE=1, Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	OE	0 0 0 0 0 1 0 1 0	Rc

```
$carry_{0:63}$ ← Carry((rA) + (rB))
$sum_{0:63}$   ←       (rA) + (rB)
if OE=1 then do
   OV   ← $carry_m \oplus carry_{m+1}$
   SO   ← SO | ($carry_m \oplus carry_{m+1}$)
if Rc=1 then do
   LT  ← $sum_{m:63} < 0$
   GT  ← $sum_{m:63} > 0$
   EQ  ← $sum_{m:63} = 0$
   CR0 ← LT || GT || EQ || SO
rD ← sum
CA      ← $carry_m$
```

The sum of the contents of **r**A and **r**B is placed into **r**D.

Other registers altered:

- CA
  CR0 (if Rc=1)
  SO OV (if OE=1)

adde

Base	User

adde

Add Extended

adde	**r**D,**r**A,**r**B	(OE=0, Rc=0)
adde.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
addeo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
addeo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)

0					5	6 10	11 15	16 20	21	22								30	31
0	1	1	1	1	1	**r**D	**r**A	**r**B	OE	0	1	0	0	0	1	0	1	0	Rc

```
$carry_{0:63}$ ← Carry((rA) + (rB) + CA)
$sum_{0:63}$   ←       (rA) + (rB) + CA
if OE=1 then do
   OV   ← $carry_m \oplus carry_{m+1}$
   SO   ← SO | ($carry_m \oplus carry_{m+1}$)
if Rc=1 then do
   LT  ← $sum_{m:63}$ < 0
   GT  ← $sum_{m:63}$ > 0
   EQ  ← $sum_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rD ← sum
CA      ← $carry_m$
```

The sum of the contents of **r**A, the contents of **r**B, and CA is placed into **r**D.

Other registers altered:

- CA
  CR0 (if Rc=1)
  SO OV (if OE=1)

addi

Base	User

addi

Add Immediate

addi **r**D,**r**A,SIMM

0 5	6 10	11 15	16 31
0 0 1 1 1 0	**r**D	**r**A	SIMM

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
rD ← a + EXTS(SIMM)
```

If **r**A=0, the sign-extended value of the SIMM field is placed into **r**D.

If **r**A≠0, the sum of the contents of **r**A and the sign-extended value of field SIMM is placed into **r**D.

Other registers altered: None

Simplified mnemonics: See Section C.2, “Subtract Simplified Mnemonics,” Section C.10.2, “Load Immediate (li),” and Section C.10.3, “Load Address (la).”

addic

Base	User

addic

Add Immediate Carrying

addic **r**D,**r**A,SIMM (Rc=0)
addic. **r**D,**r**A,SIMM (Rc=1)

0 4	5	6 10	11 15	16 31
0 0 1 1 0	Rc	**r**D	**r**A	SIMM

```
carry$_{0:63}$ ← Carry((rA) + EXTS(SIMM))
sum$_{0:63}$   ←       (rA) + EXTS(SIMM)
if Rc=1 then do
   LT  ← sum$_{m:63}$ < 0
   GT  ← sum$_{m:63}$ > 0
   EQ  ← sum$_{m:63}$ = 0
   CR0  ← LT || GT || EQ || SO
rD ← sum
CA ← carry$_{m}$
```

The sum of the contents of **r**A and the sign-extended value of the SIMM field is placed into **r**D.

Other registers altered:

- CA
  CR0 (if Rc=1)

Simplified mnemonics: See Section C.2, “Subtract Simplified Mnemonics.”

addis

Base	User

addis

Add Immediate Shifted

addis **r**D,**r**A,SIMM

0					5	6 10	11 15	16 31
0	0	1	1	1	1	**r**D	**r**A	SIMM

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
rD ← a + EXTS(SIMM || $^{16}0$)
```

If **r**A=0, the sign-extended value of the SIMM field, concatenated with 16 zeros, is placed into **r**D.

If **r**A≠0, the sum of the contents of **r**A and the sign-extended value of the SIMM field concatenated with 16 zeros, is placed into **r**D.

Other registers altered: None

Simplified mnemonics: See Section C.2, “Subtract Simplified Mnemonics,” and Section C.10.2, “Load Immediate (li).”

addme

Base	User

addme

Add to Minus One Extended

addme	**r**D,**r**A	(OE=0, Rc=0)
addme.	**r**D,**r**A	(OE=0, Rc=1)
addmeo	**r**D,**r**A	(OE=1, Rc=0)
addmeo.	**r**D,**r**A	(OE=1, Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	///	OE	0 1 1 1 0 1 0 1 0	Rc

```
carry$_{0:63}$ ← Carry((rA) + CA + 0xFFFF_FFFF_FFFF_FFFF)
sum$_{0:63}$   ←       (rA) + CA + 0xFFFF_FFFF_FFFF_FFFF
if OE=1 then do
   OV   ← carry$_m$ ⊕ carry$_{m+1}$
   SO   ← SO | (carry$_m$ ⊕ carry$_{m+1}$)
if Rc=1 then do
   LT  ← sum$_{m:63}$ < 0
   GT  ← sum$_{m:63}$ > 0
   EQ  ← sum$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rD ← sum
CA ← carry$_m$
```

The sum of the contents of **r**A, CA, and $^{64}1$ is placed into **r**D.

Other registers altered:

- CA
  CR0 (if Rc=1)
  SO OV (if OE=1)

addze

Base	User

Add to Zero Extended

addze	**r**D,**r**A	(OE=0, Rc=0)
addze.	**r**D,**r**A	(OE=0, Rc=1)
addzeo	**r**D,**r**A	(OE=1, Rc=0)
addzeo.	**r**D,**r**A	(OE=1, Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	///	OE	0 1 1 0 0 1 0 1 0	Rc

```
carry$_{0:63}$ ← Carry((rA) + CA)
sum$_{0:63}$   ←       (rA) + CA
if OE=1 then do
   OV   ← carry$_m$ ⊕ carry$_{m+1}$
   SO   ← SO | (carry$_m$ ⊕ carry$_{m+1}$)
if Rc=1 then do
   LT  ← sum$_{m:63}$ < 0
   GT  ← sum$_{m:63}$ > 0
   EQ  ← sum$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rD ← sum
CA ← carry$_{32}$
```

The sum of the contents of **r**A and CA is placed into **r**D.

Other registers altered:

- CA
  CR0 (if Rc=1)
  SO OV (if OE=1)

and

Base	User

and

AND

and **r**A,**r**S,**r**B (Rc=0)
and. **r**A,**r**S,**r**B (Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 0 0 0 1 1 1 0 0	Rc

```
result$_{0:63}$ ← (rS) & (rB)
rA ← result
if Rc=1 then do
   LT  ← result$_{m:63}$ < 0
   GT  ← result$_{m:63}$ > 0
   EQ  ← result$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
```

The contents of **r**S are ANDed with the contents of **r**B and the result is placed in **r**A.

Other registers altered: CR0 (if Rc=1)

andc andc

Base	User

AND with Complement

andc	**r**A,**r**S,**r**B	(Rc=0)
andc.	**r**A,**r**S,**r**B	(Rc=1)

0					5	6 10	11 15	16 20	21									30	31
0	1	1	1	1	1	**r**S	**r**A	**r**B	0	0	0	0	1	1	1	1	0	0	Rc

```
result$_{0:63}$ ← (rS) & ¬(rB)
rA ← result
if Rc=1 then do
   LT  ← result$_{m:63}$ < 0
   GT  ← result$_{m:63}$ > 0
   EQ  ← result$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
```

The contents of **r**S are ANDed with the one’s complement of the contents of **r**B and the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

andi. andi.

Base	User

AND Immediate

andi. **r**A,**r**S,UIMM

0 5	6 10	11 15	16 31
0 1 1 1 0 0	**r**S	**r**A	UIMM

```
$\text{result}_{0:63}$ ← (rS) & $^{48}$0 || UIMM
rA ← result
LT  ← $\text{result}_{m:63}$ < 0
GT  ← $\text{result}_{m:63}$ > 0
EQ  ← $\text{result}_{m:63}$ = 0
CR0 ← LT || GT || EQ || SO
```

The contents of **r**S are ANDed with ^{48}0 || UIMM and the result is placed into **r**A.

Other registers altered: CR0

andis. andis.

Base	User

AND Immediate Shifted

andis. **r**A,**r**S,UIMM

0 ... 5	6 ... 10	11 ... 15	16 ... 31
0 1 1 1 0 1	**r**S	**r**A	UIMM

```
result$_{0:63}$ ← (rS) & $^{32}$0 || UIMM || $^{16}$0
rA ← result
LT  ← result$_{m:63}$ < 0
GT  ← result$_{m:63}$ > 0
EQ  ← result$_{m:63}$ = 0
CR0 ← LT || GT || EQ || SO
```

The contents of **r**S are ANDed with ^{32}0 || UIMM || ^{16}0 and the result is placed into **r**A.

Other registers altered: CR0

b

Base	User

b

Branch [and Link] [Absolute]

b	LI	(AA=0, LK=0)
ba	LI	(AA=1, LK=0)
bl	LI	(AA=0, LK=1)
bla	LI	(AA=1, LK=1)

0 5	6 29	30	31
0 1 0 0 1 0	LI	AA	LK

```
if AA=1 then a ← $^{64}0$ else a ← CIA
NIA ← $^{m}0$ || (a + EXTS(LI||0b00))$_{m:63}$
if LK=1 then LR ← CIA + 4
```

The branch target address is the address of the next instruction to be executed.

If AA=0 then the branch target address is the sum of LI || 0b00 sign extended and the address of this instruction. If AA=1 then the branch target address is LI || 0b00 sign extended.

In 32-bit mode, bits 0:31 of NIA are set to zero

If LK=1, the sum CIA+4 is placed into the LR.

Other registers altered: LR (if LK=1)

Simplified mnemonics: See Section C.4, “Branch Instruction Simplified Mnemonics.”

bc

Base	User

bc

Branch Conditional [and Link] [Absolute]

bc	BO,BI,BD	(AA=0, LK=0)
bca	BO,BI,BD	(AA=1, LK=0)
bcl	BO,BI,BD	(AA=0, LK=1)
bcla	BO,BI,BD	(AA=1, LK=1)

0 5	6 10	11 15	16 29	30	31
0 1 0 0 0 0	BO	BI	BD	AA	LK

```
if ¬BO$_2$ then CTR ← CTR - 1
ctr_ok  ← BO$_2$ | ((CTR$_{m:63}$ ≠ 0) ⊕ BO$_3$)
cond_ok ← BO$_0$ | (CR$_{BI+32}$ ≡ BO$_1$)
if ctr_ok & cond_ok then
   if AA=1 then a ← $^{64}$0 else a ← CIA
   NIA ← $^{m}$0 || (a + EXTS(BD||0b00))$_{m:63}$
else            NIA ← $^{m}$0 || (CIA + 4)$_{m:63}$
if LK=1 then LR ← CIA + 4
```

The branch target address is the address of the next instruction to be executed if the branch is to be taken.

BI+32 specifies the CR bit to be tested. The BO instruction field specifies any conditions that must be met for the branch to be taken, as defined in Section 4.6.1.9, “Conditional Branch Control.” Depending on mode, all 64 bits or the lower 32 bits of CTR are used to determine whether the CTR is non-zero.

If AA=0 and the branch is to be taken then the branch target address is the sum of BD || 0b00 sign extended and the address of this instruction. If AA=1 and the branch is to be taken then the branch target address is BD || 0b00 sign extended.

In 32-bit mode, bits 0:31 of NIA are set to zero if the branch is taken.

If the branch is not taken, the next sequential instruction is the address of the next instruction to be executed.

If LK=1, the sum CIA+4 is placed into the LR.

The BI field specifies the CR bit used as the condition of the branch, as shown in Table 5-5.

Table 5-5. BI Operand Settings for CR Fields

CR*n* Bits	CR Bits	BI	Description
CR0[0]	32	00000	Negative (LT)—Set when the result is negative.
CR0[1]	33	00001	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	34	00010	Zero (EQ)—Set when the result is zero.
CR0[3]	35	00011	Summary overflow (SO). Copy of XER[SO] at the instruction’s completion.
CR1[0]	36	00100	Copy of FPSCR[FX] at the instruction’s completion.
CR1[1]	37	00101	Copy of FPSCR[FEX] at the instruction’s completion.

Table 5-5. BI Operand Settings for CR Fields (continued)

CR*n* Bits	CR Bits	BI	Description
CR1[2]	38	00110	Copy of FPSCR[VX] at the instruction's completion.
CR1[3]	39	00111	Copy of FPSCR[OX] at the instruction's completion.
CR*n*[0]	40 44 48 52 56 60	01000 01100 10000 10100 11000 11100	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A < **fr**B.
CR*n*[1]	41 45 49 53 57 61	01001 01101 10001 10101 11001 11101	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions:**fr**A > **fr**B.
CR*n*[2]	42 46 50 54 58 62	01010 01110 10010 10110 11010 11110	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
CR*n*[3]	43 47 51 55 59 63	01011 01111 10011 10111 11011 11111	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN.

Other registers altered:

- CTR(if BO_2=0)
  LR(if LK=1)

Simplified mnemonics: See Section C.4, "Branch Instruction Simplified Mnemonics."

bcctr

Base	User

bcctr

Branch Conditional to Count Register [and Link]

bcctr BO,BI,BH (LK=0)
bcctrl BO,BI,BH (LK=1)

0 … 5	6 … 10	11 … 15	16 … 18	19 20	21 … 30	31
0 1 0 0 1 1	BO	BI	///	BH	1 0 0 0 0 1 0 0 0 0	LK

```
cond_ok ← $BO_0$ | ($CR_{BI+32}$ ≡ $BO_1$)
if  cond_ok then NIA ← $^{m}0$ || $CTR_{m:61}$ || 0b00
if ¬cond_ok then NIA ← $^{m}0$ || $(CIA + 4)_{m:63}$
if LK=1 then LR ← CIA + 4
```

The branch target address is the address of the next instruction to be executed if the branch is to be taken.

BI+32 specifies the CR bit to be tested. The BO instruction field specifies any conditions that must be met for the branch to be taken, as defined in Section 4.6.1.9, “Conditional Branch Control.” The BH field is used as a static prediction of how the branch is used as defined in Table 4-28. If the branch is to be taken then the branch target address is $CTR_{0:61}$ || 0b00 with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

In 32-bit mode, bits 0:31 of NIA are set to zero if the branch is taken.

If the branch is not taken, the next sequential instruction is the address of the next instruction to be executed.

If LK=1, the sum CIA+4 is placed into the LR.

If the decrement and test CTR option is specified (BO_2=0), the instruction form is invalid.

BO specifies conditions that must be met for the branch to be taken. BI+32 specifies the CR bit to be used; see Table 5-6.

Table 5-6. BI Operand Settings for CR Fields

CR*n* Bits	CR Bits	BI	Description
CR0[0]	32	00000	Negative (LT)—Set when the result is negative.
CR0[1]	33	00001	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	34	00010	Zero (EQ)—Set when the result is zero.
CR0[3]	35	00011	Summary overflow (SO). Copy of XER[SO] at the instruction’s completion.
CR1[0]	36	00100	Copy of FPSCR[FX] at the instruction’s completion.
CR1[1]	37	00101	Copy of FPSCR[FEX] at the instruction’s completion.
CR1[2]	38	00110	Copy of FPSCR[VX] at the instruction’s completion.
CR1[3]	39	00111	Copy of FPSCR[OX] at the instruction’s completion.

Table 5-6. BI Operand Settings for CR Fields (continued)

CR*n* Bits	CR Bits	BI	Description
CR*n*[0]	40 44 48 52 56 60	01000 01100 10000 10100 11000 11100	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions:**fr**A < **fr**B.
CR*n*[1]	41 45 49 53 57 61	01001 01101 10001 10101 11001 11101	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A > **fr**B.
CR*n*[2]	42 46 50 54 58 62	01010 01110 10010 10110 11010 11110	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
CR*n*[3]	43 47 51 55 59 63	01011 01111 10011 10111 11011 11111	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN.

Other registers altered: LR (if LK=1)

Simplified mnemonics: See Section C.4, “Branch Instruction Simplified Mnemonics.”

bclr

Base	User

bclr

Branch Conditional to Link Register [and Link]

bclr	BO,BI,BH	(LK=0)
bclrl	BO,BI,BH	(LK=1)

0 … 5	6 … 10	11 … 15	16 … 18	19 … 20	21 … 30	31
0 1 0 0 1 1	BO	BI	///	BH	0 0 0 0 0 1 0 0 0 0	LK

```
if $\neg BO_2$ then CTR ← CTR - 1
ctr_ok  ← $BO_2$ | (($CTR_{m:63}$ ≠ 0) ⊕ $BO_3$)
cond_ok ← $BO_0$ | ($CR_{BI+32}$ ≡ $BO_1$)
if ctr_ok & cond_ok then
   NIA ← $^{m}0$ || ($LR_{m:61}$||0b00))
else            NIA ← $^{m}0$ || $(CIA + 4)_{m:63}$
if LK=1 then LR ← CIA + 4
```

The branch target address is the address of the next instruction to be executed if the branch is to be taken.

BI+32 specifies the CR bit to be tested. The BO instruction field specifies any conditions that must be met for the branch to be taken, as defined in Section 4.6.1.9, “Conditional Branch Control.” The BH field is used as a static prediction of how the branch is used as defined in Table 4-28. Depending on mode, all 64 bits or the lower 32 bits of CTR are used to determine whether the CTR is non-zero. If the branch is to be taken then the branch target address is $LR_{0:61}$ || 0b00 with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

In 32-bit mode, bits 0:31 of NIA are set to zero if the branch is taken.

If the branch is not taken, the next sequential instruction is the address of the next instruction to be executed.

If LK=1, the sum CIA+4 is placed into the LR.

The BI field specifies the CR bit used as the condition of the branch, as shown in Table 5-7.

Table 5-7. BI Operand Settings for CR Fields

CR*n* Bits	CR Bits	BI	Description
CR0[0]	32	00000	Negative (LT)—Set when the result is negative.
CR0[1]	33	00001	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	34	00010	Zero (EQ)—Set when the result is zero.
CR0[3]	35	00011	Summary overflow (SO). Copy of XER[SO] at the instruction’s completion.
CR1[0]	36	00100	Copy of FPSCR[FX] at the instruction’s completion.
CR1[1]	37	00101	Copy of FPSCR[FEX] at the instruction’s completion.
CR1[2]	38	00110	Copy of FPSCR[VX] at the instruction’s completion.
CR1[3]	39	00111	Copy of FPSCR[OX] at the instruction’s completion.

Table 5-7. BI Operand Settings for CR Fields (continued)

CR*n* Bits	CR Bits	BI	Description
CR*n*[0]	40 44 48 52 56 60	01000 01100 10000 10100 11000 11100	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A < **fr**B.
CR*n*[1]	41 45 49 53 57 61	01001 01101 10001 10101 11001 11101	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A > **fr**B.
CR*n*[2]	42 46 50 54 58 62	01010 01110 10010 10110 11010 11110	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
CR*n*[3]	43 47 51 55 59 63	01011 01111 10011 10111 11011 11111	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN.

Other registers altered:

- CTR (if BO_2=0)
  LR (if LK=1)

Simplified mnemonics: See Section C.4, “Branch Instruction Simplified Mnemonics.”

bpermd

64	User

bpermd

Bit Permute Doubleword

bpermd **r**A,**r**S,**r**B

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1 1 1 1 1 1 0 0	/

```
for i = 0 to 7 do
    index ← $(rS)_{8*i:8*i+7}$
    if index < 64 then $perm_i$ ← $(rB)_{index}$
    else               $perm_i$ ← 0
rA ← $^{56}0$ || $perm_{0:7}$
```

Eight permuted bits are produced. For each permuted bit i where i ranges from 0 to 7 and for each byte i of **r**S, do the following.

> If byte i if **r**S is less than 64, permuted bit i is set to the bit of **r**B specified by byte i of **r**S; otherwise permuted bit i is set to 0.

The permuted bits are placed in the least significant byte of **r**A, and the remaining bits are set to 0.

Other registers altered: None

NOTE: Software Considerations

The permuted bit is 0 if the corresponding index value exceeds 63 permits the permuted bits to be selected from a 128-bit quantity, using a single index register. For example, assume that the 128-bit quantity Q, from which the permuted bits are to be selected, is in registers r2 (high-order 64 bits of Q) and r3 (low-order 64 bits of Q), that the index values are in register r1, with each byte of r1 containing a value in the range 0:127, and that each byte of register r4 contains the value 64. The following code sequence selects eight permuted bits from Q and places them into the low-order byte of r6.

```
bpermd  r6,r1,r2 # select from high-order half of Q
xor     r0,r1,r4 # adjust index values
bpermd  r5,r0,r3 # select from low-order half of Q
or      r6,r6,r5 # merge selections
```

cmp

Base	User

Compare

cmp **crfD**,L,**r**A,**r**B

0 5	6 8	9	10	11 15	16 20	21 30	31
0 1 1 1 1 1	crfD	/	L	rA	rB	0 0 0 0 0 0 0 0 0 0	/

```
if L=0 then a ← EXTS($(rA)_{32:63}$)
else        a ← (rA)
if L=0 then b ← EXTS($(rB)_{32:63}$)
else        b ← (rB)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
$CR_{4\times\mathbf{crfD}+32:4\times\mathbf{crfD}+35}$ ← c || SO
```

If L=0, the contents of **r**$A_{32:63}$ are compared with the contents of **r**$B_{32:63}$, treating the operands as signed integers.

<64>:
If L=1, the contents of **r**A are compared with the contents of **r**B, treating the operands as signed integers.

The result of the comparison is placed into CR field **crfD**.

For 32-bit implementations, the form with L=1 is considered invalid.

Other registers altered: CR field **crfD**

Simplified mnemonics: See Section C.5, “Compare Word Simplified Mnemonics” and Section C.6, “Compare Double-word Simplified Mnemonics.”

cmpb cmpb

Base	User

Compare Bytes

cmpb **r**A,**r**S,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 1 1 1 1 1 1 1 0 0	/

```
for i = 0 to 7 do
   if $(rS)_{8*i:8*i+7}$ = $(rB)_{8*i:8*i+7}$ then
       $rA_{8*i:8*i+7}$ ← 0xFF
   else
       $rA_{*8i:8*i+7}$ ← 0x00
```

Each byte in **r**S is compared to each corresponding byte of **r**B. If they are equal, the corresponding byte in **r**A is set to 0xFF. Otherwise the corresponding byte is set to 0x00.

Other registers altered: None

Simplified mnemonics: See Section C.5, “Compare Word Simplified Mnemonics.”

cmpi

Base	User

cmpi

Compare Immediate

cmpi **crf**D,L,**r**A,SIMM

0 5	6 8	9	10	11 15	16 31
0 0 1 0 1 1	**crf**D	/	L	**r**A	SIMM

```
if L=0 then a ← EXTS((rA)$_{32:63}$)
else        a ← (rA)
b ← EXTS(SIMM)
if a < b then c ← 0b100
if a > b then c ← 0b010
if a = b then c ← 0b001
CR$_{4\times\mathbf{crfD}+32:4\times\mathbf{crfD}+35}$ ← c || SO
```

If L=0, the contents of **r**A$_{32:63}$ are compared with the sign-extended value of the SIMM field, treating the operands as signed integers.

<64>:
If L=1, the contents of **r**A are compared with the sign-extended value of the SIMM field, treating the operands as signed integers.

The result of the comparison is placed into CR field **crfD**.

For 32-bit implementations, the form with L=1 is considered invalid.

Other registers altered: CR field **crfD**

Simplified mnemonics: See Section C.5, “Compare Word Simplified Mnemonics” and Section C.6, “Compare Double-word Simplified Mnemonics.”

cmpl

Base	User

cmpl

Compare Logical

cmpl **crfD,**L**,r**A**,r**B

0 5	6 8	9	10	11 15	16 20	21 30	31
0 1 1 1 1 1	**crfD**	/	L	**rA**	**rB**	0 0 0 0 1 0 0 0 0 0	/

```
if L=0 then a ← $^{32}0$ || (rA)$_{32:63}$
else        a ← (rA)
if L=0 then b ← $^{32}0$ || (rB)$_{32:63}$
else        b ← (rB)
if a <$_u$ b then c ← 0b100
if a >$_u$ b then c ← 0b010
if a = b then c ← 0b001
CR$_{4\times\mathbf{crfD}+32:4\times\mathbf{crfD}+35}$ ← c || SO
```

If L=0, the contents of **r**A$_{32:63}$ are compared with the contents of **r**B$_{32:63}$, treating the operands as unsigned integers.

<64>:
If L=1, the contents of **r**A are compared with the contents of **r**B, treating the operands as unsigned integers.

The result of the comparison is placed into CR field **crfD**.

For 32-bit implementations, the form with L=1 is considered invalid.

Other registers altered: CR field **crfD**

Simplified mnemonics: See Section C.5, “Compare Word Simplified Mnemonics” and Section C.6, “Compare Double-word Simplified Mnemonics.”

cmpli

Base	User

cmpli

Compare Logical Immediate

cmpli **crfD,**L,**r**A,UIMM

0 5	6 8	9	10	11 15	16 31
0 0 1 0 1 0	**crfD**	/	L	**r**A	UIMM

```
if L=0 then a ← $^{32}$0 || (rA)$_{32:63}$
else        a ← (rA)
b ← $^{48}$0 || UIMM
if a <$_u$ b then c ← 0b100
if a >$_u$ b then c ← 0b010
if a = b then c ← 0b001
CR$_{4\times\mathbf{crfD}+32:4\times\mathbf{crfD}+35}$ ← c || SO
```

If L=0, the contents of **r**A$_{32:63}$ are compared with the zero-extended value of the UIMM field, treating the operands as unsigned integers.

<64>:
If L=1, the contents of **r**A are compared with the zero-extended value of the UIMM field, treating the operands as unsigned integers.

The result of the comparison is placed into CR field **crfD**.

For 32-bit implementations, the form with L=1 is considered invalid.

Other registers altered: CR field **crfD**

Simplified mnemonics: See Section C.5, “Compare Word Simplified Mnemonics.”

cntlzd cntlzd

64	User

Count Leading Zeros Doubleword

cntlzd **r**A,**r**S (Rc=0)
cntlzd. **r**A,**r**S (Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	///	0 0 0 0 1 1 1 0 1 0	Rc

```
n ← 0
while n < 64 do
   if $(rS)_n$ = 1 then leave
   n ← n + 1
rA ← n
if Rc=1 then do
    LT  ← n < 0
    GT  ← n > 0
    EQ  ← n = 0
    CR0 ← LT || GT || EQ || SO
```

A count of the number of consecutive zero bits starting at **r**S_0 is placed into **r**A. This number ranges from 0 to 64, inclusive.

Other registers altered: CR0 (if Rc=1)

cntlzw

Base	User

cntlzw

Count Leading Zeros Word

cntlzw	**r**A,**r**S	(Rc=0)
cntlzw.	**r**A,**r**S	(Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	///	0 0 0 0 0 1 1 0 1 0	Rc

```
n ← 32
while n < 64 do
   if $(rS)_n$ = 1 then leave
   n ← n + 1
rA ← n - 32
if Rc=1 then do
    LT  ← (rA) < 0
    GT  ← (rA) > 0
    EQ  ← (rA) = 0
    CR0 ← LT || GT || EQ || SO
```

A count of the number of consecutive zero bits starting at **r**S_{32} is placed into **r**A. This number ranges from 0 to 32, inclusive.

Other registers altered: CR0 (if Rc=1)

crand

Base	User

crand

Condition Register AND

crand **crb**D,**crb**A,**crb**B

0 5	6 10	11 15	16 20	21 30	31
0 1 0 0 1 1	**crb**D	**crb**A	**crb**B	0 1 0 0 0 0 0 0 0 1	/

$CR_{crbD+32} \leftarrow CR_{crbA+32}\ \&\ CR_{crbB+32}$

The content of bit **crb**A+32 of CR is ANDed with the content of bit **crb**B+32 of CR, and the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

crandc

Base	User

crandc

Condition Register AND with Complement

crandc **crb**D,**crb**A,**crb**B

0 5	6 10	11 15	16 20	21 30	31
0 1 0 0 1 1	**crb**D	**crb**A	**crb**B	0 0 1 0 0 0 0 0 0 1	/

$$\mathtt{CR}_{\mathtt{crbD+32}} \leftarrow \mathtt{CR}_{\mathtt{crbA+32}} \mathbin{\&} \neg\mathtt{CR}_{\mathtt{crbB+32}}$$

The content of bit **crb**A+32 of CR is ANDed with the one’s complement of the content of bit **crb**B+32 of CR, and the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

creqv creqv

Base	User

Condition Register Equivalent

creqv **crb**D,**crb**A,**crb**B

0 5	6 10	11 15	16 20	21 30	31
0 1 0 0 1 1	**crb**D	**crb**A	**crb**B	0 1 0 0 1 0 0 0 0 1	/

$$\mathtt{CR}_{\mathtt{crbD+32}} \leftarrow \mathtt{CR}_{\mathtt{crbA+32}} \equiv \mathtt{CR}_{\mathtt{crbB+32}}$$

The content of bit **crb**A+32 of CR is XORed with the content of bit **crb**B+32 of CR, and the one’s complement of result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

Simplified mnemonics: See Section C.7, “Condition Register Logical Simplified Mnemonics.”

crnand crnand

Base	User

Condition Register NAND

crnand **crb**D,**crb**A,**crb**B

0 5	6 10	11 15	16 20	21 30	31
0 1 0 0 1 1	**crb**D	**crb**A	**crb**B	0 0 1 1 1 0 0 0 0 1	/

$CR_{crbD+32} \leftarrow \neg(CR_{crbA+32} \ \& \ CR_{crbB+32})$

The content of bit **crb**A+32 of CR is ANDed with the content of bit **crb**B+32 of CR, and the one’s complement of the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

crnor crnor

Base	User

Condition Register NOR

crnor **crb**D,**crb**A,**crb**B

0					5	6 10	11 15	16 20	21									30	31
0	1	0	0	1	1	**crb**D	**crb**A	**crb**B	0	0	0	0	1	0	0	0	0	1	/

$\mathrm{CR}_{\mathrm{crbD+32}} \leftarrow \neg(\mathrm{CR}_{\mathrm{crbA+32}} \mid \mathrm{CR}_{\mathrm{crbB+32}})$

The content of bit **crb**A+32 of CR is ORed with the content of bit **crb**B+32 of CR, and the one’s complement of the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

Simplified mnemonics: See Section C.7, “Condition Register Logical Simplified Mnemonics.”

cror

Base	User

Condition Register OR

cror **crb**D,**crb**A,**crb**B

0 5	6 10	11 15	16 20	21 30	31
0 1 0 0 1 1	**crb**D	**crb**A	**crb**B	0 1 1 1 0 0 0 0 0 1	/

$$CR_{crbD+32} \leftarrow CR_{crbA+32} \mid CR_{crbB+32}$$

The content of bit **crb**A+32 of CR is ORed with the content of bit **crb**B+32 of CR, and the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

Simplified mnemonics: See Section C.7, “Condition Register Logical Simplified Mnemonics.”

crorc

Base	User

crorc

Condition Register OR with Complement

crorc **crb**D,**crb**A,**crb**B

0 … 5	6 … 10	11 … 15	16 … 20	21 … 30	31
0 1 0 0 1 1	**crb**D	**crb**A	**crb**B	0 1 1 0 1 0 0 0 0 1	/

```
$CR_{crbD+32} \leftarrow CR_{crbA+32} \mid \neg CR_{crbB+32}$
```

The content of bit **crb**A+32 of CR is ORed with the one’s complement of the content of bit **crb**B+32 of CR, and the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

crxor

Base	User

crxor

Condition Register XOR

crxor **crb**D,**crb**A,**crb**B

0 5	6 10	11 15	16 20	21 30	31
0 1 0 0 1 1	**crb**D	**crb**A	**crb**B	0 0 1 1 0 0 0 0 0 1	/

$$\mathtt{CR}_{\mathtt{crbD+32}} \leftarrow \mathtt{CR}_{\mathtt{crbA+32}} \oplus \mathtt{CR}_{\mathtt{crbB+32}}$$

The content of bit **crb**A+32 of CR is XORed with the content of bit **crb**B+32 of CR, and the result is placed into bit **crb**D+32 of CR.

Other registers altered: CR

Simplified mnemonics: See Section C.7, “Condition Register Logical Simplified Mnemonics.”

dcba

Base	User

Data Cache Block Allocate

dcba **r**A,**r**B

0 5	6 9	10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	0	rA	rB	1 0 1 1 1 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
if L1CSR0$_{DCBZ32}$ then sz ← 32
              else sz ← CacheBlockSize()
AllocateDataCacheBlock(EA, sz)
```

dcba is a hint that performance would likely improve if the block containing the byte addressed by EA is established in the data cache without fetching the block from main memory, because the program is likely to soon store into a portion of the block and the contents of the rest of the block are not meaningful to the program. If the hint is honored, the contents of the block are undefined when the instruction completes. The hint is ignored if the block is caching-inhibited.

If L1CSR0[DCBZ32] = 1, **dcba** operates as if the cache block size is 32 bytes instead of the entire cache block. If the actual cache block size is larger than 32 bytes, then the remaining bytes in the cache block are architecturally unaffected by **dcba**.

If the block containing the byte addressed by EA is in memory that is memory-coherence required and the block exists in a data cache of any other processors, it is kept coherent in those caches.

This instruction is treated as a store for translation, permissions, and debug events, except that an interrupt is not taken for a translation or protection violation. See Section 6.5.6.4, “Permission Control and Cache Management Instructions,” and Section 4.6.1.17, “Cache Management Instructions.”

This instruction may establish a block in the data cache without verifying that the associated real address is valid. This can cause a delayed machine check interrupt, as described in Section 7.8.2, “Machine Check, NMI, and Error Report Interrupts.”

Other registers altered: None

NOTE: Software Considerations

Software should only use L1CSR0$_{DCBZ32}$ to perform 32 byte cache block allocations to preserve software compatibility for software written that mistakenly assumed that a cache line was always 32 bytes. Operations using dcba with L1CSR0$_{DCBZ32}$ = 1 are likely to perform slower than providing no hint.

dcbal

DEO	User

dcbal

Data Cache Block Allocate Line

dcbal **r**A,**r**B

0 5	6 9	10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	1	rA	rB	1 0 1 1 1 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
sz ← CacheBlockSize()
AllocateDataCacheBlock(EA, sz)
```

dcbal is a hint that performance would likely improve if the block containing the byte addressed by EA is established in the data cache without fetching the block from main memory, because the program is likely to soon store into a portion of the block and the contents of the rest of the block are not meaningful to the program. If the hint is honored, the contents of the block are undefined when the instruction completes. The hint is ignored if the block is caching-inhibited.

If the block containing the byte addressed by EA is in memory that is memory-coherence required and the block exists in a data cache of any other processors, it is kept coherent in those caches.

This instruction is treated as a store for translation, permissions, and debug events, except that an interrupt is not taken for a translation or protection violation. See Section 6.5.6.4, "Permission Control and Cache Management Instructions," and Section 4.6.1.17, "Cache Management Instructions."

This instruction may establish a block in the data cache without verifying that the associated real address is valid. This can cause a delayed machine check interrupt, as described in Section 7.8.2, "Machine Check, NMI, and Error Report Interrupts."

This instruction behaves the same as **dcba** with the exception that it is unaffected by the setting of $L1CSR0_{DCBZ32}$.

Other registers altered: None

dcbf

Base	User

dcbf

Data Cache Block Flush

dcbf **r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	rA	rB	0 0 0 1 0 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
FlushDataCacheBlock( EA )
```

If the block containing the byte addressed by EA is in memory that is memory-coherence required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, then those locations are written to main memory. Additional locations in the block may also be written to main memory. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is in memory that is not memory-coherence required and the block containing the byte addressed by EA is in the data cache of this processor and any locations in the block are considered to be modified there, then those locations are written to main memory. Additional locations in the block may also be written to main memory. The block is invalidated in the data cache of this processor.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in memory that is write-through required or caching-inhibited.

This instruction is treated as a load for translation and permissions, and is treated as store for debug events. See Section 6.5.6.4, “Permission Control and Cache Management Instructions,” and Section 4.6.1.17, “Cache Management Instructions.”

Other registers altered: None

NOTE: Software Considerations

Some processors may only apply this instruction to local caches and may have optional features in order to cause the operation to be broadcast to other processors. See the core reference manual.

dcbfep

E.PD	Supervisor

Data Cache Block Flush by External PID

dcbfep **r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	**r**A	**r**B	0 0 0 1 1 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
FlushDataCacheBlock( EA )
```

If the block containing the byte addressed by EA is in memory that is memory-coherence required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, then those locations are written to main memory. Additional locations in the block may also be written to main memory. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is in memory that is not memory-coherence required and the block containing the byte addressed by EA is in the data cache of this processor and any locations in the block are considered to be modified there, then those locations are written to main memory. Additional locations in the block may also be written to main memory. The block is invalidated in the data cache of this processor.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in memory that is write-through required or caching-inhibited.

This instruction is treated as a load for translation and permissions, and is treated as store for debug events. See Section 6.5.6.4, “Permission Control and Cache Management Instructions,” and Section 4.6.1.17, “Cache Management Instructions.”

This instruction is guest supervisor privileged.

For **dcbfep**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID] (GESR[EPID] <E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

Note: **dcbfep** is identical to **dcbf** except for using EPLC for translation context.

NOTE: Software Considerations

Some processors may only apply this instruction to local caches and may have optional features in order to cause the operation to be broadcast to other processors. See the core reference manual.

dcbi dcbi

Embedded	Supervisor

Data Cache Block Invalidate

dcbi **r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	**r**A	**r**B	0 1 1 1 0 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
InvalidateDataCacheBlock( EA )
```

If the block containing the byte addressed by EA is in memory that is memory-coherence required and a block containing the byte addressed by EA is in the data cache of any processor and any locations in the block are considered to be modified there, then those locations are written to main memory. Additional locations in the block may also be written to main memory. The block is invalidated in the data caches of all processors.

If the block containing the byte addressed by EA is not coherence-required memory and a block containing the byte addressed by EA is in the data cache of this processor, then the block is invalidated in that data cache. On some implementations, before the block is invalidated, any locations in the block considered modified in that data cache are written to main memory; additional locations in the block may be written to main memory.This instruction is treated as a load for translation and permissions, and is treated as store for debug events

dcbi is treated as a store for translation, permissions, and debug events on implementations that invalidate a block without first writing to main memory all locations in the block that are considered to be modified in the data cache, except that the invalidation is not ordered by **mbar**. On other implementations this instruction may be treated as a load for translation and permissions, and is treated as store for debug events.

If a processor holds a reservation and some other processor executes a **dcbi** to the same reservation granule, whether the reservation is lost is undefined.

Other registers altered: None

dcblc

E.CL	User

dcblc

Data Cache Block Lock Clear

dcblc CT,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	CT	**r**A	**r**B	0 1 1 0 0 0 0 1 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
DataCacheBlockClearLock(CT, EA)
```

The data cache specified by CT has the cache line corresponding to EA unlocked allowing the line to participate in the normal replacement policy.

Cache lock clear instructions remove locks previously set by cache lock set instructions.

Section 4.6.1.17.2, “User-Level Cache Management Instructions,” lists supported CT values. An implementation may use CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

The instruction is treated as a load for translation, permissions, and debug events and can cause DSI and DTLB error interrupts accordingly.

An unable-to-unlock condition is said to occur any of the following conditions exist:

- The target address is marked cache-inhibited, or the storage attributes of the address uses a coherency protocol that does not support locking.
- The target cache is disabled or not present.
- The CT field of the instructions contains a value not supported by the implementation.
- The target address is not in the cache or is present in the cache but is not locked.

If an unable-to-unlock condition occurs, no cache operation is performed.

dcblc can only be executed by user-level software if MSR[UCLE] = 1.

dcblc can only be executed by guest supervisor software if MSRP[UCLEP] = 0. <E.HV>

NOTE: Software Considerations

Most caches contain a method for clearing all locks present in that cache through use of control and status registers. See Section 3.10.1, “L1 Cache Control and Status Register 0 (L1CSR0)” and Section 3.10.2, “L1 Cache Control and Status Register 1 (L1CSR1).”

dcbst

Base	User

dcbst

Data Cache Block Store

dcbst **r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	**r**A	**r**B	0 0 0 0 1 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
StoreDataCacheBlock( EA )
```

If the block containing the byte addressed by EA is in memory that is memory-coherence required and a block containing the byte addressed by EA is in the data cache of any processor, and any locations in the block are considered to be modified there, those locations are written to main memory. Additional locations in the block may be written to main memory. The block ceases to be considered to be modified in that data cache.

If the block containing the byte addressed by EA is in memory that is not memory-coherence required and a block containing the byte addressed by EA is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main memory. Additional locations in the block may be written to main memory. The block ceases to be considered to be modified in that cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in memory that is write-through required or caching-inhibited.

This instruction is treated as a load for translation and permissions, and is treated as store for debug events. See Section 6.5.6.4, “Permission Control and Cache Management Instructions.”

Other registers altered: None

NOTE: Software Considerations

Some processors may only apply this instruction to local caches and may have optional features in order to cause the operation to be broadcast to other processors. See the core reference manual.

dcbstep

E.PD	Supervisor

dcbstep

Data Cache Block Store by External PID

dcbstep **r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	**rA**	**rB**	0 0 0 0 1 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
StoreDataCacheBlock( EA )
```

If the block containing the byte addressed by EA is in memory that is memory-coherence required and a block containing the byte addressed by EA is in the data cache of any processor, and any locations in the block are considered to be modified there, those locations are written to main memory. Additional locations in the block may be written to main memory. The block ceases to be considered to be modified in that data cache.

If the block containing the byte addressed by EA is in memory that is not memory-coherence required and a block containing the byte addressed by EA is in the data cache of this processor and any locations in the block are considered to be modified there, those locations are written to main memory. Additional locations in the block may be written to main memory. The block ceases to be considered to be modified in that cache.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in memory that is write-through required or caching-inhibited.

This instruction is treated as a load for translation and permissions, and is treated as store for debug events. See Section 6.5.6.4, “Permission Control and Cache Management Instructions.”

This instruction is guest supervisor privileged.

For **dcbstep**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID] (GESR[EPID] <E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

Note: **dcbstep** is identical to **dcbst** except for using EPLC for translation context.

NOTE: Software Considerations

Some processors may only apply this instruction to local caches and may have optional features in order to cause the operation to be broadcast to other processors. See the core reference manual.

dcbt dcbt

Base	User

Data Cache Block Touch

dcbt TH**,r**A**,r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	TH	**r**A	**r**B	0 1 0 0 0 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← a + (rB)
PrefetchDataCacheBlock( TH, EA )
```

The **dcbt** instruction provides a hint that performance will probably be improved if the block containing the byte addressed by EA is prefetched into the data cache because the program will soon load from the addressed byte.

An implementation may use TH values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure. TH values are synonymous with CT values, except that TH values only range from 0 to 7. Section 4.6.1.17.2, "User-Level Cache Management Instructions," lists supported TH values.

The hint is ignored and no operation is performed if:

- The block addressed by EA is caching-inhibited or guarded
- The TH value is not supported by the implementation
- The access would cause a DTLB miss or a DSI
- The cache addressed by TH is disabled

The hint may be ignored for other implementation specific reasons. See the core reference manual. It is implementation dependent whether a hint that is ignored will cause debug events associated with the instruction.

This instruction is treated as a load for translation, permissions, and debug events (see Section 6.5.6.4, "Permission Control and Cache Management Instructions"), except that an interrupt is not taken for a translation or protection violation.

Other registers altered: None

dcbtep

E.PD	Supervisor

dcbtep

Data Cache Block Touch by External PID

dcbtep TH,**r**A,**r**B

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	TH	**r**A	**r**B	0 1 0 0 1 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
PrefetchDataCacheBlock( TH, EA )
```

The **dcbtep** instruction provides a hint that performance will probably be improved if the block containing the byte addressed by EA is prefetched into the data cache because the program will soon load from the addressed byte.

An implementation may use TH values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure. TH values are synonymous with CT values, except that TH values only range from 0 to 7. Section 4.6.1.17.2, “User-Level Cache Management Instructions,” lists supported TH values.

The hint is ignored and no operation is performed if:

- The block addressed by EA is caching-inhibited or guarded
- The TH value is not supported by the implementation
- The access would cause a DTLB miss or a DSI
- The cache addressed by TH is disabled

The hint may be ignored for other implementation specific reasons. See the core reference manual. It is implementation dependent whether a hint that is ignored will cause debug events associated with the instruction.

This instruction is treated as a load for translation, permissions, and debug events (see Section 6.5.6.4, “Permission Control and Cache Management Instructions”), except that an interrupt is not taken for a translation or protection violation.

This instruction is guest supervisor privileged.

For **dcbtep**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered: None

Note: **dcbtep** is identical to **dcbt** except for using EPLC for translation context.

dcbtls

E.CL	User

dcbtls

Data Cache Block Touch and Lock Set

dcbtls CT,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	CT	**r**A	**r**B	0 0 1 0 1 0 0 1 1 0	/

```
if rA = 0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
PrefetchDataCacheBlockLockSet(CT, EA)
```

The data cache specified by CT has the cache line corresponding to EA loaded and locked into the cache. If the line already exists in the cache, it is locked without being refetched.

Section 4.6.1.17.2, “User-Level Cache Management Instructions,” lists supported CT values. An implementation may use CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

Cache touch and lock set instructions let software lock cache lines into the cache to provide lower latency for critical cache accesses and more deterministic behavior. Locked lines do not participate in the normal replacement policy when a line must be victimized for replacement.

This instruction is treated as a load for translation, permissions, and debug events (see Section 6.5.6.4, “Permission Control and Cache Management Instructions”), and can cause DSI and DTLB error interrupts accordingly.

An unable to lock condition is said to occur any of the following conditions exist:

- The target address is cache-inhibited, or the storage attributes of the address uses a coherency protocol that does not support locking.
- The target cache is disabled or not present.
- The CT field of the instructions contains a value not supported by the implementation.

If an unable to lock condition occurs, no cache operation is performed and LICSR0[CUL] is set appropriately. Processors that implement **dcblq.** do not set LICSR0[CUL] when an unable to lock condition occurs and require software to query the status of the lock after attempting to lock it. Consult the core reference manual.

An overlocking condition is said to exist if all available ways for a given cache index are already locked and the locking instruction has no other exceptions. If an overlocking condition occurs for **dcbtls** and if the lock was targeted for the primary cache or secondary cache (CT = 0 or CT = 2), L1CSR0[CLO] (L2CSR0[L2LO] for CT = 2) is set. If lock overflow allocate is set (L1CSR0[CLOA] for CT = 0 and (L2CSR0[L2LOA] for CT = 2), the line is loaded and locked replacing and unlocking an implementation dependent line in the set. Some processors do not set lock overflow status in either L1CSR0 or L2CSR0. Consult the core reference manual.

The results of overlocking and unable to lock conditions for caches other than the primary cache and secondary cache are defined as part of the integrated device.

dcbtls can only be executed by user-level software if MSR[UCLE] = 1.

dcbtls can only be executed by guest supervisor software if MSRP[UCLEP] = 0. <E.HV>

Other registers altered:

- L1CSR0[CUL] if unable to lock occurs
- L1CSR0[CLO] (L2CSR0[L2LO]) if lock overflow occurs

NOTE: Software Considerations

Locks on cache lines can be lost for a variety of reasons. See Section 6.3.1, "Cache Line Locking <E.CL>."

dcbtst dcbtst

Base	User

Data Cache Block Touch for Store

dcbtst TH,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	TH	**r**A	**r**B	0 0 1 1 1 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
PrefetchForStoreDataCacheBlock( TH, EA )
```

The **dcbtst** instruction provides a hint that performance will probably be improved if the block containing the byte addressed by EA is prefetched into the data cache because the program will soon store to the addressed byte.

An implementation may use TH values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure. TH values are synonymous with CT values, except that TH values only range from 0 to 7. Section 4.6.1.17.2, “User-Level Cache Management Instructions,” lists supported TH values.

The hint is ignored and no operation is performed if:

- The block addressed by EA is caching-inhibited or guarded
- The TH value is not supported by the implementation
- The access would cause a DTLB miss or a DSI
- The cache addressed by TH is disabled

The hint may be ignored for other implementation specific reasons. See the core reference manual. It is implementation dependent whether a hint that is ignored will cause debug events associated with the instruction.

This instruction is treated as a load for translation, permissions, and debug events (see Section 6.5.6.4, “Permission Control and Cache Management Instructions”), except that an interrupt is not taken for a translation or protection violation.

Other registers altered: None

dcbtstep

E.PD	Supervisor

Data Cache Block Touch for Store by External PID

dcbtstep TH,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	TH	**r**A	**r**B	0 0 1 1 1 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← rA
EA ← a + (rB)
PrefetchForStoreDataCacheBlock( TH, EA )
```

The **dcbtstep** instruction provides a hint that performance will probably be improved if the block containing the byte addressed by EA is prefetched into the data cache because the program will soon store to the addressed byte.

An implementation may use TH values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure. TH values are synonymous with CT values, except that TH values only range from 0 to 7. Section 4.6.1.17.2, "User-Level Cache Management Instructions," lists supported TH values.

The hint is ignored and no operation is performed if:

- The block addressed by EA is caching-inhibited or guarded
- The TH value is not supported by the implementation
- The access would cause a DTLB miss or a DSI
- The cache addressed by TH is disabled

The hint may be ignored for other implementation specific reasons. See the core reference manual. It is implementation dependent whether a hint that is ignored will cause debug events associated with the instruction.

This instruction is treated as a load for translation, permissions, and debug events (see Section 6.5.6.4, "Permission Control and Cache Management Instructions"), except that an interrupt is not taken for a translation or protection violation.

This instruction is guest supervisor privileged.

For **dcbtstep**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered: None

Note: **dcbtstep** is identical to **dcbtst** except for using EPLC for translation context.

dcbtstls

E.CL	User

dcbtstls

Data Cache Block Touch for Store and Lock Set

dcbtstls CT,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	CT	**r**A	**r**B	0 0 1 0 0 0 0 1 1 0	/

```
if rA = 0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
PrefetchDataCacheBlockLockSet(CT, EA)
```

The data cache specified by CT has the cache line corresponding to EA loaded and locked into the cache. If the line already exists in the cache, it is locked without being refetched.

Section 4.6.1.17.2, “User-Level Cache Management Instructions,” lists supported CT values. An implementation may use CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

Cache touch and lock set instructions let software lock cache lines into the cache to provide lower latency for critical cache accesses and more deterministic behavior. Locked lines do not participate in the normal replacement policy when a line must be victimized for replacement.

It is implementation dependent whether this instruction is treated as a load or store for translation, permissions, and debug events (see Section 6.5.6.4, “Permission Control and Cache Management Instructions”), and can cause DSI and DTLB error interrupts accordingly.

An unable to lock condition is said to occur any of the following conditions exist:

- The target address is cache-inhibited, or the storage attributes of the address uses a coherency protocol that does not support locking.
- The target cache is disabled or not present.
- The CT field of the instructions contains a value not supported by the implementation.

If an unable to lock condition occurs, no cache operation is performed and LICSR0[CUL] is set appropriately. Processors that implement **dcblq.** do not set LICSR0[CUL] when an unable to lock condition occurs and require software to query the status of the lock after attempting to lock it. Consult the core reference manual.

An overlocking condition is said to exist if all available ways for a given cache index are already locked and the locking instruction has no other exceptions. If an overlocking condition occurs for **dcbtls** and if the lock was targeted for the primary cache or secondary cache (CT = 0 or CT = 2), L1CSR0[CLO] (L2CSR0[L2LO] for CT = 2) is set. If lock overflow allocate is set (L1CSR0[CLOA] for CT = 0 and (L2CSR0[L2LOA] for CT = 2), the line is loaded and locked replacing and unlocking an implementation dependent line in the set. Some processors do not set lock overflow status in either L1CSR0 or L2CSR0. Consult the core reference manual.

The results of overlocking and unable to lock conditions for caches other than the primary cache and secondary cache are defined as part of the integrated device.

dcbtls can only be executed by user-level software if MSR[UCLE] = 1.

dcbtls can only be executed by guest supervisor software if MSRP[UCLEP] = 0. <E.HV>

Other registers altered:

- L1CSR0[CUL] if unable to lock occurs
- L1CSR0[CLO] (L2CSR0[L2LO]) if lock overflow occurs

NOTE: Software Considerations

Locks on cache lines can be lost for a variety of reasons. SeeSection 6.3.1, "Cache Line Locking <E.CL>."

dcbz

Base	User

dcbz

Data Cache Block Set to Zero

dcbz **r**A,**r**B

0 5	6 9	10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	0	rA	rB	1 1 1 1 1 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}$0 else a ← rA
EA ← a + (rB)
if L1CSR0$_{DCBZ32}$ then sz ← 32
                else sz ← CacheBlockSize()
ZeroDataCacheBlock(EA, sz)
```

If the block containing the addressed byte is in the data cache, all bytes of the block are cleared.

If the block containing the byte addressed by EA is not in the data cache and is in memory that is not caching-inhibited, the block is established in the data cache without fetching the block from main memory, and all bytes of the block are cleared.

If L1CSR0[DCBZ32] = 1, **dcbz** operates as if the cache block size is 32 bytes instead of the entire cache block. If the actual cache block size is larger than 32 bytes, then the remaining bytes in the cache block are architecturally unaffected by **dcbz**.

If the block containing the byte addressed by EA is in storage that is caching inhibited, is write through required, or cannot be established in the cache one of the following occurs:

- All bytes of the area of main storage that corresponds to the addressed block are set to zero
- An alignment interrupt is taken.

If the block containing the byte addressed by EA is in memory that is memory-coherence required and the block exists in a data cache of any other processors, it is kept coherent in those caches.

This instruction is treated as a store for translation, permissions, and debug events,. See Section 6.5.6.4, "Permission Control and Cache Management Instructions," and Section 4.6.1.17, "Cache Management Instructions."

This instruction may establish a block in the data cache without verifying that the associated real address is valid. This can cause a delayed machine check interrupt, as described in Section 7.8.2, "Machine Check, NMI, and Error Report Interrupts."

Other registers altered: None

NOTE: Software Considerations

Software should only use L1CSR0$_{DCBZ32}$ to perform 32 byte cache block operations to preserve software compatibility for software written that mistakenly assumed that a cache line was always 32 bytes.

NOTE: Software Considerations

If the block containing the byte addressed by EA is in memory that is caching-inhibited or write-through required and an alignment interrupt occurs, the alignment interrupt handler should clear all bytes of the area of main memory that corresponds to the addressed block.

dcbzep dcbzep

E.PD	Supervisor

Data Cache Block Zero by External PID

dcbzep **r**A,**r**B

0 5	6 9	10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	0	**r**A	**r**B	1 1 1 1 1 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← rA
EA ← a + (rB)
if L1CSR0$_{DCBZ32}$ then sz ← 32
                else sz ← CacheBlockSize()
ZeroDataCacheBlock(EA, sz)
```

If the block containing the addressed byte is in the data cache, all bytes of the block are cleared.

If the block containing the byte addressed by EA is not in the data cache and is in memory that is not caching-inhibited, the block is established in the data cache without fetching the block from main memory, and all bytes of the block are cleared.

If L1CSR0[DCBZ32] = 1, **dcbzep** operates as if the cache block size is 32 bytes instead of the entire cache block. If the actual cache block size is larger than 32 bytes, then the remaining bytes in the cache block are architecturally unaffected by **dcbzep**.

If the block containing the byte addressed by EA is in storage that is caching inhibited, is write through required, or cannot be established in the cache one of the following occurs:

- All bytes of the area of main storage that corresponds to the addressed block are set to zero
- An alignment interrupt is taken.

If the block containing the byte addressed by EA is in memory that is memory-coherence required and the block exists in a data cache of any other processors, it is kept coherent in those caches.

This instruction is treated as a store for translation, permissions, and debug events. See Section 6.5.6.4, “Permission Control and Cache Management Instructions,” and Section 4.6.1.17, “Cache Management Instructions.”

This instruction may establish a block in the data cache without verifying that the associated real address is valid. This can cause a delayed machine check interrupt, as described in Section 7.8.2, “Machine Check, NMI, and Error Report Interrupts.”

This instruction is guest supervisor privileged.

For **dcbzep**, the normal translation mechanism is not used. The EPSC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC[EPR] is used in place of MSR[PR]
- EPSC[EAS] is used in place of MSR[DS]

- EPSC[EPID] is used in place of the PID value
- EPSC[EGS] is used in place of MSR[GS] <E.HV>
- EPSC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered: None

Note: **dcbzep** is identical to **dcbz** except for using EPSC for translation context.

NOTE: Software Considerations

Software should only use L1CSR0$_{DCBZ32}$ to perform 32 byte cache block operations to preserve software compatibility for software written that mistakenly assumed that a cache line was always 32 bytes.

NOTE: Software Considerations

If the block containing the byte addressed by EA is in memory that is caching-inhibited or write-through required and an alignment interrupt occurs, the alignment interrupt handler should clear all bytes of the area of main memory that corresponds to the addressed block.

dcbzl

DEO	User

dcbzl

Data Cache Block set to Zero by Line

dcbzl **r**A,**r**B

0 5	6 9	10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	1	**r**A	**r**B	1 1 1 1 1 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
sz ← CacheBlockSize()
ZeroDataCacheBlock(EA, sz)
```

If the block containing the addressed byte is in the data cache, all bytes of the block are cleared.

If the block containing the byte addressed by EA is not in the data cache and is in memory that is not caching-inhibited, the block is established in the data cache without fetching the block from main memory, and all bytes of the block are cleared.

If the block containing the byte addressed by EA is in storage that is caching inhibited, is write through required, or cannot be established in the cache one of the following occurs:

- All bytes of the area of main storage that corresponds to the addressed block are set to zero
- An alignment interrupt is taken.

If the block containing the byte addressed by EA is in memory that is memory-coherence required and the block exists in a data cache of any other processors, it is kept coherent in those caches.

This instruction is treated as a store for translation, permissions, and debug events,. See Section 6.5.6.4, “Permission Control and Cache Management Instructions,” and Section 4.6.1.17, “Cache Management Instructions.”

This instruction may establish a block in the data cache without verifying that the associated real address is valid. This can cause a delayed machine check interrupt, as described in Section 7.8.2, “Machine Check, NMI, and Error Report Interrupts.”

Other registers altered: None

NOTE: Software Considerations

If the block containing the byte addressed by EA is in memory that is caching-inhibited or write-through required and an alignment interrupt occurs, the alignment interrupt handler should clear all bytes of the area of main memory that corresponds to the addressed block.

dcbzlep

E.PD, DEO	Supervisor

dcbzlep

Data Cache Block Zero Line by External PID

dcbzlep **r**A,**r**B

0 5	6 9	10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	1	**r**A	**r**B	1 1 1 1 1 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
sz ← CacheBlockSize()
ZeroDataCacheBlock(EA, sz)
```

If the block containing the addressed byte is in the data cache, all bytes of the block are cleared.

If the block containing the byte addressed by EA is not in the data cache and is in memory that is not caching-inhibited, the block is established in the data cache without fetching the block from main memory, and all bytes of the block are cleared.

If the block containing the byte addressed by EA is in storage that is caching inhibited, is write through required, or cannot be established in the cache one of the following occurs:

- All bytes of the area of main storage that corresponds to the addressed block are set to zero
- An alignment interrupt is taken.

If the block containing the byte addressed by EA is in memory that is memory-coherence required and the block exists in a data cache of any other processors, it is kept coherent in those caches.

This instruction is treated as a store for translation, permissions, and debug events,. See Section 6.5.6.4, “Permission Control and Cache Management Instructions,” and Section 4.6.1.17, “Cache Management Instructions.”

This instruction may establish a block in the data cache without verifying that the associated real address is valid. This can cause a delayed machine check interrupt, as described in Section 7.8.2, “Machine Check, NMI, and Error Report Interrupts.”

This instruction is guest supervisor privileged.

For **dcbzlep**, the normal translation mechanism is not used. The EPSC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC[EPR] is used in place of MSR[PR]
- EPSC[EAS] is used in place of MSR[DS]
- EPSC[EPID] is used in place of the PID value
- EPSC[EGS] is used in place of MSR[GS] <E.HV>
- EPSC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered: None

Note: **dcbzlep** is identical to **dcbzl** except for using EPSC for translation context.

NOTE: Software Considerations

If the block containing the byte addressed by EA is in memory that is caching-inhibited or write-through required and an alignment interrupt occurs, the alignment interrupt handler should clear all bytes of the area of main memory that corresponds to the addressed block.

divd divd

64	User

Divide Doubleword

divd	**r**D,**r**A,**r**B	(OE=0, Rc=0)
divd.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
divdo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
divdo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	OE	1 1 1 1 0 1 0 0 1	Rc

```
dividend$_{0:63}$ ← (rA)
divisor$_{0:63}$ ← (rB)
quotient ← dividend ÷ divisor
if ((dividend=-2$^{63}$) & (divisor=-1)) | (divisor=0) then do
   quotient ← undefined
   if OE=1 then do
      OV ← 1
      SO ← SO | OV
   if Rc=1 then do
      LT  ← undefined
      GT  ← undefined
      EQ  ← undefined
      CR0 ← LT || GT || EQ || SO
else do
   if OE=1 then do
      OV ← 0
      SO ← SO | OV
   if Rc=1 then do
      LT  ← quotient$_{m:63}$ < 0
      GT  ← quotient$_{m:63}$ > 0
      EQ  ← quotient$_{m:63}$ = 0
      CR0 ← LT || GT || EQ || SO
rD ← quotient
```

The 64-bit quotient of the contents of **r**A divided by the contents of **r**B is placed into **r**D. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies the following:

$$\text{dividend} = (\text{quotient} \times \text{divisor}) + r$$

Here, $0 \le r < |\text{divisor}|$ if the dividend is nonnegative and $-|\text{divisor}| < r \le 0$ if it is negative.

If any of the following divisions is attempted, the contents of **r**D are undefined as are (if Rc=1) the contents of the CR0[LT,GT,EQ]. In these cases, if OE=1, OV is set.

0x8000_0000_0000_0000 ÷ –1
<anything> ÷ 0

Other registers altered:

- CR0 (if Rc=1)
  SO OV (if OE=1)

divdu   divdu

64	User

Divide Doubleword Unsigned

divdu	**r**D,**r**A,**r**B	(OE=0, Rc=0)
divdu.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
divduo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
divduo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	OE	1 1 1 0 0 1 0 0 1	Rc

```
dividend$_{0:63}$ ← rA
divisor$_{0:63}$ ← rB
quotient ← dividend ÷ divisor
if (divisor=0) then do
   quotient ← undefined
   if OE=1 then do
       OV ← 1
       SO ← SO | OV
   if Rc=1 then do
       LT  ← undefined
       GT  ← undefined
       EQ  ← undefined
       CR0 ← LT || GT || EQ || SO
else do
   if OE=1 then do
       OV ← 0
       SO ← SO | OV
   if Rc=1 then do
       LT  ← quotient$_{m:63}$ < 0
       GT  ← quotient$_{m:63}$ > 0
       EQ  ← quotient$_{m:63}$ = 0
       CR0 ← LT || GT || EQ || SO
rD ← quotient
```

The 64-bit quotient of the contents of **r**A divided by the contents of **r**B is placed into **r**D. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers. The quotient is the unique unsigned integer that satisfies the following:

dividend = (quotient × divisor) + r

Here, 0 ≤ r < divisor.

If the following division is attempted, the contents of **r**D are undefined as are (if Rc=1) the contents of the CR0[LT,GT,EQ]. In this case, if OE=1, OV is set.

<anything> ÷ 0

Other registers altered:

- CR0 (if Rc=1)
  SO OV (if OE=1)

divw

Base	User

divw

Divide Word

divw	**r**D,**r**A,**r**B	(OE=0, Rc=0)
divw.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
divwo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
divwo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	OE	1 1 1 1 0 1 0 1 1	Rc

```
dividend$_{0:31}$ ← (rA)$_{32:63}$
divisor$_{0:31}$  ← (rB)$_{32:63}$
quotient$_{0:31}$ ← dividend ÷ divisor
if ((dividend=-2$^{31}$) & (divisor=-1)) | (divisor=0) then do
   quotient ← undefined
   if OE=1 then do
       OV ← 1
       SO ← SO | OV
   if Rc=1 then do
       LT  ← undefined
       GT  ← undefined
       EQ  ← undefined
       CR0 ← LT || GT || EQ || SO
else do
   if OE=1 then do
       OV ← 0
       SO ← SO | OV
   if Rc=1 then do
       if 64-bit mode then CR0 ← $^3$undefined || SO
       else do
           LT  ← quotient$_{32:63}$ < 0
           GT  ← quotient$_{32:63}$ > 0
           EQ  ← quotient$_{32:63}$ = 0
           CR0 ← LT || GT || EQ || SO
rD$_{32:63}$ ← quotient
rD$_{0:31}$  ← undefined
```

The 32-bit quotient of the contents of **r**A[32–63] divided by the contents of **r**B[32–63] is placed into **r**D[32–63]. **r**D[0–31] are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies the following:

$$\text{dividend} = (\text{quotient} \times \text{divisor}) + r$$

Here, $0 \le r < |\text{divisor}|$ if the dividend is nonnegative and $-|\text{divisor}| < r \le 0$ if it is negative.

If any of the following divisions is attempted, the contents of **r**D are undefined as are (if Rc=1) the contents of the CR0[LT,GT,EQ]. In these cases, if OE=1, OV is set.

0x8000_0000 ÷ –1
<anything> ÷ 0

Other registers altered:

- CR0(bits 0:2 undefined in 64-bit mode) (if Rc=1)
  SO OV (if OE=1)

divwu divwu

Base	User

Divide Word Unsigned

divwu	**r**D,**r**A,**r**B	(OE=0, Rc=0)
divwu.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
divwuo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
divwuo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)

0–5	6–10	11–15	16–20	21	22–30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	OE	1 1 1 0 0 1 0 1 1	Rc

```
$dividend_{0:31}$ ← $(rA)_{32:63}$
$divisor_{0:31}$ ← $(rB)_{32:63}$
$quotient_{0:31}$ ← dividend ÷ divisor
if (divisor=0) then do
   quotient ← undefined
   if OE=1 then do
       OV ← 1
       SO ← SO | OV
   if Rc=1 then do
       LT  ← undefined
       GT  ← undefined
       EQ  ← undefined
       CR0 ← LT || GT || EQ || SO
else do
   if OE=1 then do
       OV ← 0
       SO ← SO | OV
   if Rc=1 then do
       if 64-bit mode then CR0 ← $^{3}$undefined || SO
       else do
           LT  ← $quotient_{32:63}$ < 0
           GT  ← $quotient_{32:63}$ > 0
           EQ  ← $quotient_{32:63}$ = 0
           CR0 ← LT || GT || EQ || SO
$rD_{32:63}$ ← quotient
$rD_{0:31}$  ← undefined
```

The 32-bit quotient of the contents of **r**A[32–63] divided by the contents of **r**B[32–63] is placed into **r**D[32–63]. **r**D[0–31] are undefined. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR field 0 are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies the following:

$$\text{dividend} = (\text{quotient} \times \text{divisor}) + r$$

Here, $0 \le r < \text{divisor}$.

If an attempt is made to perform the following division, the contents of **r**D are undefined as are (if Rc=1) the contents of the LT, GT, and EQ bits of CR0. In this case, if OE=1 OV is set.

<anything> ÷ 0

Other registers altered:

- CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
- SO   OV (if OE=1)

dnh

E.ED	User

dnh

Debugger Notify Halt

dnh **DUI,DCTL**

0–5	6–10	11–15	16–20	21–30	31
0 1 0 0 1 1	DUI	DCTL	0 0 0 0 0	0 0 1 1 0 0 0 1 1 0	/

```
if $\text{EDBCR0}_{\text{DNH\_EN}}$ = 1 then
   implementation dependent register ← DUI
   halt processor
else
   illegal instruction exception
```

Execution of the **dnh** instruction causes the processor to halt if the external debug facility has enabled such action by previously setting EDBCR0[DNH_EN]. If the processor is halted, the contents of the DUI field are sent to the external debug facility to identify the reason for the halt.

If EDBCR0[DNH_EN] has not been previously set by the external debug facility, executing **dnh** produces an illegal instruction exception.

The **dnh** instruction is not privileged, and executes the same regardless of the state of MSR[PR].

The current state of the processor debug facility, whether the processor is in IDM or EDM mode has no effect on the execution of the **dnh** instruction.

Some processors will use the assembly syntax **dnh DUI**,**DUIS** to provide an additional operand. For these processors, DUIS is encoded in bits 11:20. The DUIS field is provided to pass additional information about the halt, but requires that actions be performed by the external debug facility to access the **dnh** instruction from memory to read the contents of the field.

Other registers altered: None

NOTE: Software Considerations

After **dnh** executes, the instruction itself can be read back by the illegal instruction interrupt handler or the external debug facility if the contents of the DUI field are of interest. If the processor entered the illegal instruction interrupt handler, software can use SRR0 to obtain the address of the **dnh** instruction that caused the handler to be invoked. If the processor is halted, the external debug facility can perform a **mfspr** NIA to obtain the address of the **dnh** instruction that caused the processor to halt.

dsn dsn

DS	User

Decorated Storage Notify

dsn **r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	**r**A	**r**B	0 1 1 1 0 1 1 0 0 1	/

```
EA ← (rB)
DECORATED_MEM_ADDR_ONLY(EA,0,(rA))
```

The decoration supplied by **r**A is sent to the storage addressed by EA.

This instruction is treated like a store with regards to address translation, memory ordering, and debug events.

Decorations are device specific. Consult the integrated device manual for definitions of decorations.

Other registers altered: None

NOTE: Software Considerations

Software must be acutely aware of the effects of operations defined by the decorations supplied and must ensure that the processor only issues operations appropriately. This generally requires the addresses to be marked as guarded and caching inhibited and any appropriate memory barriers.

ehpriv ehpriv

E.HV	User

Generate Embedded Hypervisor Privilege Exception

ehpriv **OC**

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	OC			0 1 0 0 0 0 1 1 1 0	/

The **ehpriv** instruction generates an embedded hypervisor privilege exception resulting in an embedded hypervisor privilege interrupt.

The OC field may be used by hypervisor software to provide a facility for emulated virtual instructions.

Other registers altered: None

NOTE: Software Considerations

ehpriv is analogous to a guaranteed illegal instruction encoding in that it guarantees that a hypervisor privilege exception is generated. The instruction is useful for programs that need to communicate information to the hypervisor state software, particularly as a means for implementing breakpoint operations in a hypervisor state managed debugger.

NOTE: Software Considerations

ehpriv servers as both a basic and an extended mnemonic. The assembler will recognize an ehpriv mnemonic with one operand as the basic form, and an ehpriv mnemonic with no operand as the extended form. In the extended form, the OC operand is omitted and assumed to be 0.

eqv eqv

Base	User

Equivalent

eqv **r**A,**r**S,**r**B (Rc=0)
eqv. **r**A,**r**S,**r**B (Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 1 0 0 0 1 1 1 0 0	Rc

```
result ← (rS) ≡ (rB)
rA ← result
if Rc=1 then do
       LT  ← result$_{m:63}$ < 0
       GT  ← result$_{m:63}$ > 0
       EQ  ← result$_{m:63}$ = 0
       CR0 ← LT || GT || EQ || SO
```

The contents of **r**S are XORed with the contents of **r**B and the one’s complement of the result is placed into **r**A.

Other registers altered:

- CR0 (if Rc=1)

evlddepx

E.PD, SP	Supervisor

evlddepx

Vector Load Double Word into Double Word by External PID Indexed

evlddepx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	rD	rA	rB	1 1 0 0 0 1 1 1 1 1	/

```
if rA = 0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
rD ← MEM(EA,8)
```

The double word addressed by EA is loaded from memory and placed in **r**D.

Figure 5-2 shows how bytes are loaded into **r**D as determined by the endian mode.

Byte address	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	h	g	f	e	d	c	b	a

Figure 5-2. evlddepx Results in Big- and Little-Endian Modes

An attempt to execute **evlddepx** while MSR[SPV]=0 causes an SPE/embedded floating point unavailable interrupt.

This instruction is guest supervisor privileged.

For **evlddepx**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of all PID registers.
- EPLC[EGS] is used in place of MSR[GS] <E.HV>.
- EPLC[ELPID] is used in place of LPID <E.HV>.

Other registers altered:

- ESR[EPID] (GESR[EPID] <E.HV>) and ESR[SPE] (GESR[SPE] <E.HV>) if an alignment interrupt, a data TLB error interrupt or a data storage interrupt occurs

evstddepx

E.PD, SP	Supervisor

evstddepx

Vector Store Double Word into Double Word by External PID Indexed

evstddepx **r**S,**r**A,**r**B

0 ... 5	6 ... 10	11 ... 15	16 ... 20	21 ... 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 1 1 0 0 1 1 1 1 1	/

```
if rA = 0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
MEM(EA,8) ← (rS)
```

Figure 5-3 shows how bytes are stored from **r**S as determined by the endian mode.

Byte address	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	h	g	f	e	d	c	b	a

Figure 5-3. evstddepx Results in Big- and Little-Endian Modes

An attempt to execute **evstddepx** while MSR[SPV]=0 causes an SPE/embedded floating point unavailable interrupt.

This instruction is guest supervisor privileged.

For **evstddepx**, the normal translation mechanism is not used. The EPSC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC[EPR] is used in place of MSR[PR]
- EPSC[EAS] is used in place of MSR[DS]
- EPSC[EPID] is used in place of all PID registers.
- EPSC[EGS] is used in place of MSR[GS] <E.HV>.
- EPSC[ELPID] is used in place of LPID <E.HV>.

Other registers altered:

- ESR[EPID] (GESR[EPID] <E.HV>) and ESR[SPE] (GESR[SPE] <E.HV>) if an alignment interrupt, a data TLB error interrupt or a data storage interrupt occurs

extsb

Base	User

extsb

Extend Sign Byte

extsb	**r**A,**r**S	(Rc=0)
extsb.	**r**A,**r**S	(Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	///	1 1 1 0 1 1 1 0 1 0	Rc

```
s ← $(rS)_{56}$
result ← $^{n}s$ || $(rS)_{56:63}$
rA ← result
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
```

The contents of **r**S[56–63] are placed into **r**A[56–63]. Bit **r**S[56] is copied into bits 0–55 of **r**A.

Other registers altered:

- CR0 (if Rc=1)

extsh extsh

Base	User

Extend Sign Halfword

extsh	**r**A,**r**S	(Rc=0)
extsh.	**r**A,**r**S	(Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	///	1 1 1 0 0 1 1 0 1 0	Rc

```
s ← $(rS)_{48}$
result ← $^{n}s$ || $(rS)_{48:63}$
rA ← result
if Rc=1 then do
    LT  ← $result_{m:63}$ < 0
    GT  ← $result_{m:63}$ > 0
    EQ  ← $result_{m:63}$ = 0
    CR0 ← LT || GT || EQ || SO
```

The contents of **r**S[48–63] are placed into **r**A[48–63]. Bit **r**S[48] is copied into bits 0–47 of **r**A.

Other registers altered:

- CR0 (if Rc=1)

extsw

64	User

extsw

Extend Sign Word

extsw	**r**A,**r**S	(Rc=0)
extsw.	**r**A,**r**S	(Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	///	1 1 1 1 0 1 1 0 1 0	Rc

```
s ← $(rS)_{32}$
result ← $^{n}s \parallel rS_{32:63}$
rA ← result
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT ‖ GT ‖ EQ ‖ SO
```

The contents of **r**S[32–63] are placed into **r**A[32–63]. Bit **r**S[32] is copied into bits 0–31 of **r**A.

Other registers altered:

- CR0 (if Rc=1)

fabs

Floating-Point	User

fabs

Floating Absolute Value

fabs **fr**D,**fr**B (Rc=0)
fabs. **fr**D,**fr**B (Rc=1)

0					5	6 10	11 15	16 20	21									30	31
1	1	1	1	1	1	**fr**D	///	**fr**B	0	1	0	0	0	0	1	0	0	0	Rc

```
frD ← 0b0 ||(frB)$_{1:63}$
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The contents of **fr**B with bit 0 cleared are placed into **fr**D.

If MSR[FP]=0, an attempt to execute **fabs**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- CR1 (if Rc=1)

fadd

Floating-Point	User

Floating Add [Single]

fadd	**fr**D,**fr**A,**fr**B	(P=1, Rc=0)
fadd.	**fr**D,**fr**A,**fr**B	(P=1, Rc=1)
fadds	**fr**D,**fr**A,**fr**B	(P=0, Rc=0)
fadds.	**fr**D,**fr**A,**fr**B	(P=0, Rc=1)

0 2	3	4 5	6 10	11 15	16 20	21 25	26 30	31
1 1 1	P	1 1	**fr**D	**fr**A	**fr**B	///	1 0 1 0 1	Rc

```
if P=1 then frD ← (frA) $+_{dp}$ (frB)
else        frD ← (frA) $+_{sp}$ (frB)
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The floating-point operand in **fr**A is added to the floating-point operand in **fr**B.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field, FPSCR[RN], and placed into **fr**D.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits of the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

If MSR[FP]=0, an attempt to execute **fadd**[**s**][**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXISI
  CR1 (if Rc=1)

fcfid fcfid

Floating-Point	User

Floating Convert From Integer Doubleword

fcfid **fr**D,**fr**B
fcfid. **fr**D,**fr**B

0 5	6 10	11 15	16 20	21 30	31
1 1 1 1 1 1	**fr**D	///	**fr**B	1 1 0 1 0 0 1 1 1 0	Rc

```
frD ← ConvertIntDoubletoFloat(frB)
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The 64-bit signed operand in **fr**B is converted to an infinitely precise floating-point integer. The result of the conversion is rounded to double-precision, as specified by FPSCR[RN], and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

If MSR[FP]=0, an attempt to execute **fcfid** causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF FR FI FX XX
- CR1 (if Rc=1)

fcmpo

Floating-Point	User

fcmpo

Floating Compare Ordered

fcmpo **crfD**,**fr**A,**fr**B

0 5	6	10	11 15	16 20	21 30	31
1 1 1 1 1 1	**crfD**	//	**fr**A	**fr**B	0 0 0 0 1 0 0 0 0 0	/

```
if (frA) is a NaN or (frB) is a NaN then c ← 0b0001
else if (frA) < (frB) then               c ← 0b1000
else if (frA) > (frB) then               c ← 0b0100
else                                     c ← 0b0010
FPCC ← c
CR$_{4\times\mathbf{crfD}:4\times\mathbf{crfD}+3}$ ← c
if (frA) is a SNaN or (frB) is a SNaN then do
   VXSNAN ← 1
   if VE=0 then VXVC ← 1
else if (frA) is a QNaN or (frB) is a QNaN then VXVC ← 1
```

The floating-point operand in **fr**A is compared to the floating-point operand in **fr**B. The result of the compare is placed into CR field **crfD** and the FPCC.

If either of the operands is a NaN, either quiet or signaling, the CR field **crfD** and the FPCC are set to reflect unordered. If either of the operands is a signaling NaN and invalid operation is disabled (VE=0), VXVC is set. If neither operand is a signaling NaN but at least one operand is a quiet NaN, then VXVC is set.

If MSR[FP]=0, an attempt to execute **fcmpo** causes a floating-point unavailable interrupt.

Other registers altered:

- CR field **crfD**
  FPCC FX VXSNAN VXVC

fcmpu

Floating-Point	User

fcmpu

Floating Compare Unordered

fcmpu **crfDcrfD**,**fr**A,**fr**B

0 5	6	10	11 15	16 20	21 30	31
1 1 1 1 1 1	**crfD**	//	**fr**A	**fr**B	0 0 0 0 0 0 0 0 0 0	/

```
if (frA) is a NaN or (frB) is a NaN then c ← 0b0001
else if (frA) < (frB) then               c ← 0b1000
else if (frA) > (frB) then               c ← 0b0100
else                                     c ← 0b0010
FPCC ← c
$CR_{4\times\mathbf{crfD}:4\times\mathbf{crfD}+3}$ ← c
if (frA) is a SNaN or (frB) is a SNaN then
   VXSNAN ← 1
```

The floating-point operand in **fr**A is compared to the floating-point operand in **fr**B. The result of the compare is placed into CR field **crfD** and the FPCC.

If either of the operands is a NaN, either quiet or signaling, the CR field **crfD** and the FPCC are set to reflect unordered.

If either of the operands is a signaling NaN, VXSNAN is set.

If MSR[FP]=0, an attempt to execute **fcmpu** causes a floating-point unavailable interrupt.

Other registers altered:

- CR field **crfD**
  FPCC FX VXSNAN

fctid

Floating-Point	User

fctid

Floating Convert To Integer Doubleword

fctid **fr**D,**fr**B
fctid. **fr**D,**fr**B

0 5	6 10	11 15	16 20	21 30	31
1 1 1 1 1 1	**fr**D	///	**fr**B	1 1 0 0 1 0 1 1 1 0	Rc

```
if (frB) is a Nan then do
   result ← 0x8000_0000_0000_0000
   VXCVI ← 1
   if (frB) is a SNaN then VXSNAN ← 1
else do
   result ← ConvertFloattoIntDouble((frB),$\text{FPSCR}_{RN}$)
   if result > $2^{63}$-1 then do
       result ← 0x7FFF_FFFF_FFFF_FFFF
   VXCVI ← 1
   if result < -$2^{63}$ then do
       result ← 0x8000_0000_0000_0000
       VXCVI ← 1
frD ← result
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

If the floating-point operand in frB is a NaN, then the result is 0x8000_0000_0000_0000, VXCVI is set to 1, and if the operand is an SNaN, VXSNAN is set to 1.

Otherwise the floating-point operand in **fr**B is converted to a 64-bit signed integer, using the rounding mode specified by FPSCR[RN]. If the rounded value *result* is representable as a doubleword integer, then *result* is placed into **fr**D. If *result* is greater than 2^{63}–1, then 0x7FFF_FFFF_FFFF_FFFF is placed into **fr**D and VXCVI is set. If *result* is less than -2^{63}, then 0x8000_0000_0000_0000 is placed into **fr**D and VXCVI is set.

If an enabled invalid operation exception occurs, then no result is placed in **fr**D.

Except for enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

If MSR[FP]=0, an attempt to execute **fctid** causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF (undefined) FR FI FX XX VXSNAN VXCVI
- CR1 (if Rc=1)

fctidz fctidz

Floating-Point	User

Floating Convert To Integer Doubleword with round toward Zero

fctidz **fr**D,**fr**B
fctidz. **fr**D,**fr**B

0 5	6 10	11 15	16 20	21 30	31
1 1 1 1 1 1	**fr**D	///	**fr**B	1 1 0 0 1 0 1 1 1 1	Rc

```
if (frB) is a Nan then do
   result ← 0x8000_0000_0000_0000
   VXCVI ← 1
   if (frB) is a SNaN then VXSNAN ← 1
else do
   result ← ConvertFloattoIntDouble((frB),0b01)
   if result > $2^{63}$-1 then do
       result ← 0x7FFF_FFFF_FFFF_FFFF
       VXCVI ← 1
   if result < $-2^{63}$ then do
       result ← 0x8000_0000_0000_0000
       VXCVI ← 1
frD ← result
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

If the floating-point operand in frB is a NaN, then the result is 0x8000_0000_0000_0000, VXCVI is set to 1, and if the operand is an SNaN, VXSNAN is set to 1.

Otherwise the floating-point operand in **fr**B is converted to a 64-bit signed integer, using the rounding mode round towards zero. If the rounded value *result* is representable as a doubleword integer, then *result* is placed into **fr**D. If *result* is greater than 2^{63}–1, then 0x7FFF_FFFF_FFFF_FFFF is placed into **fr**D and VXCVI is set. If *result* is less than -2^{63}, then 0x8000_0000_0000_0000 is placed into **fr**D and VXCVI is set.

If an enabled invalid operation exception occurs, then no result is placed in **fr**D.

Except for enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

If MSR[FP]=0, an attempt to execute **fctid** causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF (undefined) FR FI FX XX VXSNAN VXCVI
- CR1 (if Rc=1)

fctiw

Floating-Point	User

fctiw

Floating Convert To Integer Word

fctiw	**fr**D,**fr**B	(Rc=0)
fctiw.	**fr**D,**fr**B	(Rc=1)

0 5	6 10	11 15	16 20	21 29 30	31
1 1 1 1 1 1	**fr**D	///	**fr**B	0 0 0 0 0 0 1 1 1 0	Rc

```
    if ‘fctiw[.]’  then round_mode ← FPSCR[RN]
    if ‘fctiwz[.]’ then round_mode ← 0b01
    sign ← $frB_0$
    If frB[1:11] = 2047 and frB[12:63] = 0 then goto Infinity Operand
    If frB[1:11] = 2047 and $frB_{12}$ = 0 then goto SNaN Operand
    If frB[1:11] = 2047 and $frB_{12}$ = 1 then goto QNaN Operand
    If frB[1:11] > 1086 then goto Large Operand
    If frB[1:11] > 0 then exp ← frB[1:11] : 1023   /* exp : bias */
    If frB[1:11] = 0 then exp ← :1022
/* normal; need leading 0 for later complement */
    If frB[1:11] > 0 then $frac_{0:64}$ ← 0b01 || frB[12:63] || $^{11}0$
/* denormal */
    If frB[1:11] = 0 then $frac_{0:64}$ ← 0b00 || frB[12:63] || $^{11}0$
    gbit || rbit || xbit ← 0b000
    Do i=1,63:exp   /* do the loop 0 times if exp = 63 */
         $frac_{0:64}$ || gbit || rbit || xbit ← 0b0 || $frac_{0:64}$ || gbit || (rbit | xbit)
    End
    Round Integer( sign, $frac_{0:64}$, gbit, rbit, xbit, round_mode )
/* needed leading 0 for :$2^{64}$ < frB < :$2^{63}$ */
    If sign=1 then $frac_{0:64}$ ← ¬$frac_{0:64}$ + 1
    If $frac_{0:64}$ > $2^{31}$:1 then goto Large Operand
    If $frac_{0:64}$ < :$2^{31}$ then goto Large Operand
    FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
    frD ← 0xuuuu_uuuu || $frac_{33:64}$  /* u is undefined hex digit */
    FPSCR[FPRF] ← undefined
    Done
Round Integer( sign, frac0:64, gbit, rbit, xbit, round_mode ):
    inc ← 0
    If round_mode = 0b00 then  /* comparison ignores u bits */
         Do
              If sign || $frac_{64}$ || gbit || rbit || xbit = 0bu11uu then inc ← 1
              If sign || $frac_{64}$ || gbit || rbit || xbit = 0bu011u then inc ← 1
              If sign || $frac_{64}$ || gbit || rbit || xbit = 0bu01u1 then inc ← 1
         End
    If round_mode = 0b10 then  /* comparison ignores u bits */
         Do
              If sign || $frac_{64}$ || gbit || rbit || xbit = 0b0u1uu then inc ← 1
              If sign || $frac_{64}$ || gbit || rbit || xbit = 0b0uu1u then inc ← 1
              If sign || $frac_{64}$ || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
         End
    If round_mode = 0b11 then   /* comparison ignores u bits */
         Do
              If sign || $frac_{64}$ || gbit || rbit || xbit = 0b1u1uu then inc ← 1
              If sign || $frac_{64}$ || gbit || rbit || xbit = 0b1uu1u then inc ← 1
              If sign || $frac_{64}$ || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
         End

    $frac_{0:64}$ ← $frac_{0:64}$ + inc
    FPSCR[FR] ← inc
    FPSCR[FI] ← gbit | rbit | xbit
    Return
Infinity Operand:
    FPSCR[FR,FI,VXCVI] ← 0b001
    If FPSCR[VE] = 0 then Do  /* u is undefined hex digit */
         If sign = 0 then frD ← 0xuuuu_uuuu_7FFF_FFFF
         If sign = 1 then frD ← 0xuuuu_uuuu_8000_0000
         FPSCR[FPRF] ← undefined
    End
    Done
```

```
SNaN Operand:
    FPSCR[FR,FI,VXSNAN,VXCVI] ← 0b0011
    If FPSCR[VE] = 0 then Do    /* u is undefined hex digit */
        frD ← 0xuuuu_uuuu_8000_0000
        FPSCR[FPRF] ← undefined
    End
    Done
QNaN Operand:
    FPSCR[FR,FI,VXCVI] ← 0b001
    If FPSCR[VE] = 0 then Do  /* u is undefined hex digit */
        frD ← 0xuuuu_uuuu_8000_0000
        FPSCR[FPRF] ← undefined
    End
    Done
Large Operand:
    FPSCR[FR,FI,VXCVI] ← 0b001
    If FPSCR[VE] = 0 then Do  /* u is undefined hex digit */
        If sign = 0 then frD ← 0xuuuu_uuuu_7FFF_FFFF
        If sign = 1 then frD ← 0xuuuu_uuuu_8000_0000
        FPSCR[FPRF] ← undefined
    End
    Done
```

```
if (frB) is a Nan then do
    result ← 0x8000_0000
    VXCVI ← 1
    if (frB) is a SNaN then VXSNAN ← 1
else do
    result ← ConvertFloattoIntWord((frB),$FPSCR_{RN}$)
    if result > $2^{31}$-1 then do
        result ← 0x7FFF_FFFF
        VXCVI ← 1
    if result < -$2^{31}$ then do
        result ← 0x8000_0000
        VXCVI ← 1
frD ← $^{32}$undefined || $result_{32:63}$
if Rc=1 then do
    CR1 ← FX || FEX || VX || OX
```

If the floating-point operand in frB is a NaN, then the result is 0x8000_0000, VXCVI is set to 1, and if the operand is an SNaN, VXSNAN is set to 1.

Otherwise the floating-point operand in **fr**B is converted to a 32-bit signed integer, using the rounding mode specified by FPSCR[RN]. If the rounded value *result* is representable as a 32-bit integer, then *result* is placed into **fr**$D_{32:63}$. If *result* is greater than 2^{31}–1, then 0x7FFF_FFFF is placed into **fr**$D_{32:63}$ and VXCVI is set. If *result* is less than -2^{31}, then 0x8000_0000 is placed into **fr**$D_{32:63}$ and VXCVI is set.

fr$D_{32:63}$ are undefined.

If an enabled invalid operation exception occurs, then no result is placed in **fr**D.

Except for enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

If MSR[FP]=0, an attempt to execute **fctiw** causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF (undefined)   FR   FI   FX   XX   VXSNAN VXCVI
- CR1 (if Rc=1)

fctiwz

Floating-Point	User

fctiwz

Floating Convert To Integer Word with round toward Zero

fctiwz	**fr**D,**fr**B	(Rc=0)
fctiwz.	**fr**D,**fr**B	(Rc=1)

0 5	6 10	11 15	16 20	21 29 30	31
1 1 1 1 1 1	**fr**D	///	**fr**B	0 0 0 0 0 0 1 1 1 1	Rc

```
        if ‘fctiw[.]’  then round_mode ← FPSCR[RN]
        if ‘fctiwz[.]’ then round_mode ← 0b01
        sign ← $frB_0$
        If frB[1:11] = 2047 and frB[12:63] = 0 then goto Infinity Operand
        If frB[1:11] = 2047 and $frB_{12}$ = 0 then goto SNaN Operand
        If frB[1:11] = 2047 and $frB_{12}$ = 1 then goto QNaN Operand
        If frB[1:11] > 1086 then goto Large Operand
        If frB[1:11] > 0 then exp ← frB[1:11] : 1023   /* exp : bias */
        If frB[1:11] = 0 then exp ← :1022
    /* normal; need leading 0 for later complement */
        If frB[1:11] > 0 then $frac_{0:64}$ ← 0b01 || frB[12:63] || $^{11}0$
    /* denormal */
        If frB[1:11] = 0 then $frac_{0:64}$ ← 0b00 || frB[12:63] || $^{11}0$
        gbit || rbit || xbit ← 0b000
        Do i=1,63:exp   /* do the loop 0 times if exp = 63 */
            $frac_{0:64}$ || gbit || rbit || xbit ← 0b0 || $frac_{0:64}$ || gbit || (rbit | xbit)
        End
        Round Integer( sign, $frac_{0:64}$, gbit, rbit, xbit, round_mode )
    /* needed leading 0 for $:2^{64}$ < frB < $:2^{63}$ */
        If sign=1 then $frac_{0:64}$ ← ¬$frac_{0:64}$ + 1
        If $frac_{0:64}$ > $2^{31}$:1 then goto Large Operand
        If $frac_{0:64}$ < $:2^{31}$ then goto Large Operand
        FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
        frD ← 0xuuuu_uuuu || $frac_{33:64}$  /* u is undefined hex digit */
        FPSCR[FPRF] ← undefined
        Done
    Round Integer( sign, frac0:64, gbit, rbit, xbit, round_mode ):
        inc ← 0
        If round_mode = 0b00 then  /* comparison ignores u bits */
            Do
                If sign || $frac_{64}$ || gbit || rbit || xbit = 0bu11uu then inc ← 1
                If sign || $frac_{64}$ || gbit || rbit || xbit = 0bu011u then inc ← 1
                If sign || $frac_{64}$ || gbit || rbit || xbit = 0bu01u1 then inc ← 1
            End
        If round_mode = 0b10 then  /* comparison ignores u bits */
            Do
                If sign || $frac_{64}$ || gbit || rbit || xbit = 0b0u1uu then inc ← 1
                If sign || $frac_{64}$ || gbit || rbit || xbit = 0b0uu1u then inc ← 1
                If sign || $frac_{64}$ || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
            End
        If round_mode = 0b11 then   /* comparison ignores u bits */
            Do
                If sign || $frac_{64}$ || gbit || rbit || xbit = 0b1u1uu then inc ← 1
                If sign || $frac_{64}$ || gbit || rbit || xbit = 0b1uu1u then inc ← 1
                If sign || $frac_{64}$ || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
            End

        $frac_{0:64}$ ← $frac_{0:64}$ + inc
        FPSCR[FR] ← inc
        FPSCR[FI] ← gbit | rbit | xbit
        Return
    Infinity Operand:
        FPSCR[FR,FI,VXCVI] ← 0b001
        If FPSCR[VE] = 0 then Do  /* u is undefined hex digit */
            If sign = 0 then frD ← 0xuuuu_uuuu_7FFF_FFFF
            If sign = 1 then frD ← 0xuuuu_uuuu_8000_0000
            FPSCR[FPRF] ← undefined
        End
        Done
```

```
SNaN Operand:
    FPSCR[FR,FI,VXSNAN,VXCVI] ← 0b0011
    If FPSCR[VE] = 0 then Do    /* u is undefined hex digit */
         frD ← 0xuuuu_uuuu_8000_0000
         FPSCR[FPRF] ← undefined
    End
    Done
QNaN Operand:
    FPSCR[FR,FI,VXCVI] ← 0b001
    If FPSCR[VE] = 0 then Do  /* u is undefined hex digit */
         frD ← 0xuuuu_uuuu_8000_0000
         FPSCR[FPRF] ← undefined
    End
    Done
Large Operand:
    FPSCR[FR,FI,VXCVI] ← 0b001
    If FPSCR[VE] = 0 then Do  /* u is undefined hex digit */
         If sign = 0 then frD ← 0xuuuu_uuuu_7FFF_FFFF
         If sign = 1 then frD ← 0xuuuu_uuuu_8000_0000
         FPSCR[FPRF] ← undefined
    End
    Done
```

```
if (frB) is a Nan then do
    result ← 0x8000_0000
    VXCVI ← 1
    if (frB) is a SNaN then VXSNAN ← 1
else do
    result ← ConvertFloattoIntWord((frB),0b01)
    if result > $2^{31}$-1 then do
         result ← 0x7FFF_FFFF
         VXCVI ← 1
    if result < -$2^{31}$ then do
         result ← 0x8000_0000
         VXCVI ← 1
frD ← $^{32}$undefined || $result_{32:63}$
if Rc=1 then do
    CR1 ← FX || FEX || VX || OX
```

If the floating-point operand in frB is a NaN, then the result is 0x8000_0000, VXCVI is set to 1, and if the operand is an SNaN, VXSNAN is set to 1.

Otherwise the floating-point operand in **fr**B is converted to a 32-bit signed integer, using the rounding mode round toward zero. If the rounded value *result* is representable as a 32-bit integer, then *result* is placed into **fr**$D_{32:63}$. If *result* is greater than 2^{31}–1, then 0x7FFF_FFFF is placed into **fr**$D_{32:63}$ and VXCVI is set. If *result* is less than -2^{31}, then 0x8000_0000 is placed into **fr**$D_{32:63}$ and VXCVI is set.

fr$D_{32:63}$ are undefined.

If an enabled invalid operation exception occurs, then no result is placed in **fr**D.

Except for enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

If MSR[FP]=0, an attempt to execute **fctiwz** causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF (undefined)   FR   FI   FX   XX   VXSNAN VXCVI
- CR1 (if Rc=1)

fdiv fdiv

Floating-Point	User

Floating Divide [Single]

fdiv	**fr**D,**fr**A,**fr**B	(P=1, Rc=0)
fdiv.	**fr**D,**fr**A,**fr**B	(P=1, Rc=1)
fdivs	**fr**D,**fr**A,**fr**B	(P=0, Rc=0)
fdivs.	**fr**D,**fr**A,**fr**B	(P=0, Rc=1)

0 2	3	4 5	6 10	11 15	16 20	21 25	26 30	31
1 1 1	P	1 1	**fr**D	**fr**A	**fr**B	///	1 0 0 1 0	Rc

```
if P=1 then frD ← (frA) $\div_{dp}$ (frB)
else        frD ← (frA) $\div_{sp}$ (frB)
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The floating-point operand in **fr**A is divided by the floating-point operand in **fr**B. The remainder is not supplied as a result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field, FPSCR[RN], and placed into **fr**D.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1 and zero divide exceptions when FPSCR[ZE]=1.

If MSR[FP]=0, an attempt to execute **fdiv**[**.**] or **fdivs**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF FR FI FX OX UX ZX XX VXSNAN VXIDI VXZDZ
- CR1 (if Rc=1)

fmadd

Floating-Point	User

fmadd

Floating Multiply-Add [Single]

fmadd	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=0)
fmadd.	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=1)
fmadds	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=0)
fmadds.	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=1)

0 2	3	4 5	6 10	11 15	16 20	21 25	26 30	31
1 1 1	P	1 1	**fr**D	**fr**A	**fr**B	**fr**C	1 1 1 0 1	Rc

```
if P=1 then frD ← ((frA) $\times_{fp}$ (frC)) $+_{dp}$ (frB)
else        frD ← ((frA) $\times_{fp}$ (frC)) $+_{sp}$ (frB)
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is added to this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field, FPSCR[RN], and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

If MSR[FP]=0, an attempt to execute **fmadd**[**.**] or **fmadd**s[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ
- CR1 (if Rc=1)

fmr

Floating-Point	User

fmr

Floating Move Register

fmr **fr**D,**fr**B (Rc=0)
fmr. **fr**D,**fr**B (Rc=1)

0 5	6 10	11 15	16 20	21 30	31
1 1 1 1 1 1	**fr**D	///	**fr**B	0 0 0 1 0 0 1 0 0 0	Rc

```
frD ← (frB)
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The contents of **fr**B are placed into **fr**D.

If MSR[FP]=0, an attempt to execute **fmr[.]** causes a floating-point unavailable interrupt.

Other registers altered:

- CR1 (if Rc=1)

fmsub fmsub

Floating-Point	User

Floating Multiply-Subtract [Single]

fmsub	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=0)
fmsub.	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=1)
fmsubs	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=0**)**
fmsubs.	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=1)

0 2	3	4 5	6 10	11 15	16 20	21 25	26 30	31
1 1 1	P	1 1	**fr**D	**fr**A	**fr**B	**fr**C	1 1 1 0 0	Rc

```
if P=1 then frD ← ((frA) $\times_{fp}$ (frC)) $-_{dp}$ (frB)
else        frD ← ((frA) $\times_{fp}$ (frC)) $-_{sp}$ (frB)
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is subtracted from this intermediate result.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field, FPSCR[RN], and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

If MSR[FP]=0, an attempt to execute **fmsub**[**.**] or **fmsub**s[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ
- CR1 (if Rc=1)

fmul fmul

Floating-Point	User

Floating Multiply [Single]

fmul	**fr**D,**fr**A,**fr**C	(P=1, Rc=0)
fmul.	**fr**D,**fr**A,**fr**C	(P=1, Rc=1)
fmuls	**fr**D,**fr**A,**fr**C	(P=0, Rc=0)
fmuls.	**fr**D,**fr**A,**fr**C	(P=0, Rc=1)

0 2	3	4 5	6 10	11 15	16 20	21 25	26 30	31
1 1 1	P	1 1	**fr**D	**fr**A	///	**fr**C	1 1 0 0 1	Rc

```
if P=1 then frD ← (frA) $\times_{dp}$ (frC)
else        frD ← (frA) $\times_{sp}$ (frC)
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field, FPSCR[RN], and placed into **fr**D.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

If MSR[FP]=0, an attempt to execute **fmul**[**.**] or **fmuls**[**.**]causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXIMZ
- CR1 (if Rc=1)

fnabs

Floating-Point	User

fnabs

Floating Negative Absolute Value

fnabs	**fr**D,**fr**B	(Rc=0)
fnabs.	**fr**D,**fr**B	(Rc=1)

0 5	6 10	11 15	16 20	21 30	31
1 1 1 1 1 1	**fr**D	///	**fr**B	0 0 1 0 0 0 1 0 0 0	Rc

```
frD ← 0b1 ||(frB)$_{1:63}$
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The contents of **fr**B with bit 0 set are placed into **fr**D.

If MSR[FP]=0, an attempt to execute **fnabs**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- CR1 (if Rc=1)

fneg fneg

Floating-Point	User

Floating Negate

fneg	**fr**D,**fr**B	(Rc=0)
fneg.	**fr**D,**fr**B	(Rc=1)

0 5	6 10	11 15	16 20	21 30	31
1 1 1 1 1 1	**fr**D	///	**fr**B	0 0 0 0 1 0 1 0 0 0	Rc

```
frD ← ¬(frB)$_0$ || (frB)$_{1:63}$
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The contents of **fr**B with bit 0 inverted are placed into **fr**D.

If MSR[FP]=0, an attempt to execute **fneg**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- CR1 (if Rc=1)

fnmadd

Floating-Point	User

fnmadd

Floating Negative Multiply-Add [Single]

fnmadd	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=0)
fnmadd.	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=1)
fnmadds	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=0)
fnmadds.	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=1)

0 2	3	4 5	6 10	11 15	16 20	21 25	26 30	31
1 1 1	P	1 1	**fr**D	**fr**A	**fr**B	**fr**C	1 1 1 1 1	Rc

```
if P=1 then frD ← -(((frA) $\times_{fp}$ (frC)) $+_{dp}$ (frB))
else        frD ← -(((frA) $\times_{fp}$ (frC)) $+_{sp}$ (frB))
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is added to this intermediate result, then the result is negated.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field, FPSCR[RN], then negated and placed into **fr**D.

This instruction produces the same result as would be obtained by using the Floating Multiply-Add instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

An attempt to execute **fnmadd**[**.**] or **fnmadds**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ
- CR1 (if Rc=1)

fnmsub

Floating-Point	User

fnmsub

Floating Negative Multiply-Subtract [Single]

fnmsub	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=0)
fnmsub.	**fr**D,**fr**A,**fr**C,**fr**B	(P=1, Rc=1)
fnmsubs	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=0)
fnmsubs.	**fr**D,**fr**A,**fr**C,**fr**B	(P=0, Rc=1)

0 2	3	4 5	6 10	11 15	16 20	21 25	26 30	31
1 1 1	P	1 1	**fr**D	**fr**A	**fr**B	**fr**C	1 1 1 1 0	Rc

```
if P=1 then frD ← -(((frA) $\times_{fp}$ )frC)) $-_{dp}$ (frB))
else        frD ← -(((frA) $\times_{fp}$ (frC)) $-_{sp}$ (frB))
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is subtracted from this intermediate result, then the result is negated..

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field, FPSCR[RN], then negated and placed into **fr**D.

This instruction produces the same result as would be obtained by using the Floating Multiply-Subtract instruction and then negating the result, with the following exceptions.

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of 0.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

An attempt to execute **fnmsub**[**.**] or **fnmsubs**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXISI VXIMZ
- CR1 (if Rc=1)

fres | Floating-Point | User | fres

Floating Reciprocal Estimate Single

fres	**fr**D,**fr**B	(Rc=0)
fres.	**fr**D,**fr**B	(Rc=1)

0 5	6 10	11 15	16 20	21 25	26 30	31
1 1 1 0 1 1	**fr**D	///	**fr**B	///	1 1 0 0 0	Rc

```
frD ← FPReciprocalEstimate( (frB) )
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

A single-precision estimate of the reciprocal of the floating-point operand in **fr**B is placed into **fr**D. The estimate placed into **fr**D is correct to a precision of one part in 256 of the reciprocal of (**fr**B), that is,

$$\left|\frac{\text{estimate} - \frac{1}{x}}{\frac{1}{x}}\right| \le \frac{1}{256}$$

In this example, x is the initial value in **fr**B. Note that the value placed into **fr**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the operand is summarized in Table 5-8.

Table 5-8. Operations with Special Values

Operand	Result	Exception
–∞	–0	None
–0	–∞ (No result if FPSCR[ZE] = 1)	ZX
+0	+∞ (No result if FPSCR[ZE] = 1)	ZX
+∞	+0	None
SNaN	QNaN (No result if FPSCR[VE] = 1.)	VXSNAN
QNaN	QNaN	None

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1 and zero divide exceptions when FPSCR[ZE]=1.

If MSR[FP]=0, an attempt to execute **fres**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF   FR (undefined)   FI (undefined)
  FX   OX   UX   ZX   VXSNAN
  CR1 (if Rc=1)

frsp

Floating-Point	User

frsp

Floating Round to Single-Precision

frsp **fr**D,**fr**B (Rc=0)
frsp. **fr**D,**fr**B (Rc=1)

0 5	6 10	11 15	16 20	21 30	31
1 1 1 1 1 1	**fr**D	///	**fr**B	0 0 0 0 0 0 1 1 0 0	Rc

```
frD ← FPRoundtoSingle( (frB) )
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The floating-point operand in **fr**B is rounded to single-precision, using the rounding mode specified by FPSCR[RN], and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

If MSR[FP]=0, an attempt to execute **frsp[.]** causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN
- CR1 (if Rc=1)

frsqrte Floating-Point User frsqrte

Floating Reciprocal Square Root Estimate

frsqrte **fr**D,**fr**B (Rc=0)
frsqrte. **fr**D,**fr**B (Rc=1)

0 5	6 10	11 15	16 20	21 25	26 30	31
1 1 1 1 1 1	**fr**D	///	**fr**B	///	1 1 0 1 0	Rc

```
frD ← FPReciprocalSquareRootEstimate( (frB) )
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

A double-precision estimate of the reciprocal of the square root of the floating-point operand in **fr**B is placed into **fr**D. The estimate is correct to a precision of one part in 32 of the reciprocal of the square root of (**fr**B), that is,

$$\left| \frac{\left(\text{estimate} - \frac{1}{\sqrt{x}}\right)}{\frac{1}{\sqrt{x}}} \right| \le \frac{1}{32}$$

Here, x is the initial value in **fr**B. Note that the value placed into **fr**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the operand is summarized in Table 5-9.

Table 5-9. Operations with Special Values

Operand	Result	Exception
–∞	QNaN (No result if FPSCR[VE] = 1.)	VXSQRT
< 0	QNaN (No result if FPSCR[VE] = 1.)	VXSQRT
–0	–∞ (No result if FPSCR[ZE] = 1.)	ZX
+0	+∞ (No result if FPSCR[ZE] = 1.)	ZX
+∞	+0	None
SNaN	QNaN (No result if FPSCR[VE] = 1.)	VXSNAN
QNaN	QNaN	None

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1 and zero divide exceptions when FPSCR[ZE]=1.

If MSR[FP]=0, attempting to execute **frsqrte[.]** causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF   FR (undefined)   FI (undefined)
  FX   ZX   VXSNAN   VXSQRT
- CR1   (if Rc=1)

fsel fsel

Floating-Point	User

Floating Select

fsel **fr**D,**fr**A,**fr**C,**fr**B (Rc=0)
fsel. **fr**D,**fr**A,**fr**C,**fr**B (Rc=1)

0 5	6 10	11 15	16 20	21 25	26 30	31
1 1 1 1 1 1	**fr**D	**fr**A	**fr**B	**fr**C	1 0 1 1 1	Rc

```
if (frA) ≥ 0.0 then frD ← (frC)
else                frD ← (frB)
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The floating-point operand in **fr**A is compared to the value zero. If the operand is greater than or equal to zero, **fr**D is set to the contents of **fr**C. If the operand is less than zero or is a NaN, **fr**D is set to the contents of **fr**B. The comparison ignores the sign of zero (that is, +0 and –0 are regarded as equal).

If MSR[FP]=0, an attempt to execute **fsel**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- CR1 (if Rc=1)

NOTE: Software Considerations

Care must be taken in using **fsel** if IEEE-754 compatibility is required, or if the values being tested can be NaNs or infinities. Examples of uses of this instruction can be found in Section D.4, “Floating-Point Selection.”

fsub fsub

Floating-Point	User

Floating Subtract [Single]

fsub **fr**D,**fr**A,**fr**B (P=1, Rc=0)
fsub. **fr**D,**fr**A,**fr**B (P=1, Rc=1)
fsubs **fr**D,**fr**A,**fr**B (P=0, Rc=0)
fsubs. **fr**D,**fr**A,**fr**B (P=0, Rc=1)

0		2	3	4	5	6 10	11 15	16 20	21 25	26				30	31
1	1	1	P	1	1	**fr**D	**fr**A	**fr**B	///	1	0	1	0	0	Rc

```
if P=1 then frD ← (frA) $-_{dp}$ (frB)
else        frD ← (frA) $-_{sp}$ (frB)
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The floating-point operand in **fr**B is subtracted from the floating-point operand in **fr**A.

If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field, FPSCR[RN]. and placed into **fr**D.

The execution of the Floating Subtract instruction is identical to that of Floating Add, except that the contents of **fr**B participate in the operation with the sign bit (bit 0) inverted.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

If MSR[FP]=0, an attempt to execute **fsub**[**.**] or **fsubs**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPRF FR FI FX OX UX XX VXSNAN VXISI
- CR1 (if Rc=1)

icbi icbi

Base	User

Instruction Cache Block Invalidate

icbi **r**A,**r**B

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	///	**r**A	**r**B	1 1 1 1 0 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
InvalidateInstructionCacheBlock( EA )
```

If the block containing the byte addressed by EA is in memory that is memory-coherence required and a block containing the byte addressed by EA is in the instruction cache of any processors, the block is invalidated in those instruction caches, so that subsequent references cause the block to be fetched from main memory.

If the block containing the byte addressed by EA is in memory that is not memory-coherence required and a block containing the byte addressed by EA is in the instruction cache of this processor, the block is invalidated in that instruction cache, so that subsequent references cause the block to be fetched from main memory.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in memory that is write-through required or caching-inhibited.

This instruction is treated as a load for translation, permissions, and debug events. See Section 6.5.6.4, "Permission Control and Cache Management Instructions." It is implementation dependent whether read (R) or execute (X) permissions are required.

icbi may cause a cache-locking exception on some implementations. See the implementation documentation.

Other registers altered: None

NOTE: Software Considerations

Some processors may only apply this instruction to local caches and may have optional features in order to cause the operation to be broadcast to other processors. See the core reference manual.

icbiep

E.PD	Supervisor

icbiep

Instruction Cache Block Invalidate by External PID

icbiep **r**A,**r**B Form: X

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	**r**A	**r**B	1 1 1 1 0 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← a + (rB)
InvalidateInstructionCacheBlock( EA )
```

If the block containing the byte addressed by EA is in memory that is memory-coherence required and a block containing the byte addressed by EA is in the instruction cache of any processors, the block is invalidated in those instruction caches, so that subsequent references cause the block to be fetched from main memory.

If the block containing the byte addressed by EA is in memory that is not memory-coherence required and a block containing the byte addressed by EA is in the instruction cache of this processor, the block is invalidated in that instruction cache, so that subsequent references cause the block to be fetched from main memory.

The function of this instruction is independent of whether the block containing the byte addressed by EA is in memory that is write-through required or caching-inhibited.

This instruction is treated as a load for translation, permissions, and debug events. See Section 6.5.6.4, “Permission Control and Cache Management Instructions.” It is implementation dependent whether read (R) or execute (X) permissions are required.

icbi may cause a cache-locking exception on some implementations. See the implementation documentation.

This instruction is guest supervisor privileged.

For **icbiep**, the normal translation mechanism is not used. EPLC provides the context in which translation occurs. The following substitutions are made for translation and access control:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID] (GESR[EPID]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs.

Note: **icbiep** is identical to **icbi** except for using EPLC for translation context.

icblc

E.CL	User

Instruction Cache Block Lock Clear

icblc CT,**r**A,**r**B

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	CT	**r**A	**r**B	0 0 1 1 1 0 0 1 1 0	/

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← a + (rB)
InstructionCacheBlockClearLock(CT, EA)
```

The instruction cache specified by CT has the cache line corresponding to EA unlocked allowing the line to participate in the normal replacement policy.

Cache lock clear instructions remove locks previously set by cache lock set instructions.

Section 4.6.1.17.2, “User-Level Cache Management Instructions,” lists supported CT values. An implementation may use CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

The instruction is treated as a load for translation, permissions, and debug events and can cause DSI and DTLB error interrupts accordingly. It is implementation dependent whether read (R) or execute (X) permissions are required.

An unable-to-unlock condition is said to occur any of the following conditions exist:

- The target address is marked cache-inhibited, or the storage attributes of the address uses a coherency protocol that does not support locking.
- The target cache is disabled or not present.
- The CT field of the instructions contains a value not supported by the implementation.
- The target address is not in the cache or is present in the cache but is not locked.

If an unable-to-unlock condition occurs, no cache operation is performed.

icblc can only be executed by user-level software if MSR[UCLE] = 1.

icblc can only be executed by guest supervisor software if MSRP[UCLEP] = 0. <E.HV>

NOTE: Software Considerations

> Most caches contain a method for clearing all locks present in that cache through use of control and status registers. See Section 3.10.1, “L1 Cache Control and Status Register 0 (L1CSR0)” and Section 3.10.2, “L1 Cache Control and Status Register 1 (L1CSR1).”

icbt

Embedded	User

icbt

Instruction Cache Block Touch

icbt CT,**r**A,**r**B

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	CT	**r**A	**r**B	0 0 0 0 0 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← a + (rB)
PrefetchInstructionCacheBlock( CT, EA )
```

This instruction is a hint that performance would likely be improved if the block containing the byte addressed by EA is fetched into the instruction cache, because the program will probably soon execute code from the addressed location.

Section 4.6.1.17.2, “User-Level Cache Management Instructions,” lists supported CT values. An implementation may use other CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

The hint is ignored and no operation is performed if:

- The block addressed by EA is caching-inhibited or guarded
- The CT value is not supported by the implementation
- The access would cause a DTLB miss or a DSI
- The cache addressed by CT is disabled

The hint may be ignored for other implementation specific reasons. See the core reference manual. It is implementation dependent whether a hint that is ignored will cause debug events associated with the instruction.

This instruction is treated as a load for translation, permissions, and debug events (see Section 6.5.6.4, “Permission Control and Cache Management Instructions”), except that an interrupt is not taken for a translation or protection violation. It is implementation dependent whether read (R) or execute (X) permissions are required.

Other registers altered: None

icbtls

E.CL	User

icbtls

Instruction Cache Block Touch and Lock Set

icbtls CT,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	CT	**r**A	**r**B	0 1 1 1 1 0 0 1 1 0	/

```
if rA = 0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
PrefetchInstructionCacheBlockLockSet(CT, EA)
```

The instruction cache specified by CT has the cache line corresponding to EA loaded and locked into the cache. If the line already exists in the cache, it is locked without being refetched.

Section 4.6.1.17.2, “User-Level Cache Management Instructions,” lists supported CT values. An implementation may use CT values to enable software to target specific, implementation-dependent portions of its cache hierarchy or structure.

Cache touch and lock set instructions let software lock cache lines into the cache to provide lower latency for critical cache accesses and more deterministic behavior. Locked lines do not participate in the normal replacement policy when a line must be victimized for replacement.

This instruction is treated as a load for translation, permissions, and debug events (see Section 6.5.6.4, “Permission Control and Cache Management Instructions”), and can cause DSI and DTLB error interrupts accordingly. It is implementation dependent whether read (R) or execute (X) permissions are required.

An unable to lock condition is said to occur any of the following conditions exist:

- The target address is cache-inhibited, or the storage attributes of the address uses a coherency protocol that does not support locking.
- The target cache is disabled or not present.
- The CT field of the instructions contains a value not supported by the implementation.

If an unable to lock condition occurs and if the lock was targeted for the primary cache or secondary cache (CT = 0 or CT = 2), no cache operation is performed and LICSR1[ICUL] is set appropriately. Processors that implement **icblq.** do not set LICSR1[ICUL] when an unable to lock condition occurs and require software to query the status of the lock after attempting to lock it. Consult the core reference manual.

An overlocking condition is said to exist if all available ways for a given cache index are already locked and the locking instruction has no other exceptions. If an overlocking condition occurs for **icbtls** and if the lock was targeted for the primary cache or secondary cache (CT = 0 or CT = 2), L1CSR1[ICLO] (L2CSR0[L2LO] for CT = 2) is set. If lock overflow allocate is set (L1CSR1[ICLOA] for CT = 0 and (L2CSR0[L2LOA] for CT = 2), the line is loaded and locked replacing and unlocking an implementation dependent line in the set. Some processors do not set lock overflow status in either L1CSR1 or L2CSR0. Consult the core reference manual.

The results of overlocking and unable to lock conditions for caches other than the primary cache and secondary cache are defined as part of the integrated device.

icbtls can only be executed by user-level software if MSR[UCLE] = 1.

icbtls can only be executed by guest supervisor software if MSRP[UCLEP] = 0. <E.HV>

Other registers altered:

- L1CSR1[ICUL] if unable to lock occurs
- L1CSR1[ICLO] (L2CSR0[L2LO]) if lock overflow occurs

NOTE: Software Considerations

Locks on cache lines can be lost for a variety of reasons. SeeSection 6.3.1, "Cache Line Locking <E.CL>."

isel isel

Base	User

Integer Select

isel **r**D,**r**A,**r**B,**cr**b

0 5	6 10	11 15	16 20	21 25	26 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	**cr**b	0 1 1 1 1	/

```
if rA = 0 then a ← $^{64}0$ else a ← (rA)
c ← $cr_{crb + 32}$
if c then rD ← a
else      rD ← (rB)
```

If CR[**cr**b + 32] is set, the contents of **r**A|0 are copied into **r**D. If CR[**cr**b + 32] is clear, the contents of **r**B are copied into **r**D.

Simplified mnemonics: See Section C.10.8, “Integer Select (isel).”

isync

Base	User

isync

Instruction Synchronize

isync

0 5	6 20	21 30	31
0 1 0 0 1 1	///	0 0 1 0 0 1 0 1 1 0	/

isync provides an ordering function for the effects of all instructions executed by the processor executing the **isync** instruction. Executing an **isync** ensures that all instructions preceding the **isync** have completed before **isync** completes, and that no subsequent instructions are initiated until after **isync** completes. It also ensures that all instruction cache block invalidations caused by **icbi** and **icbiep** preceding the **isync** instruction have been performed with respect to the processor executing the **isync** instruction, and then causes any prefetched instructions to be discarded, with the effect that subsequent instructions are fetched and executed in the context established by the instructions preceding **isync**.

Except as described in the previous paragraph, the **isync** may complete before memory accesses associated with instructions preceding **isync** have been performed.

isync is context synchronizing. See Section 4.5.4.4, “Context Synchronization.”

Other registers altered: None

lbarx lbarx

ER	User

Load Byte And Reserve Indexed

lbarx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	rD	rA	rB	0 0 0 0 1 1 0 1 0 0	EH

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
RESERVE ← 1
RESERVE_LENGTH ← 1
RESERVE_ADDR ← real_addr(EA)
rD ← $^{56}0$ || MEM(EA,1)
```

EA is the sum of the contents of **r**A (zero if **r**A=0), and the contents of **r**B.

The byte addressed by EA is loaded into **r**D[56–63]; **r**D[0–55] are cleared.

lbarx creates a reservation for use by a **stbcx.** instruction. A real address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation. A length of 1 byte is associated with the reservation, and replaces any length previously associated with the reservation. See Section 4.6.1.16, “Atomic Update Primitives Using **lwarx** and **stwcx.**.”

If EA is in memory that is caching-inhibited or write-through required, it is implementation dependent whether this instruction executes normally, or whether a data storage interrupt occurs.

The value of EH provides a hint as to whether the program will perform a subsequent store to the byte in storage addressed by EA before some other processor attempts to modify it.

0	Other programs might attempt to modify the byte in memory addressed by EA regardless of the result of the corresponding **stbcx.** instruction.
1	Other programs will not attempt to modify the byte in memory addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

NOTE: Software Considerations

Some older processors may treat EH=1 as an illegal instruction.

Other registers altered: None

NOTE: Software Considerations

Because load and reserve instructions have implementation dependencies (such as the granularity at which reservations are managed), they must be used with care. System library programs should use these instructions to implement high-level synchronization functions (such as test and set, compare and swap) needed by application programs. Application programs should use these library programs, rather than use the load and reserve instructions directly.

NOTE: Software Considerations

Load and reserve when used with store conditional instructions provide atomic read-modify-write sequences when multiple processors are sharing memory. In general, such memory should be marked as memory coherence required.

lbdx lbdx

DS	User

Load Byte with Decoration Indexed

lbdx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	1 0 0 0 0 0 0 0 1 1	/

```
EA ← (rB)
rD ← $^{56}0$ || DECORATED_MEM(EA,1,(rA))
```

The byte addressed by EA with the decoration supplied by **r**A is loaded into **r**D[56–63]; **r**D[0–55] are cleared. The decoration specified in **r**A is sent to the device corresponding to EA. The behavior of the device with respect to the decoration is implementation specific. See the integrated device manual.

Other registers altered: None

NOTE: Software Considerations

Software must be acutely aware of the effects of operations defined by the decorations supplied and must ensure that the processor only issues operations appropriately. This generally requires addresses to be marked as guarded, caching inhibited and any appropriate memory barriers.

lbepx

E.PD	Supervisor

lbepx

Load Byte by External PID Indexed

lbepx **r**D**,r**A**,r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 0 0 1 0 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
rD ← $^{56}0$ || MEM(EA,1)
```

The byte addressed by EA is loaded into **r**D[56–63]; **r**D[0–55] are cleared.

This instruction is guest supervisor privileged.

For **lbepx**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value.
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID] (GESR[EPID]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

lbz

Base	User

lbz

Load Byte and Zero

lbz **r**D,D(**r**A)

0 5	6 10	11 15	16 31
1 0 0 0 1 0	**r**D	**r**A	D

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + EXTS(D)
rD ← $^{56}0$ || MEM(EA,1)
```

The byte addressed by EA is loaded into **r**D[56–63]; **r**D[0–55] are cleared.

Other registers altered: None

lbzu lbzu

Base	User

Load Byte and Zero with Update

lbzu **r**D,D(**r**A)

0 5	6 10	11 15	16 31
1 0 0 0 1 1	**r**D	**r**A	D

```
a ← (rA)
EA ← a + EXTS(D)
rD ← $^{56}0$ || MEM(EA,1)
rA ← EA
```

The byte addressed by EA is loaded into **r**D[56–63]; **r**D[0–55] are cleared.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered: None

lbzux

Base	User

Load Byte and Zero with Update Indexed

lbzux **r**D,**r**A,**r**B

0 5	6 10	11 15	16	30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 0 0 1 1 1 0 1 1 1	/

```
EA ← (rA) + (rB)
rD ← $^{56}0$ || MEM(EA,1)
rA ← EA
```

The byte addressed by EA is loaded into **r**D[56–63]; **r**D[0–55] are cleared.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered: None

lbzx lbzx

Base	User

Load Byte and Zero Indexed

lbzx **r**D,**r**A,**r**B

0 5	6 10	11 15	16	30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 0 0 1 0 1 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
rD ← $^{56}0$ || MEM(EA,1)
```

The byte addressed by EA is loaded into **r**D[56–63]; **r**D[0–55] are cleared.

Other registers altered: None

ld

64	User

ld

Load Doubleword

ld **r**D,D(**r**A)

0 5	6 10	11 15	16 29	30	31
1 1 1 0 1 0	**r**D	**r**A	DS	0	0

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + EXTS(DS || 0b00)
rD ← MEM(EA,8)
```

The doubleword addressed by EA is loaded into **r**D.

Other registers altered: None

NOTE: Software Considerations

D as specified in the assembly syntax is transformed by the assembler when encoded into the instruction such that DS is formed. The specification of the instruction limits the D field of the instruction to be divisible by 4.

ldarx ldarx

64	User

Load Doubleword And Reserve Indexed

ldarx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 0 0 1 0 1 0 1 0 0	EH

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
RESERVE ← 1
RESERVE_LENGTH ← 8
RESERVE_ADDR ← real_addr(EA)
rD ← MEM(EA,8)
```

EA is the sum of the contents of **r**A (zero if **r**A=0), and the contents of **r**B.

The word addressed by EA is loaded into **r**D[0–63].

ldarx creates a reservation for use by a **stdcx.** instruction. A real address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation. A length of 8 bytes is associated with the reservation, and replaces any length previously associated with the reservation. See Section 4.6.1.16, “Atomic Update Primitives Using **lwarx** and **stwcx.**.”

If EA is in memory that is caching-inhibited or write-through required, it is implementation dependent whether this instruction executes normally, or whether a data storage interrupt occurs.

EA must be a multiple of 8. If it is not, an alignment exception occurs.

The value of EH provides a hint as to whether the program will perform a subsequent store to the byte in storage addressed by EA before some other processor attempts to modify it.

0	Other programs might attempt to modify the byte in memory addressed by EA regardless of the result of the corresponding **stdcx.** instruction.
1	Other programs will not attempt to modify the byte in memory addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

NOTE: Software Considerations

Some older processors may treat EH=1 as an illegal instruction.

Other registers altered: None

NOTE: Software Considerations

Because load and reserve instructions have implementation dependencies (such as the granularity at which reservations are managed), they must be used with care. System library programs should use these instructions to implement high-level synchronization functions (such as test and set, compare and swap) needed by application programs. Application programs should use these library programs, rather than use the load and reserve instructions directly.

NOTE: Software Considerations

Load and reserve when used with store conditional instructions provide atomic read-modify-write sequences when multiple processors are sharing memory. In general, such memory should be marked as memory coherence required.

ldbrx ldbrx

64	User

Load Doubleword Byte-Reversed Indexed

ldbrx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	1 0 0 0 0 1 0 1 0 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
$data_{0:63}$ ← MEM(EA,8)
rD ←   $data_{56:63}$ || $data_{48:55}$
      || $data_{40:47}$ || $data_{32:39}$
      || $data_{24:31}$ || $data_{16:23}$
      || $data_{8:15}$  || $data_{0:7}$
```

Bits 0–7 of the doubleword addressed by EA are loaded into **r**D[56–63]. Bits 8–15 of the doubleword addressed by EA are loaded into **r**D[48–55]. Bits 16–23 of the doubleword addressed by EA are loaded into **r**D[40–47]. Bits 24–31 of the doubleword addressed by EA are loaded into **r**D[32–39]. Bits 32–39 of the doubleword addressed by EA are loaded into **r**D[24–31]. Bits 40–47 of the doubleword addressed by EA are loaded into **r**D[16–23]. Bits 48–55 of the doubleword addressed by EA are loaded into **r**D[8–15]. Bits 56–63 of the doubleword addressed by EA are loaded into **r**D[0–7].

Other registers altered: None

NOTE: Software Considerations

When EA references big-endian memory, these instructions have the effect of loading data in little-endian byte order. Likewise, when EA references little-endian memory, these instructions have the effect of loading data in big-endian byte order.

In some implementations, the Load Doubleword Byte-Reverse Indexed instructions may have greater latency than other load instructions.

lddx lddx

DS, 64	User

Load Doubleword with Decoration Indexed

lddx **r**D**,r**A**,r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	1 0 0 1 1 0 0 0 1 1	/

```
EA ← (rB)
rD ← DECORATED_MEM(EA,8,(rA))
```

The doubleword addressed by EA with the decoration supplied by **r**A is loaded into **r**D[0–63]. The decoration specified in **r**A is sent to the device corresponding to EA. The behavior of the device with respect to the decoration is implementation specific. See the integrated device manual.

Other registers altered: None

NOTE: Software Considerations

Software must be acutely aware of the effects of operations defined by the decorations supplied and must ensure that the processor only issues operations appropriately. This generally requires addresses to be marked as guarded, caching inhibited and any appropriate memory barriers.

NOTE: Software Considerations

Software should ensure that accesses are aligned, otherwise atomic accesses are not guaranteed and may require multiple accesses to perform the operation which is not likely to produce the intended result.

ldepx

E.PD, 64	Supervisor

ldepx

Load Doubleword by External PID Indexed

ldepx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 0 0 0 0 1 1 1 0 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
rD ← MEM(EA,8)
```

The doubleword addressed by EA is loaded into **r**D.

This instruction is guest supervisor privileged.

For **ldepx**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value.
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID] (GESR[EPID]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

ldu ldu

64	User

Load Doubleword with Update

ldu **r**D,D(**r**A)

0 5	6 10	11 15	16 29	30	31
1 1 1 0 1 0	**r**D	**r**A	DS	0	1

```
EA ← (rA) + EXTS(DS || 0b00)
rD ← MEM(EA,8)
rA ← EA
```

The doubleword addressed by EA is loaded into **r**D.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered: None

NOTE: Software Considerations

D as specified in the assembly syntax is transformed by the assembler when encoded into the instruction such that DS is formed. The specification of the instruction limits the D field of the instruction to be divisible by 4.

ldux ldux

64	User

Load Doubleword with Update Indexed

ldux **r**D**,r**A,**r**B

0 5	6 10	11 15	16	29 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 0 0 0 1 1 0 1 0 1	/

```
EA ← (rA) + (rB)
rD ← MEM(EA,8)
rA ← EA
```

The doubleword addressed by EA is loaded into **r**D.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered: None

ldx

64	User

ldx

Load Doubleword Indexed

ldx **r**D,**r**A,**r**B

0 5	6 10	11 15	16	30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 0 0 0 0 1 0 1 0 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
rD ← MEM(EA,8)
```

The doubleword addressed by EA is loaded into **r**D.

Other registers altered: None

lfd

FP	user

lfd

Load Floating-Point Double

lfd **fr**D,D**(r**A)

0 5	6 10	11 15	16 31
1 1 0 0 1 0	**fr**D	**r**A	D

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + EXTS(D)
frD ← MEM(EA,8)
```

The doubleword addressed by EA is loaded into **fr**D.

If MSR[FP]=0, an attempt to execute **lfd** causes a floating-point unavailable interrupt.

lfddx

DS, FP	User

lfddx

Load Floating-Point Double with Decoration Indexed

lfddx **fr**D**,r**A**,r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**fr**D	**r**A	**r**B	1 1 0 0 1 0 0 0 1 1	/

```
EA ← (rB)
frD ← DECORATED_MEM(EA,8,(rA))
```

The doubleword addressed by EA with the decoration supplied by **r**A is loaded into **fr**D. The decoration specified in **r**B is sent to the device corresponding to EA. The behavior of the device with respect to the decoration is implementation specific. See the integrated device manual.

If MSR[FP]=0, an attempt to execute **lfddx** causes a floating-point unavailable interrupt.

Other registers altered: None

NOTE: Software Considerations

Software must be acutely aware of the effects of operations defined by the decorations supplied and must ensure that the processor only issues operations appropriately. This generally requires addresses to be marked as guarded, caching inhibited and any appropriate memory barriers.

NOTE: Software Considerations

Software should ensure that accesses are aligned, otherwise atomic accesses are not guaranteed and may require multiple accesses to perform the operation which is not likely to produce the intended result.

lfdepx lfdepx

E.PD, FP	Supervisor

Load Floating-Point Double by External PID Indexed

lfdepx **fr**D**,r**A**,r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**fr**D	**r**A	**r**B	1 0 0 1 0 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
frD ← MEM(EA,8)
```

The doubleword addressed by EA is loaded into **fr**D.

This instruction is guest supervisor privileged.

If MSR[FP]=0, an attempt to execute **lfdepx** causes a floating-point unavailable interrupt.

For **lfdepx**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value.
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID,FP] (GESR[EPID,FP]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

lfdu lfdu

FP	user

Load Floating-Point Double with Update

lfdu **fr**D**,**D**(r**A)

0 5	6 10	11 15	16 31
1 1 0 0 1 1	**fr**D	**r**A	D

```
EA ← (rA) + EXTS(D)
frD ← MEM(EA,8)
rA ← EA
```

The doubleword addressed by EA is loaded into **fr**D.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

If MSR[FP]=0, an attempt to execute **lfdu** causes a floating-point unavailable interrupt.

lfdux lfdux

FP	user

Load Floating-Point Double with Update Indexed

lfdux **fr**D**,r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**fr**D	**r**A	**r**B	1 0 0 1 1 1 0 1 1 1	/

```
EA ← (rA) + (rB)
frD ← MEM(EA,8)
rA ← EA
```

The doubleword addressed by EA is loaded into **fr**D.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

If MSR[FP]=0, an attempt to execute **lfdux** causes a floating-point unavailable interrupt.

lfdx

FP	user

lfdx

Load Floating-Point Double Indexed

lfdx **fr**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**fr**D	**r**A	**r**B	1 0 0 1 0 1 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
frD ← MEM(EA,8)
```

The doubleword addressed by EA is loaded into **fr**D.

If MSR[FP]=0, an attempt to execute **lfdx** causes a floating-point unavailable interrupt.

lfs lfs

FP	user

Load Floating-Point Single

lfs **fr**D,D(**r**A)

0 5	6 10	11 15	16 31
1 1 0 0 0 0	**fr**D	**r**A	D

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + EXTS(D)
frD ← DOUBLE(MEM(EA,8))
```

The word addressed by EA is interpreted as a single-precision operand, converted to floating-point double format, and loaded into **fr**D.

If MSR[FP]=0, an attempt to execute **lfs** causes a floating-point unavailable interrupt.

lfsu lfsu

FP	user

Load Floating-Point Single with Update

lfsu **fr**D,D(**r**A)

0 5	6 10	11 15	16 31
1 1 0 0 0 1	**fr**D	**r**A	D

```
EA ← (rA) + EXTS(D)
frD ← DOUBLE(MEM(EA,8))
rA ← EA
```

The word addressed by EA is interpreted as a single-precision operand, converted to floating-point double format, and loaded into **fr**D.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

If MSR[FP]=0, an attempt to execute **lfsu** causes a floating-point unavailable interrupt.

lfsux

FP	user

lfsux

Load Floating-Point Single with Update Indexed

lfsux **fr**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**fr**D	**r**A	**r**B	1 0 0 0 1 1 0 1 1 1	/

```
EA ← (rA) + (rB)
frD ← DOUBLE(MEM(EA,8))
rA ← EA
```

The word addressed by EA is interpreted as a single-precision operand, converted to floating-point double format, and loaded into **fr**D.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

If MSR[FP]=0, an attempt to execute **lfsux** causes a floating-point unavailable interrupt.

lfsx lfsx

FP	user

Load Floating-Point Single Indexed

lfsx **fr**D**,r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**fr**D	**r**A	**r**B	1 0 0 0 0 1 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
frD ← DOUBLE(MEM(EA,8))
```

The word addressed by EA is interpreted as a single-precision operand, converted to floating-point double format, and loaded into **fr**D.

If MSR[FP]=0, an attempt to execute **lfsx** causes a floating-point unavailable interrupt.

lha

Base	User

Load Half Word Algebraic

lha **r**D,D(**r**A)

0 5	6 10	11 15	16 31
1 0 1 0 1 0	**r**D	**r**A	D

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + EXTS(D))
rD ← EXTS(MEM(EA,2))
```

The half word addressed by EA is loaded into **r**D[48–63]. **r**D[0–47] are filled with a copy of the high-order bit of the loaded half word.

Other registers altered: None

lhau

Base	User

Load Half Word Algebraic with Update

lhau **r**D,D(**r**A)

0 5	6 10	11 15	16 31
1 0 1 0 1 1	**r**D	**r**A	D

```
EA ← ((rA) + EXTS(D))
rD ← EXTS(MEM(EA,2))
rA ← EA
```

The half word addressed by EA is loaded into **r**D[48–63]. **r**D[0–47] are filled with a copy of the high-order bit of the loaded half word.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered: None

lhaux lhaux

Base	User

Load Half Word Algebraic with Update Indexed

lhaux **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 1 0 1 1 1 0 1 1 1	/

```
EA ← ((rA) + (rB))
rD ← EXTS(MEM(EA,2))
rA ← EA
```

The half word addressed by EA is loaded into **r**D[48–63]. **r**D[0–47] are filled with a copy of the high-order bit of the loaded half word.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered: None

lhax lhax

Base	User

Load Half Word Algebraic Indexed

lhax **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 1 0 1 0 1 0 1 1 1	/

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← (a + (rB))
rD ← EXTS(MEM(EA,2))
```

The half word addressed by EA is loaded into **r**D[48–63]. **r**D[0–47] are filled with a copy of the high-order bit of the loaded half word.

Other registers altered: None

lharx

ER	User

lharx

Load Halfword And Reserve Indexed

lharx r**D**,r**A**,r**B**

0					5	6 10	11 15	16 20	21									30	31
0	1	1	1	1	1	rD	rA	rB	0	0	0	1	1	1	0	1	0	0	EH

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
RESERVE ← 1
RESERVE_LENGTH ← 2
RESERVE_ADDR ← real_addr(EA)
rD ← $^{48}0$ || MEM(EA,2)
```

EA is the sum of the contents of **r**A (zero if **r**A=0), and the contents of **r**B.

The halfword addressed by EA is loaded into **r**D[48–63]; **r**D[0–47] are cleared.

lharx creates a reservation for use by a **sthcx.** instruction. A real address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation. A length of 2 bytes is associated with the reservation, and replaces any length previously associated with the reservation. See Section 4.6.1.16, "Atomic Update Primitives Using **lwarx** and **stwcx.**."

If EA is in memory that is caching-inhibited or write-through required, it is implementation dependent whether this instruction executes normally, or whether a data storage interrupt occurs.

EA must be a multiple of 2. If it is not, an alignment exception occurs.

The value of EH provides a hint as to whether the program will perform a subsequent store to the byte in storage addressed by EA before some other processor attempts to modify it.

0 Other programs might attempt to modify the byte in memory addressed by EA regardless of the result of the corresponding **sthcx.** instruction.

1 Other programs will not attempt to modify the byte in memory addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

NOTE: Software Considerations

Some older processors may treat EH=1 as an illegal instruction.

Other registers altered: None

NOTE: Software Considerations

Because load and reserve instructions have implementation dependencies (such as the granularity at which reservations are managed), they must be used with care. System library programs should use these instructions to implement high-level synchronization functions (such as test and set, compare and swap) needed by application programs. Application programs should use these library programs, rather than use the load and reserve instructions directly.

NOTE: Software Considerations

Load and reserve when used with store conditional instructions provide atomic read-modify-write sequences when multiple processors are sharing memory. In general, such memory should be marked as memory coherence required.

lhbrx lhbrx

Base	User

Load Half Word Byte-Reverse Indexed

lhbrx **r**D,**r**A,**r**B

0 … 5	6 … 10	11 … 15	16 … 20	21 … 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	1 1 0 0 0 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
$data_{0:15}$ ← MEM(EA,2)
rD ← $^{48}0$ || $data_{8:15}$ || $data_{0:7}$
```

Bits 0–7 of the half word addressed by EA are loaded into **r**D[56–63]. Bits 8–15 of the half word addressed by EA are loaded into **r**D[48–55]; **r**D[0–47] are cleared.

Other registers altered: None

NOTE: Software Considerations

When EA references big-endian memory, these instructions have the effect of loading data in little-endian byte order. Likewise, when EA references little-endian memory, these instructions have the effect of loading data in big-endian byte order.

NOTE: Software Considerations

In some implementations, the Load Half Word Byte-Reverse Indexed instructions may have greater latency than other load instructions.

lhdx

DS	User

lhdx

Load Halfword with Decoration Indexed

lhdx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	1 0 0 0 1 0 0 0 1 1	/

```
EA ← (rB)
rD ← $^{48}0$ || DECORATED_MEM(EA,2,(rA))
```

The halfword addressed by EA with the decoration supplied by **r**A is loaded into **r**D[48–63]; **r**D[0–47] are cleared. The decoration specified in **r**A is sent to the device corresponding to EA. The behavior of the device with respect to the decoration is implementation specific. See the integrated device manual.

Other registers altered: None

NOTE: Software Considerations

Software must be acutely aware of the effects of operations defined by the decorations supplied and must ensure that the processor only issues operations appropriately. This generally requires addresses to be marked as guarded, caching inhibited and any appropriate memory barriers.

NOTE: Software Considerations

Software should ensure that accesses are aligned, otherwise atomic accesses are not guaranteed and may require multiple accesses to perform the operation which is not likely to produce the intended result.

lhepx

E.PD	Supervisor

lhepx

Load Halfword by External PID Indexed

lhepx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 1 0 0 0 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
rD ← $^{48}0$ || MEM(EA,2)
```

The halfword addressed by EA is loaded into **r**D[48–63]; **r**D[0–47] are cleared.

This instruction is guest supervisor privileged.

For **lhepx**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value.
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID] (GESR[EPID]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

lhz | lhz

Base	User

Load Half Word and Zero

lhz **r**D,D(**r**A)

0 5	6 10	11 15	16 31
1 0 1 0 0 0	**r**D	**r**A	D

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + EXTS(D))
rD ← $^{48}0$ || MEM(EA,2)
```

The half word addressed by EA is loaded into **r**D[48–63]. **r**D[0–47] are cleared.

Other registers altered: None

lhzu

Base	User

lhzu

Load Half Word and Zero with Update

lhzu **r**D,D(**r**A)

0–5	6–10	11–15	16–31
1 0 1 0 0 1	**r**D	**r**A	D

```
EA ← ((rA) + EXTS(D))
rD ← $^{48}0$ || MEM(EA,2)
rA ← EA
```

The half word addressed by EA is loaded into **r**D[48–63]. **r**D[0–47] are cleared.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered: None

lhzux

Base	User

lhzux

Load Half Word and Zero with Update Indexed

lhzux rD**,**rA**,**rB

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 1 0 0 1 1 0 1 1 1	/

```
EA ← ((rA) + (rB))
rD ← $^{48}$0 || MEM(EA,2)
rA ← EA
```

The half word addressed by EA is loaded into **r**D[48–63]. **r**D[0–47] are cleared.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered: None

lhzx lhzx

Base	User

Load Half Word and Zero Indexed

lhzux **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 1 0 0 0 1 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + (rB))
rD ← $^{48}0$ || MEM(EA,2)
```

The half word addressed by EA is loaded into **r**D[48–63]. **r**D[0–47] are cleared.

Other registers altered: None

lmw lmw

Base	User

Load Multiple Word

lmw **r**D,D(**r**A)

0 5	6 10	11 15	16 31
1 0 1 1 1 0	**r**D	**r**A	D

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + EXTS(D))
r ← rD
do while r ≤ 31
   GPR(r) ← $^{32}0$ ‖ MEM(EA,4)
   r  ← r + 1
   EA ← EA + 4
```

Here n = (32 – **r**D). n consecutive words starting at EA are loaded into bits 32–63 of registers **r**D through GPR31. Bits 0–31 of these GPRs are cleared.

EA must be a multiple of 4. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

If **r**A is in the range of registers to be loaded, including the case in which **r**A=0, the instruction form is invalid.

Other registers altered: None

lvepx

E.PD, V	Supervisor

lvepx

Load Vector by External PID Indexed

lvepx **v**D**,r**A**,r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**v**D	**r**A	**r**B	0 1 0 0 1 0 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
vD ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0,16)
```

The quadword addressed by EA truncated to the nearest quadword boundary is loaded into **v**D. The data is loaded in big-endian form regardless of the setting of the E storage attribute.

This instruction is guest supervisor privileged.

An attempt to execute **lvepx** while MSR[SPV] = 0 causes an AltiVec unavailable interrupt.

For **lvepx**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value.
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID] (GESR[EPID]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

lvepxl lvepxl

E.PD, V	Supervisor

Load Vector by External PID Indexed LRU

lvepxl **v**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**v**D	**r**A	**r**B	0 1 0 0 0 0 0 1 1 1	/

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← a + (rB)
vD ← MEM(EA & 0xFFFF_FFFF_FFFF_FFF0,16)
```

The quadword addressed by EA truncated to the nearest quadword boundary is loaded into **v**D. The data is loaded in big-endian form regardless of the setting of the E storage attribute.

The **lvepxl** instruction provides a hint that the quad word addressed by EA will probably not be needed again by the program in the near future.

Note that on some implementations, the hint provided by the **lvepxl** instruction and the corresponding hint provided by the **stvepxl** instruction are applied to the entire cache block containing the specified quadword. On such implementations, the effect of the hint may be to cause that cache block to be considered a likely candidate for reuse when space is needed in the cache for a new block. Thus, on such implementations, the hint should be used with caution if the cache block containing the quadword also contains data that may be needed by the program in the near future. Also, the hint may be used before the last reference in a sequence of references to the quadword if the subsequent references are likely to occur sufficiently soon that the cache block containing the quadword is not likely to be displaced from the cache before the last reference.

This instruction is guest supervisor privileged.

An attempt to execute **lvepxl** while MSR[SPV] = 0 causes an AltiVec unavailable interrupt.

For **lvepxl**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value.
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID] (GESR[EPID]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

lwa lwa

64	User

Load Word Algebraic

lwa **r**D**,**DS**(r**A**)**
)

0 5	6 10	11 15	16 29	30	31
1 1 1 0 1 0	**r**D	**r**A	DS	1	0

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← (a + EXTS(DS || 0b00))
rD ← EXTS(MEM(EA,4))
```

The word addressed by EA is loaded into **r**D[32–63]. **r**D[0–31] are filled with a copy of the high-order bit of the loaded word.

Other registers altered: None

NOTE: Software Considerations

D as specified in the assembly syntax is transformed by the assembler when encoded into the instruction such that DS is formed. The specification of the instruction limits the D field of the instruction to be divisible by 4.

lwarx

Base	User

lwarx

Load Word And Reserve Indexed

lwarx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	rD	rA	rB	0 0 0 0 0 1 0 1 0 0	EH

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
RESERVE ← 1
RESERVE_LENGTH ← 4
RESERVE_ADDR ← real_addr(EA)
rD ← $^{32}0$ || MEM(EA,4)
```

EA is the sum of the contents of **r**A (zero if **r**A=0), and the contents of **r**B.

The word addressed by EA is loaded into **r**D[32–63]; **r**D[0–31] are cleared.

lwarx creates a reservation for use by a **stwcx.** instruction. A real address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation. A length of 4 bytes is associated with the reservation, and replaces any length previously associated with the reservation. See Section 4.6.1.16, “Atomic Update Primitives Using **lwarx** and **stwcx.**.”

If EA is in memory that is caching-inhibited or write-through required, it is implementation dependent whether this instruction executes normally, or whether a data storage interrupt occurs.

EA must be a multiple of 4. If it is not, an alignment exception occurs.

The value of EH provides a hint as to whether the program will perform a subsequent store to the byte in storage addressed by EA before some other processor attempts to modify it.

0 Other programs might attempt to modify the byte in memory addressed by EA regardless of the result of the corresponding **stwcx.** instruction.

1 Other programs will not attempt to modify the byte in memory addressed by EA until the program that has acquired the lock performs a subsequent store releasing the lock.

NOTE: Software Considerations

Some older processors may treat EH=1 as an illegal instruction.

Other registers altered: None

NOTE: Software Considerations

Because load and reserve instructions have implementation dependencies (such as the granularity at which reservations are managed), they must be used with care. System library programs should use these instructions to implement high-level synchronization functions (such as test and set, compare and swap) needed by application programs. Application programs should use these library programs, rather than use the load and reserve instructions directly.

NOTE: Software Considerations

Load and reserve when used with store conditional instructions provide atomic read-modify-write sequences when multiple processors are sharing memory. In general, such memory should be marked as memory coherence required.

lwaux lwaux

64	User

Load Word Algebraic with Update Indexed

lwaux **r**D**,r**A**,r**B
)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 1 0 1 1 1 0 1 0 1	/

```
EA ← ((rA) + (rB))
rD ← EXTS(MEM(EA,4))
rA ← EA
```

The word addressed by EA is loaded into **r**D[32–63]. **r**D[0–31] are filled with a copy of the high-order bit of the loaded word.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered: None

lwax

Base	User

lwax

Load Word Algebraic Indexed

lwax **r**D,**r**A,**r**B
)

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 1 0 1 0 1 0 1 0 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + (rB))
rD ← EXTS(MEM(EA,4))
```

The word addressed by EA is loaded into **r**D[32–63]. **r**D[0–31] are filled with a copy of the high-order bit of the loaded word.

Other registers altered: None

lwbrx lwbrx

Base	User

Load Word Byte-Reverse Indexed

lwbrx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	1 0 0 0 0 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← a + (rB)
data$_{0:15}$ ← MEM(EA,4)
rD ← $^{32}$0 || data$_{24:31}$ || data$_{16:23}$ || data$_{8:15}$ || data$_{0:7}$
```

Bits 0–7 of the word addressed by EA are loaded into **r**D[56–63]. Bits 8–15 of the word addressed by EA are loaded into **r**D[48–55]. Bits 16–23 of the word addressed by EA are loaded into **r**D[40–47]. Bits 24–31 of the word addressed by EA are loaded into **r**D[32–39]. Bits **r**D[0–31] are cleared.

Other registers altered: None

NOTE: Software Considerations

When EA references big-endian memory, these instructions have the effect of loading data in little-endian byte order. Likewise, when EA references little-endian memory, these instructions have the effect of loading data in big-endian byte order.

NOTE: Software Considerations

In some implementations, the Load Word Byte-Reverse Indexed instructions may have greater latency than other load instructions.

lwdx

DS	User

Load Word with Decoration Indexed

lwdx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	1 0 0 1 0 0 0 0 1 1	/

```
EA ← (rB)
rD ← $^{32}0$ || DECORATED_MEM(EA,4,(rA))
```

The word addressed by EA with the decoration supplied by **r**A is loaded into **r**D[32–63]; **r**D[0–31] are cleared. The decoration specified in **r**A is sent to the device corresponding to EA. The behavior of the device with respect to the decoration is implementation specific. See the integrated device manual.

Other registers altered: None

NOTE: Software Considerations

Software must be acutely aware of the effects of operations defined by the decorations supplied and must ensure that the processor only issues operations appropriately. This generally requires addresses to be marked as guarded, caching inhibited and any appropriate memory barriers.

NOTE: Software Considerations

Software should ensure that accesses are aligned, otherwise atomic accesses are not guaranteed and may require multiple accesses to perform the operation which is not likely to produce the intended result.

lwepx

E.PD	Supervisor

lwepx

Load Word by External PID Indexed

lwepx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 0 0 0 0 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
rD ← $^{32}0$ || MEM(EA,4)
```

The word addressed by EA is loaded into **r**D[32–63]; **r**D[0–31] are cleared.

This instruction is guest supervisor privileged.

For **lwepx**, the normal translation mechanism is not used. The EPLC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPLC[EPR] is used in place of MSR[PR]
- EPLC[EAS] is used in place of MSR[DS]
- EPLC[EPID] is used in place of the PID value.
- EPLC[EGS] is used in place of MSR[GS] <E.HV>
- EPLC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID] (GESR[EPID]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

lwz

Base	User

lwz

Load Word and Zero

lwz **r**D,D(**r**A)

0–5	6–10	11–15	16–31
1 0 0 0 0 0	**r**D	**r**A	D

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← (a + EXTS(D))
rD ← $^{48}$0 || MEM(EA,4)
```

The word addressed by EA is loaded into **r**D[32–63]. **r**D[0–31] are cleared.

Other registers altered: None

lwzu

Base	User

lwzu

Load Word and Zero and Update

lwzu **r**D,D(**r**A)

0 5	6 10	11 15	16 31
1 0 0 0 0 1	**r**D	**r**A	D

```
EA ← ((rA) + EXTS(D))
rD ← $^{48}0$ || MEM(EA,4)
rA ← EA
```

The word addressed by EA is loaded into **r**D[32–63]. **r**D[0–31] are cleared.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered: None

lwzux

Base	User

lwzux

Load Word and Zero with Update Indexed

lwzux **r**D**,r**A**,r**B
)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 0 0 0 1 1 0 1 1 1	/

```
EA ← ((rA) + (rB))
rD ← $^{48}0$ || MEM(EA,4)
rA ← EA
```

The word addressed by EA is loaded into **r**D[32–63]. **r**D[0–31] are cleared.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered: None

lwzx

Base	User

lwzx

Load Word and Zero Indexed

lwzx **r**D,**r**A,**r**B
)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	0 0 0 0 0 1 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + (rB))
rD ← $^{48}0$ || MEM(EA,4)
```

The word addressed by EA is loaded into **r**D[32–63]. **r**D[0–31] are cleared.

Other registers altered: None

mbar mbar

Embedded	User

Memory Barrier

mbar MO

0 5	6 10	11 20	21 30	31
0 1 1 1 1 1	MO	///	1 1 0 1 0 1 0 1 1 0	/

When MO=0, **mbar** creates a cumulative memory barrier, which provides a memory ordering function for all memory access instructions executed by the processor executing the **mbar** instruction. Executing an **mbar** instruction ensures that all data memory accesses caused by instructions preceding the **mbar** have completed before any data memory accesses caused by any instructions after the **mbar**. This order is seen by all mechanisms.

When **mbar** MO = 1, **mbar** functions like **eieio** as it is defined by the original PowerPC architecture. It provides ordering for the effects of load and store instructions. These instructions consist of two sets, which are ordered separately. Memory accesses caused by a **dcbz**, **dcbzl**, **dcbzep**, **dcbzlep**, **dcba**, or **dcbal** are ordered like a store. The two sets follow:

- Caching-inhibited, guarded loads and stores to memory and write-through-required stores to memory. **mbar** (MO=1) controls the order in which accesses are performed in main memory. It ensures that all applicable memory accesses caused by instructions preceding the **mbar** have completed with respect to main memory before any applicable memory accesses caused by instructions following **mbar** access main memory. It acts like a barrier that flows through the memory queues and to main memory, preventing the reordering of memory accesses across the barrier. No ordering is performed for **dcbz**, **dcbzl**, **dcbzep** or **dcbzlep**, if the instruction causes the system alignment error handler to be invoked.

  All accesses in this set are ordered as one set; there is not one order for guarded, caching-inhibited loads and stores and another for write-through-required stores.
- Stores to memory that are caching-allowed, write-through not required, and memory-coherency required. **mbar** (MO=1) controls the order in which accesses are performed with respect to coherent memory. It ensures that, with respect to coherent memory, applicable stores caused by instructions before the **mbar** complete before any applicable stores caused by instructions after it.

Except for **dcbz**, **dcbzl**, **dcbzep**, **dcbzlep**, **dcba**, and **dcbal**, **mbar** (MO=1) does not affect the order of cache operations (whether caused explicitly by a cache management instruction or implicitly by the cache coherency mechanism). Also. **mbar** does not affect the order of accesses in one set with respect to accesses in the other.

mbar (MO=1) may complete before memory accesses caused by instructions preceding it have been performed with respect to main memory or coherent memory as appropriate. **mbar** (MO=1) is intended for use in managing shared data structures, in accessing memory-mapped I/O, and in preventing load/store combining operations in main memory. For the first use, the shared data structure and the lock that protects it must be altered only by stores that are in the same set (for both cases described above). For the second use, **mbar** (MO=1) can be thought of as placing a barrier into the stream of memory accesses issued by a core, such that any given memory access appears to be on the same side of the barrier to both the core and the I/O device.

If MO is not 0 or 1, an implementation may support the **mbar** instruction ordering a particular subset of memory accesses. An implementation may also support multiple, non-zero values of MO that each specify a different subset of memory accesses that are ordered by the **mbar** instruction. Which subsets of memory accesses are ordered and which values of MO specify these subsets is implementation-dependent. See the user’s manual for the implementation.

Some older implementations define an HID0[ABE] field that is used to enable some specific symmetrical multiprocessing capabilities; see the core reference manual for more information.

Other registers altered: None

NOTE: Software Considerations

mbar is provided to implement a pipelined memory barrier. The following sequence shows one use of **mbar** in supporting shared data, ensuring the action is completed before releasing the lock.

```
P1              P2
lock            . . .
mbar            . . .
read & write    . . .
mbar            . . .
free lock       . . .
. . .           lock
. . .           mbar
. . .           read & write
. . .           mbar
```

NOTE: Software Considerations

- The **mbar** instruction is intended for use in doing memory-mapped I/O, and in preventing load/store combining operations in main storage. See Section 6.2.1, “Storage Attributes and Coherency.”
- Because loads, stores, and loads followed by stores to locations that are both caching inhibited and guarded are performed in program order (see Section 6.4.8.1, “Memory Access Ordering”), **mbar** is needed for such storage only when loads must be ordered with respect to previous stores.

NOTE: Software Considerations

- The functions provided by **mbar** MO=0 are a strict subset of those provided by **sync** with L=0.
- Because **eieio** (from the PowerISA Server environment) and **mbar** share the same op-code, software designed for both server and embedded environments must assume that only the **eieio** functionality applies since the functions provided by **eieio** are a subset of those provided by **mbar**.

mcrf mcrf

Base	User

Move Condition Register Field

mcrf **crfD,cr**S

0 5	6 8	9 10	11 13	14 20	21 30	31
0 1 0 0 1 1	**crfD**	//	**cr**S	///	0 0 0 0 0 0 0 0 0 0	/

$CR_{4\times BF+32:4\times BF+35} \leftarrow CR_{4\times \mathbf{cr}S+32:4\times \mathbf{cr}S+35}$

The contents of field **cr**S (bits 4×**cr**S+32–4×**cr**S+35) of CR are copied to field **crfD** (bits 4×**crfD**+32–4×**crfD**+35) of CR.

Other registers altered: CR

mcrfs

Floating-point	User

mcrfs

Move to Condition Register from FPSCR

mcrfs **crfD,cr**S

0 5	6 8	9 10	11 13	14 20	21 30	31
1 1 1 1 1 1	**crfD**	//	**cr**S	///	0 0 0 1 0 0 0 0 0 0	/

```
$CR_{BF\times 4:\mathbf{crfD}\times 4+3} \leftarrow FPSCR_{\mathbf{cr}S\times 4:\mathbf{cr}S\times 4+3}$
$FPSCR_{\mathbf{cr}S\times 4:\mathbf{cr}S\times 4+3} \leftarrow$ 0b0000
```

The contents of FPSCR[**cr**S] are copied to CR field **crfD**. All exception bits copied are cleared in the FPSCR. If the FX bit is copied, it is cleared in the FPSCR.

If MSR[FP]=0, an attempt to execute **mcrfs** causes a floating-point unavailable interrupt.

Other registers altered:

- CR field **crfD**
  FX OX(if **cr**S=0)
  UX ZX XX VXSNAN(if **cr**S=1)
  VXISI VXIDI VXZDZ VXIMZ(if **cr**S=2)
  VXVC(if **cr**S=3)
  VXSOFT VXSQRT VXCVI(if **cr**S=5)crfD

mcrxr mcrxr

Base	User

Move to Condition Register from Integer Exception Register

mcrxr **crfD**

0 5	6 8	9 20	21 30	31
0 1 1 1 1 1	**crfD**	///	1 0 0 0 0 0 0 0 0 0	/

```
$CR_{4\times\mathbf{crfD}+32:4\times\mathbf{crfD}+35} \leftarrow XER_{32:35}$
$XER_{32:35} \leftarrow$ 0b0000
```

The contents of XER[32–35] are copied to CR field **crfD**. XER[32–35] are cleared.

Other registers altered: CR XER[32–35]

mfcr

Base	User

Move from Condition Register

mfcr **r**D

0 5	6 10	11	20	21 30	31
0 1 1 1 1 1	**r**D	0	///	0 0 0 0 0 1 0 0 1 1	/

```
rD ← $^{32}0$ || CR
```

The contents of the CR are placed into **r**D[32–63]. Bits **r**D[0–31] are cleared.

Other registers altered: None

mfdcr

E.DC	Hypervisor

mfdcr

Move from Device Control Register

mfdcr **r**D,DCRN

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	$dcrn_{5:9}$	$dcrn_{0:4}$	0 1 0 1 0 0 0 0 1 1	/

```
DCRN ← $dcrn_{0:4}$ || $dcrn_{5:9}$
rD ← DCREG(DCRN)
```

DCRN identifies the DCR (see the user’s manual for a list of DCRs supported by the implementation).

The contents of the designated DCR are placed into **r**D. For 32-bit DCRs, the contents of the DCR are placed into **r**D[32–63]. Bits **r**D[0–31] are cleared.

This instruction is hypervisor privileged.

Other registers altered: None

NOTE: Software Considerations

Other than **mtdcr**, EIS does not define other device control register instructions from PowerISA such as **mfdcrx**, **mfdcrux**, **mtdcrux**.

mffs mffs

Floating-Point	User

Move from FPSCR

mffs	**fr**D	(Rc=0)
mffs.	**fr**D	(Rc=1)

0 5	6 10	11 20	21 30	31
1 1 1 1 1 1	**fr**D	///	1 0 0 1 0 0 0 1 1 1	Rc

```
frD ← FPSCR
```

The contents of the FPSCR are placed into **fr**D[32–63]; **fr**D[0–31] are undefined.

If MSR[FP]=0, an attempt to execute **mffs**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- CR1 ← FX || FEX || VX || OX (if Rc=1)

mfmsr

Base	Supervisor

mfmsr

Move from Machine State Register

mfmsr **r**D

0 5	6 10	11 20	21 30	31
0 1 1 1 1 1	**r**D	///	0 0 0 1 0 1 0 0 1 1	/

```
rD ← $^{32}$0 || MSR
```

The contents of the MSR are placed into **r**D[32–63]. Bits **r**D[0–31] are cleared.

This instruction is guest supervisor privileged.

Other registers altered: None

mfocrf

Base	User

mfocrf

Move From One Condition Register Field

mfocrf **r**D,FXM

0 5	6 10	11	12 19	20	21 30	31
0 1 1 1 1 1	**r**D	1	CRM	/	0 0 0 0 0 1 0 0 1 1	/

```
rD ← undefined
count ← 0
do i = 0 to 7
  if CRM$_i$ = 1 then
    n ← i
    count ← count + 1
if count = 1 then
    rD$_{4×n+32:4×n+35}$ ← CR$_{4×n+32:4×n+35}$
```

If exactly one bit of the CRM field is set, let n be the position of that bit in the field (0 ≤ n ≤ 7). The contents of CR field n (CR bits 4×n+32:4×n+35) are placed into bits 4×n+32:4×n+35 of register **r**D and the contents of the remaining bits of register **r**D are undefined. Otherwise, the contents of register **r**D are undefined.

Other registers altered: None

NOTE: Software Considerations

Architected forms of **mtocrf** and **mfocrf** are intended to replace the old forms of the **mtcrf** and **mfcr**, which will eventually be phased out of the architecture. The new forms are backward compatible with processors that ignore reserved fields in instruction encodings. On those processors, the new forms are treated as the old forms.

However, on processors that strictly decode and do not ignore reserved fields in instructions, the new forms may cause the system illegal instruction error handler to be invoked.

mfpmr

E.PM	User

mfpmr

Move from Performance Monitor Register

mfpmr **r**D,PMRN

0 5	6 10	11 15	16 20	21	31
0 1 1 1 1 1	rD	pmrn$_{5:9}$	pmrn$_{0:4}$	0 1 0 1 0 0 1 1 1 0	/

```
PMRN ← pmrn$_{0:4}$ || pmrn$_{5:9}$
rD ← PMREG(PMRN)
```

PMRN denotes a performance monitor register. Section 3.15, “Performance Monitor Registers (PMRs) <E.PM>,” lists EIS architected performance monitor registers (PMRs).

The contents of the performance monitor register specified by PMRN are placed into **r**D.

Execution of **mfpmr** depends on the PMRN specified, the current privilege level, and the value of MSRP[PMMP] as defined by Table 5-10:

Table 5-10. mfpmr Execution

PMR Implemented	PMRN[5] (privileged PMR)	MSR[PR] (supervisor or user)	MSR[GS] (hypervisor or guest) <E.HV>	MSRP[PMMP] <E.HV>	Result
No	x	x	x	x	Illegal instruction exception
Yes	0	x	x	0	Read PMR[PMRN] register
		x	0	1	Read PMR[PMRN] register
		0	1	1	Embedded hypervisor privilege exception <E.HV>
		1	1	1	Read value of 0
	1	0	0	x	Read PMR[PMRN] register
		1	x	x	Privilege exception
		0	1	0	Read PMR[PMRN] register
		0	1	1	Embedded hypervisor privilege exception <E.HV>

Other registers altered: None

mfspr

Base	User

mfspr

Move from Special Purpose Register

mfspr **r**D,SPRN

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	rD	sprn$_{5:9}$	sprn$_{0:4}$	0 1 0 1 0 1 0 0 1 1	/

```
SPRN ← sprn$_{0:4}$ || sprn$_{5:9}$
if SPR(SPRN) is a 64-bit SPR then
   rD ← SPR(SPRN)
else
   rD ← $^{32}$0 || SPR(SPRN)
```

SPRN denotes an SPR (see Section 3.2.2, “Special-Purpose Registers (SPRs)”).

The contents of the designated SPR are placed into **r**D. For 32-bit SPRs, the contents of the SPR are placed into **r**D[32–63]. Bits **r**D[0–31] are cleared.

Execution of **mfspr** depends on the SPRN specified and the current privilege level as defined by Table 5-11:

Table 5-11. mfspr Execution

SPR Implemented	SPRN[5] (privileged SPR)	Hypervisor Privileged SPR	MSR[PR] (supervisor or user)	MSR[GS] (hypervisor or guest) <E.HV>	Result
No	x	N/A	1	x	Illegal instruction exception
	x	N/A	0	x	Boundedly undefined
Yes	0	N/A	x	x	Read SPR[SPRN] register
	1	x	1	x	Privilege exception
		No	0	x	Read SPR[SPRN] register
		Yes	0	0	Read SPR[SPRN] register
			0	1	Embedded hypervisor privilege exception <E.HV>

Other registers altered: None

Simplified mnemonics: See Section C.9, “Simplified Mnemonics for Accessing SPRs.”

mftb mftb

Base	User

Move from Time Base

mftb **r**D,**TBRN**

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	$tbrn_{5:9}$	$tbrn_{0:4}$	0 1 0 1 1 1 0 0 1 1	0

```
TBRN    ← $tbrn_{0:4}$ || $tbrn_{5:9}$
if Mode32 then
  if TBRN = 268 then
    rD ← $^{32}$undefined || $TB_{32:63}$
  else if TBRN = 269 then
    rD ← $^{32}$undefined || $TB_{0:31}$
if Mode64 then
  if TBRN = 268 then
    rD ← TB
  else if TBRN = 269 then
    rD ← $^{32}$0 || $TB_{0:31}$
```

Reads the time base (TB) using the defined SPRs time base lower (TBL, SPR 268) and time base upper (TBU, SPR 269).

The contents of the designated SPR are placed into **r**D. In 64-bit mode, reading TBL copies the entire contents of the time base (TBU || TBL) into **r**D. In 32-bit mode, the upper 32 bits of the destination register are undefined.

Values other than 268 or 269 for TBRN produce boundedly undefined results.

Other registers altered: None.

Simplified mnemonics:

- **mftb r**D equivalent to: **mftb r**D**,**268
- **mftbu r**D equivalent to: **mftb r**D**,**269

NOTE: Software Considerations

mftb was part of the original PowerPC architecture, but is being phased out of PowerISA. Software should use **mfspr** to access the time base.

msgclr msgclr

E.PC	Hypervisor

Message Clear

msgclr **r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	///	rB	0 0 1 1 1 0 1 1 1 0	/

```
msgtype ← $(rB)_{32:36}$
clear_received_message(msgtype)
```

msgclr clears a message of *msgtype* previously accepted by the processor executing the **msgclr**. *msgtype* is defined by the contents of **r**B[32–36]. A message is said to be cleared when a pending exception generated by an accepted message has not yet taken its associated interrupt.

If a pending exception exists for *msgtype* that exception is cleared at the completion of the **msgclr** instruction.

For processors, the types of messages that can be cleared are defined in Section 7.12, “Processor Signaling (msgsnd and msgclr) <E.PC>.”

This instruction is hypervisor privileged.

Other registers altered: None

NOTE: Software Considerations

Execution of a **msgclr** instruction that clears a pending exception when the associated interrupt is masked because the interrupt enable (MSR[EE], MSR[CE], or MSR[ME]) is not set to 1 will always clear the pending exception (and thus the interrupt will not occur) if a subsequent instruction causes the associated enabling condition to be set.

NOTE: Software Considerations

There is no synchronization with respect to when messages are received and accepted by the processor and the **msgclr** instruction.

msgsnd

E.PC	Hypervisor

msgsnd

Message Send

msgsnd **r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	///	///	rB	0 0 1 1 0 0 1 1 1 0	/

```
msgtype ← $(rB)_{32:36}$
payload ← $(rB)_{37:63}$
send_msg_to_domain(msgtype, payload)
```

msgsnd sends a message to all devices in the coherence domain. The message contains a type and a payload. The message type (*msgtype*) is defined by the contents of **r**B[32–36] and the message payload is defined by the contents of **r**B[37–63]. Message delivery is reliable and guaranteed. Each device may perform specific actions based on the message type and payload or may ignore messages. Consult the implementation user’s manual for specific actions taken based on message type and payload.

For processors, actions taken on receipt of a message are defined in Section 7.12, “Processor Signaling (msgsnd and msgclr) <E.PC>.”

For storage access ordering, **msgsnd** is ordered with stores for memory barriers established by **sync** 0.

This instruction is hypervisor privileged.

Other registers altered: None

msync

Base	User

msync

See **sync**. **msync** is now an extended mnemonic for **sync** 0.

NOTE: Software Considerations

With the advent of PowerISA 2.03, the **msync** instruction reverted to its mnemonic from the original PowerPC architecture. **msync** as defined by Book E, used the same encodings as **sync** 0 from the PowerPC server architecture. Thus the mnemonic was returned to its original definition as **sync**. Assemblers should treat **msync** as an extended mnemonic for **sync** 0.

mtcrf

Base	User

Move to Condition Register Fields

mtcrf CRM**,r**S

0 5	6 10	11	12 19	20	21 30	31
0 1 1 1 1 1	**r**S	0	CRM	/	0 0 1 0 0 1 0 0 0 0	/

```
i ← 0
do while i < 8
   if $CRM_i$=1 then $CR_{4\times i+32:4\times i+35}$ ← $(rS)_{4\times i+32:4\times i+35}$
   i ← i+1
```

The contents of **r**S[32–63] are placed into the CR under control of the field mask specified by CRM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0–7. If CRM_i = 1, CR field i (CR bits 4×i+32 through 4×i+35) is set to the contents of the corresponding field of **r**S[32–63].

Other registers altered: CR fields selected by mask

NOTE: Software Considerations

See the description of **mtocrf**.

Simplified mnemonics: See Section C.10.6, “Move to Condition Register (mtcr).”

mtdcr

E.DC	Supervisor

mtdcr

Move to Device Control Register

mtdcr DCRN,**r**S

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	**r**S	$\text{dcrn}_{5:9}$	$\text{dcrn}_{0:4}$	0 1 1 1 0 0 0 0 1 1	/

```
DCRN ← $\text{dcrn}_{0:4}$ || $\text{dcrn}_{5:9}$
DCREG(DCRN) ← (rS)
```

DCRN identifies the DCR (see user’s manual for a list of DCRs supported by the implementation).

The contents of **r**S are placed into the designated DCR. For 32-bit DCRs, **r**S[32–63] are placed into the DCR.

This instruction is hypervisor privileged.

Other registers altered: None

NOTE: Software Considerations

Other than **mfdcr**, EIS does not define other device control register instructions from PowerISA such as **mfdcrx**, **mfdcrux**, **mtdcrux**.

mtfsb0 mtfsb0

Floating-Point	User

Move to FPSCR Bit 0

mtfsb0	**crb**D	(Rc=0)
mtfsb0.	**crb**D	(Rc=1)

0 5	6 10	11 20	21 30	31
1 1 1 1 1 1	**crb**D	///	0 0 0 1 0 0 0 1 1 0	Rc

```
FPSCR[crbD+32]← 0b0
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

FPSCR[**crb**D+32] is cleared.

If MSR[FP]=0, an attempt to execute **mtfsb0**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPSCR bit **crb**D+32
  CR1 (if Rc=1)

NOTE: Software Considerations

Bits 33 and 34 of FPSCR (FEX and VX) cannot be explicitly reset.

mtfsb1

Floating-Point	User

mtfsb1

Move to FPSCR Bit 1

mtfsb1	**crb**D	(Rc=0)
mtfsb1.	**crb**D	(Rc=1)

0 5	6 10	11 20	21 30	31
1 1 1 1 1 1	**crb**D	///	0 0 0 0 1 0 0 1 1 0	Rc

```
FPSCR[crbD+32]← 0b1
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

FPSCR[**crb**D+32] is set.

If MSR[FP]=0, an attempt to execute **mtfsb1**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPSCR bit **crb**D+32 and FX
  CR1 (if Rc=1)

NOTE: Software Considerations

Bits 33 and 34 of FPSCR (FEX and VX) cannot be explicitly set.

mtfsf

Floating-Point	User

mtfsf

Move to FPSCR Fields

mtfsf	FM,**fr**B	(Rc=0)
mtfsf.	FM,**fr**B	(Rc=1)

0 5	6	7 14	15	16 20	21 30	31
1 1 1 1 1 1	L	FM	W	**fr**B	1 0 1 1 0 0 0 1 1 1	Rc

```
i ← 0
do while i<8
   b ← (i+8) × 4
   if $FM_i$=1 then $FPSCR_{b:b+3}$ ← $frB_{b:b+3}$
   i ← i+1
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The contents of **fr**B[32–63] are placed into the FPSCR under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0–7. If FM_i=1, FPSCR field i+8 (FPSCR bits 4×(i+8) through 4×(i+8)+3) is set to the contents of the corresponding field of the low-order 32 bits of **fr**B.

FPSCR[FX] is altered only if FM_0 = 1.

The L and W fields of the instruction should always be set to 0.

If MSR[FP]=0, an attempt to execute **mtfsf**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPSCR fields selected by mask
  CR1 (if Rc=1)

NOTE: Software Considerations

Updating fewer than all eight fields of the FPSCR may have substantially poorer performance on some implementations than updating all the fields.

NOTE: Software Considerations

When FPSCR[32–35] is specified, bits 32 (FX) and 35 (OX) are set to the values of $(\mathbf{fr}B)_{32}$ and $(\mathbf{fr}B)_{35}$ (that is, even if this instruction causes OX to change from 0 to 1, FX is set from $(\mathbf{fr}B)_{32}$ and not by the usual rule that FX is set when an exception bit changes from 0 to 1). Bits 33 and 34 (FEX and VX) are set according to the usual rule (see Table 3-5) and not from $(\mathbf{fr}B)_{33–34}$.

NOTE: Software Considerations

Some processors may take an illegal instruction exception if the W or L fields contain non-zero values.

mtfsfi mtfsfi

Floating-Point	User

Move to FPSCR Field Immediate

mtfsfi	**crfD,**UIMM	(Rc=0)
mtfsfi.	**crfD,**UIMM	(Rc=1)

0 5	6 8	9 10	11 14	15	16 19	20	21 30	31
1 1 1 1 1 1	**crfD**	///	///	W	UIMM	/	0 0 1 0 0 0 0 1 1 0	Rc

```
b ← (crfD+8) × 4
FPSCR$_{b:b+3}$ ← UIMM
if Rc=1 then do
   CR1 ← FX || FEX || VX || OX
```

The value of the UIMM field is placed into FPSCR field **crfD**+8.

FPSCR[FX] is altered only if **crfD** = 0.

The W field of the instruction should always be set to 0.

If MSR[FP]=0, an attempt to execute **mtfsfi**[**.**] causes a floating-point unavailable interrupt.

Other registers altered:

- FPSCR field **crfD**+8
  CR1 (if Rc=1)

NOTE: Software Considerations

When FPSCR[32–35] is specified, bits 32 (FX) and 35 (OX) are set to the values of UIMM$_0$ and UIMM$_3$ (that is, even if this instruction causes OX to change from 0 to 1, FX is set from UIMM$_0$ and not by the usual rule that FX is set when an exception bit changes from 0 to 1). Bits 33 and 34 (FEX and VX) are set according to the usual rule (see Table 3-5), and not from UIMM$_{1:2}$.

NOTE: Software Considerations

Some processors may take an illegal instruction exception if the W field contains a non-zero value.

mtmsr

Embedded	Supervisor

Move to Machine State Register

mtmsr **r**S

0 5	6 10	11 20	21 30	31
0 1 1 1 1 1	**r**S	///	0 0 1 0 0 1 0 0 1 0	/

```
newmsr ← $(rS)_{32:63}$
if MSR[CM] ≠ newmsr[CM] then $NIA_{0:31}$ ← 0
if MSR[GS] = 1 then
    newmsr[GS] ← MSR[GS]
    newmsr[WE] ← MSR[WE]
    prots ← 0
    prots[UCLEP DEP PMMP] ← MSRP[UCLEP DEP PMMP]
    newmsr ← prots & MSR | ~prots & newmsr
MSR ← newmsr
```

The contents of **r**S[32–63] are placed into the MSR. If the processor is changing between 32-bit mode and 64-bit mode, the high-order 32 bits of the fetch address are set to zero. <64-bit>

<E.HV>:
If the processor is executing in guest supervisor state when **mtmsr** is executed, GS, ME, and other MSR bits protected by MSRP are not modified.

This instruction is guest supervisor privileged.

This instruction is execution synchronizing, but is not context synchronizing and requires software to perform a context synchronizing instruction before the changes to the MSR are guaranteed to be performed. See Section 4.5.4.5, “Execution Synchronization.” and Section 4.5.4.4, “Context Synchronization.”

In addition, changes to the EE or CE bits are effective as soon as the instruction completes. Thus if MSR[EE]=0 and an external interrupt is pending, executing an **mtmsr** that sets MSR[EE] causes the external interrupt to be taken before the next instruction is executed, if no higher priority exception exists. Likewise, if MSR[CE]=0 and a critical input interrupt is pending, executing an **mtmsr** that sets MSR[CE] causes the critical input interrupt to be taken before the next instruction is executed if no higher priority exception exists.

Other registers altered: MSR

NOTE: Software Considerations

- Changing MSR bits associated with the current context must be done with extreme care using **mtmsr**. The MSR context bits are: CM<64>, IS, DS, and GS<E.HV>.
- For example, changing MSR[IS] from 0 to 1 can have the side effect of changing where instructions are fetched from (as does MSR[GS] and MSR[CM]<64>). The effects of changing bits in the MSR associated with instruction or data context are not guaranteed to occur until a

context synchronizing operation has been performed. Until the context synchronizing operation is performed, instructions may be fetched and executed using any permutation of the context bits from the old and new MSR. To avoid an implicit branch, software must ensure that the permutation of context bits from the old and new MSR has translations that all map to the same real address. Such permutations can change from cycle to cycle until the context synchronizing operation has been performed.

- For this reason, it is recommend to always use a return from interrupt instruction when changing MSR context bits since the return from interrupt instruction is context synchronizing.

mtocrf mtocrf

Base	User

Move To One Condition Register Field

mtocrf CRM,**r**S

0 … 5	6 … 10	11	12 … 19	20	21 … 30	31
0 1 1 1 1 1	**r**S	1	CRM	/	0 0 1 0 0 1 0 0 0 0	/

```
count ← 0
do i = 0 to 7
   if $CRM_i$ = 1 then
       n ← (i × 4) + 32
       count ← count + 1
if count = 1 then
   $CR_{n:n+3}$ ← $(rS)_{n:n+3}$
else
   CR ← undefined
```

If exactly one CRM field bit is set, let *i* be the position of that bit in the field (0 ≤ *i* ≤ 7) and *n* be the corresponding starting bit number in the CR. The contents of bits *n*:*n*+3 of register **r**S are placed into CR field *i* (CR bits *n*:*n*+3). Otherwise, the contents of the CR are undefined.

Special registers altered: CR field selected by CRM

NOTE: Software Considerations

Architected forms of **mtocrf** and **mfocrf** are intended to replace the old forms of the **mtcrf** and **mfcr** which will eventually be phased out of the architecture. The new forms are backward compatible with processors that ignore reserved fields in instructions. On those processors, the new forms are treated as the old forms.

However, on older processors that do not ignore reserved fields in instructions the new forms may be treated as follows:

mtocrf may cause the system illegal instruction error handler to be invoked.

mfocrf may place an undefined value into register **r**D.

mtpmr

E.PM	User

mtpmr

Move To Performance Monitor Register

mtpmr PMRN,**r**S

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	rS	$pmrn_{5:9}$	$pmrn_{0:4}$	0 1 1 1 0 0 1 1 1 0	/

```
PMRN ← $pmrn_{0:4}$ || $pmrn_{5:9}$
PMREG(PMRN) ← (rS)
```

PMRN denotes a performance monitor register. Section 3.15, “Performance Monitor Registers (PMRs) <E.PM>,” lists supported performance monitor registers).

The contents of **r**S are placed into the designated performance monitor register.

Execution of **mtpmr** depends on the PMRN specified, the current privilege level, and the value of MSRP[PMMP] as defined by Table 5-12:

Table 5-12. mtpmr Execution

PMR Implemented	PMRN[5] (privileged PMR)	MSR[PR] (supervisor or user)	MSR[GS] (hypervisor or guest) <E.HV>	MSRP[PMMP] <E.HV>	Result
No	x	x	x	x	Illegal instruction exception
Yes	0	x	x	0	Write PMR[PMRN] register
		x	0	1	Write PMR[PMRN] register
		0	1	1	Embedded hypervisor privilege exception <E.HV>
		1	1	1	Write PMR[PMRN] register
	1	0	0	x	Write PMR[PMRN] register
		1	x	x	Privilege exception
		0	1	0	Write PMR[PMRN] register
		0	1	1	Embedded hypervisor privilege exception <E.HV>

Other registers altered: None

NOTE: Software Considerations

A PMRN that is “read-only” is considered to be unimplemented for writing (**mtpmr** will treat PMRN as unimplemented).

NOTE: Architecture

User writable PMRs should not be defined since the architecture does not allow them to be virtualized.

mtspr

Base	User

mtspr

Move to Special Purpose Register

mtspr SPRN,**r**S

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	rS	sprn$_{5:9}$	sprn$_{0:4}$]	0 1 1 1 0 1 0 0 1 1	/

```
SPRN ← sprn$_{0:4}$ || sprn$_{5:9}$
if SPR(SPRN) is a 64-bit SPR then
    SPR(SPRN) ← (rS)
else
    SPR(SPRN) ← (rS)$_{32:63}$
```

SPRN denotes an SPR (see Section 3.2.2, “Special-Purpose Registers (SPRs),” and the user’s manual of the implementation for a list of all SPRs that are implemented).

The contents of **r**S are placed into the designated SPR. For 32-bit SPRs, the contents of **r**S[32–63] are placed into the SPR.

Execution of **mtspr** depends on the SPRN specified and the current privilege level as defined by Table 5-13:

Table 5-13. mtspr Execution

SPR Implemented	SPRN[5] (privileged SPR)	Hypervisor Privileged SPR	MSR[PR] (supervisor or user)	MSR[GS] (hypervisor or guest) <E.HV>	Result
No	x	N/A	1	x	Illegal instruction exception
	x	N/A	0	x	Boundedly undefined
Yes	0	N/A	x	x	Write SPR(SPRN) register
	1	x	1	x	Privilege exception
		No	0	x	Write SPR(SPRN) register
		Yes	0	0	Write SPR(SPRN) register
			0	1	Embedded hypervisor privilege exception <E.HV>

Other registers altered:

- SPR(SPRN)

NOTE: Software Considerations

Altering certain SPRs requires synchronization. See Section 4.5.4.3, “Synchronization Requirements.”

NOTE: Software Considerations

A SPRN that is “read-only” is considered to be unimplemented for writing (**mtspr** will treat SPRN as unimplemented).

Simplified mnemonics: See Section C.9, “Simplified Mnemonics for Accessing SPRs.”

mulhd

64	User

mulhd

Multiply High Doubleword

mulhd	**r**D,**r**A,**r**B	(Rc=0)
mulhd.	**r**D,**r**A,**r**B	(Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	/	0 0 1 0 0 1 0 0 1	Rc

```
prod$_{0:127}$ ← (rA) × (rB)
if Rc=1 then do
   LT  ← prod$_{m:63}$ < 0
   GT  ← prod$_{m:63}$ > 0
   EQ  ← prod$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rD ← prod$_{0:63}$
```

Bits 0–63 of the 128-bit product of the contents of **r**A and the contents of **r**B are placed into **r**D.

Both operands and the product are interpreted as signed integers.

Other registers altered: CR0 (if Rc=1)

mulhdu

64	User

mulhdu

Multiply High Doubleword Unsigned

mulhdu	**r**D,**r**A,**r**B	(Rc=0)
mulhdu.	**r**D,**r**A,**r**B	(Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	/	0 0 0 0 0 1 0 0 1	Rc

```
prod$_{0:127}$ ← (rA) × (rB)
if Rc=1 then do
   LT  ← prod$_{m:63}$ < 0
   GT  ← prod$_{m:63}$ > 0
   EQ  ← prod$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rD ← prod$_{0:63}$
```

Bits 0–63 of the 128-bit product of the contents of **r**A and the contents of **r**B are placed into **r**D.

Both operands and the product are interpreted as unsigned integers except that if Rc=1 the first three bits of CR0 are set by signed comparison of the result to zero.

Other registers altered: CR0 (if Rc=1)

mulhw mulhw

Base	User

Multiply High Word

mulhw **r**D,**r**A,**r**B (Rc=0)
mulhw. **r**D,**r**A,**r**B (Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	/	0 0 1 0 0 1 0 1 1	Rc

```
$prod_{0:63}$  ← $(rA)_{32:63}$ × $(rB)_{32:63}$
if Rc=1 then do
   if Mode32 then do
       LT  ← $prod_{0:31}$ < 0
       GT  ← $prod_{0:31}$ > 0
       EQ  ← $prod_{0:31}$ = 0
       CR0 ← LT || GT || EQ || SO
   else
       CR0 ← $^{3}$undefined || SO
$rD_{32:63}$ ← $prod_{0:31}$
$rD_{0:31}$  ← undefined
```

Bits 0–31 of the 64-bit product of the contents of **r**A[32–63] and the contents of **r**B[32–63] are placed into **r**D[32–63]. Bits **r**D[0–31] are undefined.

Both operands and the product are interpreted as signed integers.

Other registers altered: CR0 (bits 0:2 are undefined in 64-bit mode) (if Rc=1)

mulhwu

Base	User

Multiply High Word Unsigned

mulhwu	**r**D,**r**A,**r**B	(Rc=0)
mulhwu.	**r**D,**r**A,**r**B	(Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	/	0 0 0 0 0 1 0 1 1	Rc

```
$prod_{0:63} \leftarrow (rA)_{32:63} \times (rB)_{32:63}$
   if Mode32 then do
       LT  $\leftarrow prod_{0:31} < 0$
       GT  $\leftarrow prod_{0:31} > 0$
       EQ  $\leftarrow prod_{0:31} = 0$
       CR0 $\leftarrow$ LT || GT || EQ || SO
   else
       CR0 $\leftarrow {}^{3}$undefined || SO
$rD_{32:63} \leftarrow prod_{0:31}$
$rD_{0:31} \leftarrow$ undefined
```

Bits 0–31 of the 64-bit product the contents of **r**A[32–63] and the contents of **r**B[32–63] are placed into **r**D[32–63]. Bits **r**D[0–31] are undefined.

Both operands and the product are interpreted as unsigned integers, except that if Rc=1 the first three bits of CR field 0 are set by signed comparison of the result to zero.

Other registers altered: CR0 (bits 0:2 are undefined in 64-bit mode) (if Rc=1)

mulld

64	User

mulld

Multiply Low Doubleword

mulld	**r**D,**r**A,**r**B	(OE=0, Rc=0)
mulld.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
mulldo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
mulldo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)

0 – 5	6 – 10	11 – 15	16 – 20	21	22 – 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	OE	0 1 1 1 0 1 0 0 1	Rc

```
$prod_{0:127} \leftarrow (rA) \times (rB)$
if OE=1 then do
   $OV \leftarrow (prod_{0:63} \neq {}^{32}0)$ & $(prod_{0:63} \neq {}^{32}1)$
   $SO \leftarrow SO \mid OV$
if Rc=1 then do
   $LT \leftarrow prod_{64:127} < 0$
   $GT \leftarrow prod_{64:127} > 0$
   $EQ \leftarrow prod_{64:127} = 0$
   $CR0 \leftarrow LT \parallel GT \parallel EQ \parallel SO$
$rD \leftarrow prod_{64:127}$
```

Bits 64–127 of the 128-bit product of the contents of **r**A and the contents of **r**B are placed into **r**D.

If OE=1 then OV is set if the product cannot be represented in 64 bits.

Both operands and the product are interpreted as signed integers.

Other registers altered: CR0 (if Rc=1)

mulli

Base	User

mulli

Multiply Low Immediate

mulli **r**D,**r**A,SIMM

0–5	6–10	11–15	16–31
0 0 0 1 1 1	**r**D	**r**A	SIMM

```
$prod_{0:127}$ ← (rA) × EXTS(SIMM)
rD ← $prod_{64:127}$
```

Bits 64–127 of the 128-bit product of the contents of **r**A and the sign-extended value of the SIMM field are placed into **r**D.

Both operands and the product are interpreted as signed integers.

Other registers altered: None

NOTE: Software Considerations

For **mulli** and **mullw**, bits 32–63 of the product are the correct 32-bit product for 32-bit mode.

For **mulli** and **mulld**, the low-order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers.

For **mulli** and **mullw**, bits 32–63 of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

mullw | Base | User | mullw

Multiply Low Word

mullw	**r**D,**r**A,**r**B	(OE=0, Rc=0)
mullw.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
mullwo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
mullwo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	OE	0 1 1 1 0 1 0 1 1	Rc

```
$prod_{0:63} \leftarrow (rA)_{32:63} \times (rB)_{32:63}$
if OE=1 then do
   $OV \leftarrow (prod_{0:31} \neq {}^{32}0)$ & $(prod_{0:31} \neq {}^{32}1)$
   $SO \leftarrow SO \mid OV$
if Rc=1 then do
       $LT \leftarrow prod_{m:63} < 0$
       $GT \leftarrow prod_{m:63} > 0$
       $EQ \leftarrow prod_{m:63} = 0$
       $CR0 \leftarrow LT \parallel GT \parallel EQ \parallel SO$
$rD \leftarrow prod$
```

The 64-bit product of the contents of **r**A[32–63] and the contents of **r**B[32–63] is placed into **r**D.

If OE=1, OV is set if the product cannot be represented in 32 bits.

Both operands and the product are interpreted as signed integers.

Other registers altered:

- CR0 (if Rc=1)
  SO   OV (if OE=1)

NOTE: Software Considerations

For **mulli** and **mullw**, bits 32–63 of the product are the correct 32-bit product for 32-bit mode.

For **mulli** and **mulld**, the low-order 64 bits of the product are independent of whether the operands are regarded as signed or unsigned 64-bit integers.

For **mulli** and **mullw**, bits 32–63 of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

nand

Base	User

nand

NAND

nand	**r**A,**r**S,**r**B	(Rc=0)
nand.	**r**A,**r**S,**r**B	(Rc=1)

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 1 1 1 0 1 1 1 0 0	Rc

```
result ← ¬((rS) & (rB))
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are ANDed with the contents of **r**B and the one’s complement of the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

neg

Base	User

neg

Negate

neg **r**D,**r**A (OE=0, Rc=0)
neg. **r**D,**r**A (OE=0, Rc=1)
nego **r**D,**r**A (OE=1, Rc=0)
nego. **r**D,**r**A (OE=1, Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	///	OE	0 0 1 1 0 1 0 0 0	Rc

```
result ← ¬(rA) + 1
if OE=1 then do
    if Mode32 then do
        if $(rA)_{32:63}$ = 0x8000_0000 then do
            OV ← 1
            SO ← SO | OV
    if Mode64 then do
        if $(rA)_{0:63}$ = 0x8000_0000_0000_0000 then do
            OV ← 1
            SO ← SO | OV
if Rc=1 then do
    LT  ← $result_{m:63}$ < 0
    GT  ← $result_{m:63}$ > 0
    EQ  ← $result_{m:63}$ = 0
    CR0 ← LT || GT || EQ || SO
rD ← result
```

The sum of the one’s complement of the contents of **r**A and 1 is placed into **r**D.

If the processor is in 64-bit mode and **r**A contains the most negative 64-bit number (0x8000_0000_0000_0000), the result is the most negative number. Similarly, if the processor is in 32-bit mode and **r**A[32–63] contain the most negative 32-bit number (0x8000_0000), bits 32–63 of the result contain the most negative 32-bit number and, if OE=1, OV is set.

Other registers altered:

- CR0 (if Rc=1)
  SO OV (if OE=1)

nor nor

Base	User

NOR

nor	**r**A,**r**S,**r**B	(Rc=0)
nor.	**r**A,**r**S,**r**B	(Rc=1)

0	5	6	10	11	15	16	20	21	30	31
0 1 1 1 1 1		**r**S		**r**A		**r**B		0 0 0 1 1 1 1 1 0 0		Rc

```
result ← ¬((rS) | (rB))
if Rc=1 then do
        LT  ← result$_{m:63}$ < 0
        GT  ← result$_{m:63}$ > 0
        EQ  ← result$_{m:63}$ = 0
        CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are ORed with the contents of **r**B and the one’s complement of the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

Simplified mnemonics: See Section C.10.5, “Complement Register (not).”

or

Base	User

or

OR

or	**r**A,**r**S,**r**B	(Rc=0)
or.	**r**A,**r**S,**r**B	(Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	rS	rA	rB	0 1 1 0 1 1 1 1 0 0	Rc

```
result ← (rS) | (rB)
if Rc=1 then do
   LT  ← result$_{m:63}$ < 0
   GT  ← result$_{m:63}$ > 0
   EQ  ← result$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are ORed with the contents of **r**B and the result is placed into **r**A.

Forms of the or instruction where **r**A = **r**B = **r**S (**or** rx,rx,rx) are used to provide hints to the processor associated with performance as defined in Table 5-14.

EIS provides enhanced instruction descriptions of **yield**, **mdoio**, and **mdoom** referenced by their extended mnemonic.

Table 5-14. or rx,rx,rx Performance Hints

Instruction	Hint	Extended Mnemonic
or r27,**r**27,**r**27	This form of **or** provides a hint that performance will probably be improved if shared resources dedicated to the executing thread are released for use by other threads.	**yield**
or r29,**r**29,**r**29	This form of or provides a hint that performance will probably be improved if shared resources dedicated to the executing thread are released until all outstanding storage accesses to caching-inhibited storage have been completed.	**mdoio**
or r30,**r**30,**r**30	This form of or provides a hint that performance will probably be improved if shared resources dedicated to the executing thread are released until all outstanding storage accesses to cacheable storage for which the data is not in the cache have been completed.	**mdoom**

Other registers altered: CR0 (if Rc=1)

NOTE: Software Considerations

> **Warning:** Other forms of **or r**x,**r**x,**r**x that are not described in this section may also have effects on performance. Use of these forms should be avoided. If a no-op is needed, the preferred no-op (**ori** 0,0,0) should be used.

Simplified mnemonics: See Section C.10.4, “Move Register (mr).”

orc

Base	User

orc

OR with Complement

orc	**r**A,**r**S,**r**B	(Rc=0)
orc.	**r**A,**r**S,**r**B	(Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 1 1 0 0 1 1 1 0 0	Rc

```
result ← (rS) | ¬(rB)
if Rc=1 then do
   LT  ← $\text{result}_{m:63}$ < 0
   GT  ← $\text{result}_{m:63}$ > 0
   EQ  ← $\text{result}_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are ORed with the complement of the contents of **r**B and the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

ori ori

Base	User

OR Immediate

ori **r**A,**r**S,UIMM (Rc=0)

0					5	6 10	11 15	16 31
0	1	1	0	0	0	**r**S	**r**A	UIMM

```
result ← (rS) | ($^{32}$0 || UIMM)
rA ← result
```

The contents of **r**S are ORed with ^{48}0 || UIMM and the result is placed into **r**A.

The preferred no-op is **ori 0,0,0**

Other registers altered: None

NOTE: Software Considerations

Software should avoid using other no-op forms of **ori** (**ori** rx,rx,0) since such forms may be used in a future version of the architecture to provide hints.

Simplified mnemonics: See Section C.10.1, “No-Op (nop).”

oris oris

Base	User

OR Immediate Shifted

oris **r**A,**r**S,UIMM (Rc=0)

0 5	6 10	11 15	16 31
0 1 1 0 0 1	**r**S	**r**A	UIMM

```
result ← (rS) | ($^{32}0$ || UIMM || $^{16}0$)
rA ← result
```

The contents of **r**S are ORed with $^{32}0$ || UIMM || $^{16}0$and the result is placed into **r**A.

Other registers altered: None

NOTE: Software Considerations

Software should avoid using no-op forms of **oris** (**oris** rx,rx,0) since such forms may be used in a future version of the architecture to provide hints.

popcntb

Base	User

popcntb

Population Count Bytes

popcntb **r**A,**r**S

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	///	0 0 0 1 1 1 1 0 1 0	/

```
do i = 0 to 7
   n ← 0
   do j = 0 to 7
       if $(rS)_{(i\times8)+j}$ = 1 then
           n ← n + 1
   $rA_{(i\times8):(i\times8)+7}$ ← n
```

A count of the number of one bits in each byte of **r**S is placed into the corresponding byte of **r**A. This number ranges from 0 to 8, inclusive.

Other registers altered: None

popcntd

64	User

popcntd

Population Count Doubleword

popcntd **r**A,**r**S

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	///	0 1 1 1 1 1 1 0 1 0	/

```
n ← 0
do j = 0 to 63
   if $(rS)_j$ = 1 then
       n ← n + 1
rA ← n
```

A count of the number of one bits in **r**S is placed into **r**A. This number ranges from 0 to 64, inclusive.

Other registers altered: None

popcntw

Base	User

popcntw

Population Count Words

popcntw **r**A,**r**S

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	///	0 1 0 1 1 1 1 0 1 0	/

```
do i = 0 to 1
   n ← 0
   do j = 0 to 31
       if $(rS)_{(i\times32)+j}$ = 1 then
           n ← n + 1
   $rA_{(i\times32):(i\times32)+31}$ ← n
```

A count of the number of one bits in each word of **r**S is placed into the corresponding word of **r**A. This number ranges from 0 to 32, inclusive.

Other registers altered: None

prtyd

64	User

Parity Doubleword

prtyd **r**A,**r**S

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	///	0 0 1 0 1 1 1 0 1 0	/

```
s ← 0
do i = 0 to 7
    s ← s ⊕ $(rS)_{i\times8+7}$
rA ← $^{63}0 \parallel s$
```

The least significant bit in each byte of **r**S is examined. If there is an odd number of one bits the value 1 is placed into **r**A; otherwise the value of 0 is placed into **r**A.

Other registers altered: None

prtyw

Base	User

prtyw

Parity Word

prtyw **r**A,**r**S

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	///	0 0 1 0 0 1 1 0 1 0	/

```
s ← 0
t ← 0
do i = 0 to 3
    s ← s ⊕ $(rS)_{i\times8+7}$
do i = 4 to 7
    t ← t ⊕ $(rS)_{i\times8+7}$

$rA_{0:31}$ ← $^{31}0$ || s
$rA_{32:63}$ ← $^{31}0$ || t
```

The least significant bit in each byte of **r**S[0–31] is examined. If there is an odd number of one bits the value 1 is placed into **r**A[0–31] ; otherwise the value of 0 is placed into **r**A[0–31]. The least significant bit in each byte of **r**S[32–63] is examined. If there is an odd number of one bits the value 1 is placed into **r**A[32–63] ; otherwise the value of 0 is placed into **r**A[32–63].

Other registers altered: None

rfci

Embedded	Hypervisor

rfci

Return from Critical Interrupt

rfci

0 5	6 20	21 30	31
0 1 0 0 1 1	///	0 0 0 0 1 1 0 0 1 1	/

```
if CSRR1[CM] = 0 then z ← 32
                 else z ← 0
NIA ← $^{z}0$ || CSRR0$_{z:61}$ || 0b00
MSR ← CSRR1
```

The **rfci** instruction is used to return from a critical class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of CSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address CSRR0[0–61]||0b00 (Note: VLE uses CSRR0[0–62]||0b0; see the VLE PEM.). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case, the value placed into the appropriate save/restore register 0 by the interrupt processing mechanism (see Section 7.7, “Interrupt Processing”) is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address in CSRR0 at the time of the execution of the **rfci**).

This instruction is hypervisor privileged.

Execution of this instruction is context synchronizing. See Section 4.5.4.4, “Context Synchronization.”

Other registers altered: MSR

NOTE: Virtualization

Hypervisor’s are required to emulate **rfci** for guests, and they should assure that MSR values established by the guest do not set GS, WE, or any MSR bits protected by MSRP.

rfdi

E.ED	Hypervisor

rfdi

Return From Debug Interrupt

rfdi

0 5	6 10 11 15 16 20	21 30	31
0 1 0 0 1 1	///	0 0 0 0 1 0 0 1 1 1	/

```
if DSRR1[CM] = 0 then z ← 32
                 else z ← 0
NIA ← $^{z}0$ || $DSRR0_{z:61}$ || 0b00
MSR ← DSRR1
```

The **rfdi** instruction is used to return from a debug interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of DSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address DSRR0[0–61]||0b00 (Note: VLE uses DSRR0[0–62]||0b0; see the VLE PEM.). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the appropriate save/restore register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address in DSRR0 at the time of the execution of the **rfdi**).

This instruction is hypervisor privileged.

Execution of this instruction is context synchronizing. See Section 4.5.4.4, “Context Synchronization.”

Other registers altered: MSR

NOTE: Virtualization

Hypervisor’s are required to emulate **rfdi** for guests, and they should assure that MSR values established by the guest do not set GS, WE, or any MSR bits protected by MSRP.

rfgi

E.HV	Supervisor

rfi

Return from Guest Interrupt

rfi

0					5	6 20	21									30	31
0	1	0	0	1	1	///	0	0	0	1	1	0	0	1	1	0	/

```
if GSRR1[CM] = 0 then z ← 32
                 else z ← 0
newmsr ← GSRR1
if MSR[GS] = 1 then do
   newmsr[GS WE] ← MSR[GS WE]
   prots ← MSRP{UCLEP DEP PMMP]
   newmsr ← prots & MSR | ~prots & newmsr
NIA ← $^{z}0$ || GSRR0$_{z:61}$ || 0b00
MSR ← newmsr
```

The **rfgi** instruction is used to return from a base class interrupt in the guest state, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of GSRR1 are placed into the MSR. If rfgi is executed in the guest supervisor state, the bits MSR[GS] and MSR[WE] are not modified and the bits MSR[UCLE DE PMM] are modified only if the associated bits in the MSRP are set to 0. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address GSRR0[0–61]||0b00 (Note: VLE uses GSRR0[0–62]||0b0; see the VLE PEM.). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the appropriate save/restore register 0 by the interrupt processing mechanism (see Section 7.7, “Interrupt Processing”) is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address in GSRR0 at the time of the execution of the **rfgi**).

This instruction is guest supervisor privileged.

When **rfi** is executed in guest supervisor state, the instruction is mapped to **rfgi** and **rfgi** is executed instead.

Execution of this instruction is context synchronizing. See Section 4.5.4.4, “Context Synchronization.”

Other registers altered: MSR

rfi

Embedded	Supervisor

Return from Interrupt

rfi

0 5	6 20	21 30	31
0 1 0 0 1 1	///	0 0 0 0 1 1 0 0 1 0	/

```
if SRR1[CM] = 0 then z ← 32
                else z ← 0
NIA ← $^{z}0$ || SRR0$_{z:61}$ || 0b00
MSR ← SRR1
```

The **rfi** instruction is used to return from a base class interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of SRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address SRR0[0–61]||0b00 (Note: VLE uses SRR0[0–62]||0b0; see the VLE PEM.). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the appropriate save/restore register 0 by the interrupt processing mechanism (see Section 7.7, “Interrupt Processing”) is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address in SRR0 at the time of the execution of the **rfi**).

This instruction is guest supervisor privileged.

<Embedded Hypervisor>:
When **rfi** is executed in guest supervisor state, the instruction is mapped to **rfgi** and **rfgi** is executed instead.

Execution of this instruction is context synchronizing. See Section 4.5.4.4, “Context Synchronization.”

Other registers altered: MSR

rfmci

Embedded	Hypervisor

rfmci

Return from Machine Check Interrupt

rfmci

0 5	6 20	21 30	31
0 1 0 0 1 1	///	0 0 0 0 1 0 0 1 1 0	0

```
if MCSRR1[CM] = 0 then z ← 32
                  else z ← 0
NIA ← $^{z}0$ || $\text{MCSRR0}_{z:61}$ || 0b00
MSR ← MCSRR1
```

The **rfmci** instruction is used to return from a machine check class interrupt, or as a means of establishing a new context and synchronizing on that new context simultaneously.

The contents of MCSRR1 are placed into the MSR. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address MCSRR0[0–61]||0b00 (Note: VLE uses MCSRR0[0–62]||0b0; see the VLE PEM.). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the appropriate save/restore register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address in MCSRR0 at the time of the execution of the **rfmci**).

This instruction is hypervisor privileged.

Execution of this instruction is context synchronizing. See Section 4.5.4.4, “Context Synchronization.”

Other registers altered: MSR

NOTE: Virtualization

Hypervisors are required to emulate **rfmci** for guests, and they should assure that MSR values established by the guest do not set GS, WE, or any MSR bits protected by MSRP.

rldcl rldcl

64	User

Rotate Left Doubleword then Clear Left

rldcl	**r**A,**r**S,**r**B,MB	(Rc=0)
rldcl.	**r**A,**r**S,**r**B,MB	(Rc=1)

0					5	6 10	11 15	16 20	21 26	27			30	31
0	1	1	1	1	0	**r**S	**r**A	**r**B	mb	1	0	0	0	Rc

```
n ← (rB)$_{58:63}$
r ← ROTL$_{64}$((rS), n)
b ← mb$_5$ || mb$_{0:4}$
k ← MASK(b,63)
result ← r & k
if Rc=1 then do
   LT  ← result$_{m:63}$ < 0
   GT  ← result$_{m:63}$ > 0
   EQ  ← result$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are rotated$_{64}$ left the number of bits specified by **r**B[58–63]. A mask is generated having 1 bits from bit MB through bit 63 and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

NOTE: Software Considerations

rldcl can be used to extract an n-bit field that starts at variable bit position b in register **r**S, right-justified into register rA (clearing the remaining 64-n bits of **r**A), by setting **r**B[58–63]=b+n and MB=64-n. It can be used to rotate the contents of a register left (right) by variable n bits, by setting **r**B[58-63]=n (64-n) and MB=0.

NOTE: Software Considerations

MB as specified in the assembly syntax is transformed by the assembler when encoded into the instruction such that mb ← $MB_{1:5}$ || MB_0.

Simplified mnemonics: See Section C.3, “Rotate and Shift Simplified Mnemonics.”

rldcr rldcr

64	User

Rotate Left Doubleword then Clear Right

rldcr	**r**A,**r**S,**r**B,ME	(Rc=0)
rldcr.	**r**A,**r**S,**r**B,ME	(Rc=1)

0 5	6 10	11 15	16 20	21 26	27			30	31
0 1 1 1 1 0	**r**S	**r**A	**r**B	me	1	0	0	1	Rc

```
n ← $(rB)_{58:63}$
r ← $ROTL_{64}((rS), n)$
e ← $me_5 \parallel me_{0:4}$
k ← MASK(0, e)
result ← r & k
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are rotated$_{64}$ left the number of bits specified by **r**B[58–63]. A mask is generated having 1 bits from bit 0 through bit ME and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

NOTE: Software Considerations

rldcl can be used to extract an n-bit field that starts at variable bit position b in register **r**S, left-justified into register rA (clearing the remaining 64-n bits of **r**A), by setting **r**B[58–63]=b and MB=n-1. It can be used to rotate the contents of a register left (right) by variable n bits, by setting **r**B[58-63]=n (64-n) and ME=63.

NOTE: Software Considerations

ME as specified in the assembly syntax is transformed by the assembler when encoded into the instruction such that me ← $ME_{1:5} \parallel ME_0$.

rldic rldic

64	User

Rotate Left Doubleword Immediate then Clear

rldic	**r**A,**r**S,SH,MB	(Rc=0)
rldic.	**r**A,**r**S,SH,MB	(Rc=1)

0 5	6 10	11 15	16 20	21 26	27		29	30	31
0 1 1 1 1 0	**r**S	**r**A	sh	mb	0	1	0	sh	Rc

```
n ← $sh_5$ || $sh_{0:4}$
r ← $ROTL_{64}$((rS), n)
b ← $mb_5$ || $mb_{0:4}$
k ← MASK(b,¬n)
result ← r & k
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are rotated_{64} left SH bits. A mask is generated having 1 bits from bit MB through bit 63-SH and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

NOTE: Software Considerations

rldic can be used to clear the high-order b bits of the contents of a register and then shift the result left by n bits, by setting SH=n and MB=b-n. It can be used to clear the high-order n bits of a register by setting SH=0 and MB=n.

NOTE: Software Considerations

SH and MB as specified in the assembly syntax are transformed by the assembler when encoded into the instruction such that sh ← $SH_{1:5}$ || SH_0 and mb ← $MB_{1:5}$ || MB_0.

Simplified mnemonics: See Section C.3, “Rotate and Shift Simplified Mnemonics.”

rldicl rldicl

64	User

Rotate Left Doubleword Immediate then Clear Left

rldicl	**r**A,**r**S,SH,MB	(Rc=0)
rldicl.	**r**A,**r**S,SH,MB	(Rc=1)

0 5	6 10	11 15	16 20	21 26	27		29	30	31
0 1 1 1 1 0	rS	rA	sh	mb	0	0	0	sh	Rc

```
n ← $sh_5$ || $sh_{0:4}$
r ← $ROTL_{64}$((rS), n)
b ← $mb_5$ || $mb_{0:4}$
k ← MASK(b,63)
result ← r & k
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are rotated$_{64}$ left SH bits. A mask is generated having 1 bits from bit MB through bit 63 and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

NOTE: Software Considerations

rldicl can be used to extract an n-bit field that starts at bit position b in register **r**S, right-justified into register **r**A (clearing the remaining 64-n bits of **r**A), by setting SH=b+n and MB=64-n. It can be used to rotate the contents of a register left (right) by n bits, by setting SH=n (64-n) and MB=0. It can be used to shift the contents of a register right by n bits, by setting SH=64-n and MB=n. It can be used to clear the high-order n bits of a register left (right) by n bits, by setting SH=0 and MB=n.

NOTE: Software Considerations

SH and MB as specified in the assembly syntax are transformed by the assembler when encoded into the instruction such that sh ← $SH_{1:5}$ || SH_0 and mb ← $MB_{1:5}$ || MB_0.

Simplified mnemonics: See Section C.3, “Rotate and Shift Simplified Mnemonics.”

rldicr rldicr

64	User

Rotate Left Doubleword Immediate then Clear Right

rldicr **r**A,**r**S,SH,ME (Rc=0)
rldicr. **r**A,**r**S,SH,ME (Rc=1)

0 5	6 10	11 15	16 20	21 26	27		29	30	31
0 1 1 1 1 0	**r**S	**r**A	sh	me	0	0	1	sh	Rc

```
n ← $sh_5$ || $sh_{0:4}$
r ← $ROTL_{64}$((rS), n)
e ← $me_5$ || $me_{0:4}$
k ← MASK(0,e)
result ← r & k
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are rotated$_{64}$ left SH bits. A mask is generated having 1 bits from bit 0 through bit ME and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

NOTE: Software Considerations

rldicr can be used to extract an n-bit field that starts at bit position b in register **r**S, left-justified into register **r**A (clearing the remaining 64-n bits of **r**A), by setting SH=b and ME=n-1. It can be used to rotate the contents of a register left (right) by n bits, by setting SH=n (64-n) and ME=63. It can be used to shift the contents of a register left by n bits, by setting SH=n and ME=63-n. It can be used to clear the low-order n bits of a register left (right) by n bits, by setting SH=0 and ME=63-n.

NOTE: Software Considerations

SH and ME as specified in the assembly syntax are transformed by the assembler when encoded into the instruction such that sh ← $SH_{1:5} \| SH_0$ and me ← $ME_{1:5} \| ME_0$.

Simplified mnemonics: See Section C.3, “Rotate and Shift Simplified Mnemonics.”

rldimi

64	User

rldimi

Rotate Left Doubleword Immediate then Mask Insert

rldimi **r**A,**r**S,SH,MB (Rc=0)
rldimi. **r**A,**r**S,SH,MB (Rc=1)

0 5	6 10	11 15	16 20	21 26	27		29	30	31
0 1 1 1 1 0	**r**S	**r**A	sh	mb	0	1	1	sh	Rc

```
n ← $sh_5 \| sh_{0:4}$
r ← $ROTL_{64}$((rS), n)
b ← $mb_5 \| mb_{0:4}$
k ← MASK(b,¬n)
result ← r & k | (rA) & ¬k
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are rotated$_{64}$ left SH bits. A mask is generated having 1 bits from bit MB through bit 63-SH and 0 bits elsewhere. The rotated data is inserted into **r**A under control of the generated mask. (If a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register remains unchanged.)

Other registers altered: CR0 (if Rc=1)

NOTE: Software Considerations

rldimi can be used to insert an n-bit field that is right-justified in register **r**S, into register rA starting at bit position b, by setting SH=64-(b+n) and MB=b.

NOTE: Software Considerations

SH and MB as specified in the assembly syntax are transformed by the assembler when encoded into the instruction such that sh ← $SH_{1:5} \| SH_0$ and mb ← $MB_{1:5} \| MB_0$.

Simplified mnemonics: See Section C.3, “Rotate and Shift Simplified Mnemonics.”

rlwimi

Base | User

rlwimi

Rotate Left Word Immediate then Mask Insert

rlwimi **r**A,**r**S,SH,MB,ME (Rc=0)
rlwimi. **r**A,**r**S,SH,MB,ME (Rc=1)

0 5	6 10	11 15	16 20	21 25	26 30	31
0 1 0 1 0 0	**r**S	**r**A	SH	MB	ME	Rc

```
n ← SH
b ← MB+32
e ← ME+32
r ← ROTL$_{32}$((rS)$_{32:63}$,n)
k ← MASK(b,e)
result ← r&k | (rA)&¬k
if Rc=1 then do
   LT  ← result$_{m:63}$ < 0
   GT  ← result$_{m:63}$ > 0
   EQ  ← result$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are rotated$_{32}$ left SH bits. A mask is generated having 1 bits from bit MB+32 through bit ME+32 and 0 bits elsewhere. The rotated data is inserted into **r**A under control of the generated mask. (If a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register remains unchanged.)

Other registers altered: CR0 (if Rc=1)

NOTE: Software Considerations

To insert a k-bit field that is left-justified in **r**S[32–63], into **r**A[32–63] starting at bit position j, by setting SH=64-j, MB=j-32, and ME=(j+k)-33.

To insert an k-bit field that is right-justified in **r**S[32–63], into **r**A[32–63] starting at bit position j, by setting SH=64-(j+k), MB=j-32, and ME=(j+k)-33.

Simplified mnemonics: See Section C.3, “Rotate and Shift Simplified Mnemonics.”

rlwinm

Base	User

rlwinm

Rotate Left Word Immediate then AND with Mask

rlwinm **r**A**,r**S,SH,MB**,**ME (Rc=0)
rlwinm. **r**A**,r**S,SH,MB**,**ME (Rc=1)

0 5	6 10	11 15	16 20	21 25	26 30	31
0 1 0 1 0 1	**r**S	**r**A	SH	MB	ME	Rc

```
n ← SH
b ← MB+32
e ← ME+32
r ← ROTL$_{32}$((rS)$_{32:63}$,n)
k ← MASK(b,e)
result ← r & k
if Rc=1 then do
   LT  ← result$_{m:63}$ < 0
   GT  ← result$_{m:63}$ > 0
   EQ  ← result$_{m:63}$ = 0
   CR0 ← LT ‖ GT ‖ EQ ‖ SO
rA ← result
```

The **r**S contents are rotated$_{32}$ left SH bits. A mask is generated having 1 bits from bit MB+32 through bit ME+32 and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

NOTE: Software Considerations

rlwinm can be used to perform rotations and extractions of the low order 32 bits of a register, clearing the high order 32 bits as follows:

To extract a k-bit field starting at bit position j in **r**S[32–63], right-justified into **r**A[32–63] (clearing the remaining 32–k bits of **r**A[32–63]), set SH=j+k-32, MB=32–k, and ME=31.

To extract a k-bit field that starts at bit position j in **r**S[32–63], left-justified into **r**A[32–63] (clearing the remaining 32–k bits of **r**A[32–63]), set SH=j-32, MB=0, and ME=k–1.

To rotate the contents of bits 32–63 of a register left by k bits, set SH=k, MB=0, and ME=31.

To rotate the contents of bits 32–63 of a register right by k bits, set SH=32–k, MB=0, and ME=31.

To shift the contents of bits 32–63 of a register right by k bits, set SH=32–k, MB=k, and ME=31.

To clear the high-order j bits of the contents of bits 32–63 of a register and then shift the result left by k bits, set SH=k, MB=j–k, and ME=31–k.

To clear the low-order k bits of bits 32–63 of a register, set SH=0, MB=0, and ME=31–k.

Simplified mnemonics: See Section C.3, “Rotate and Shift Simplified Mnemonics.”

rlwnm

Base	User

rlwnm

Rotate Left Word then AND with Mask

rlwnm	**r**A,**r**S,**r**B,MB,ME	(Rc=0)
rlwnm.	**r**A,**r**S,**r**B,MB,ME	(Rc=1)

0 5	6 10	11 15	16 20	21 25	26 30	31
0 1 0 1 1 1	**r**S	**r**A	**r**B	MB	ME	Rc

```
n ← $rB_{59:63}$
b ← MB+32
e ← ME+32
r ← $ROTL_{32}((rS)_{32:63},n)$
k ← MASK(b,e)
result ← r & k
if Rc=1 then do
   LT  ← $result_{m:63} < 0$
   GT  ← $result_{m:63} > 0$
   EQ  ← $result_{m:63} = 0$
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The **r**S contents are rotated_{32} left the number of bits specified by **r**B[59–63]. A mask is generated having 1 bits from bit MB+32 through bit ME+32 and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

NOTE: Software Considerations

rlwnm can be used to perform the same rotations and extractions of the low order 32 bits of a register, clearing the high order 32 bits as rlwinm, except the shift count is taken from the low-order 5 bits of **r**B instead of from SH. See the software note for **rlwinm** for such uses, substituting **r**B[59–63] for SH.

Simplified mnemonics: See Section C.3, “Rotate and Shift Simplified Mnemonics.”

sc

Base, E.HV	User

sc

System Call

sc

sc LEV <E.HV>

0 5	6	20 26	27 29	30	31
0 1 0 0 0 1	///	LEV	///	1	/

If category Embedded.Hypervisor is supported:

```
if LEV = 0 & $MSR_{GS}$ = 0 then do
    if ($MSR_{CM}$ = 0) & ($EPCR_{ICM}$ = 0) then addr ← $^{32}$undefined || $CIA_{32:63}$
    if ($MSR_{CM}$ = 0) & ($EPCR_{ICM}$ = 1) then addr ← $^{32}$0 || $CIA_{32:63}$
    if ($MSR_{CM}$ = 1) & ($EPCR_{ICM}$ = 1) then addr ← $CIA_{32:63}$
    if ($MSR_{CM}$ = 1) & ($EPCR_{ICM}$ = 0) then addr ← undefined
    SRR0 ← addr + 4
    SRR1 ← MSR
    NIA ← $IVPR_{0:47}$ || $IVOR8_{48:59}$ || 0b0000
    newmsr ← $^{32}$0
    $newmsr_{CM}$ ← $EPCR_{ICM}$
    $newmsr_{DE,CE,ME,RI}$ ← $MSR_{DE,CE,ME,RI}$
    MSR ← newmsr

if LEV = 0 & $MSR_{GS}$ = 1 then do
    if ($MSR_{CM}$ = 0) & ($EPCR_{GICM}$ = 0) then addr ← $^{32}$undefined || $CIA_{32:63}$
    if ($MSR_{CM}$ = 0) & ($EPCR_{GICM}$ = 1) then addr ← $^{32}$0 || $CIA_{32:63}$
    if ($MSR_{CM}$ = 1) & ($EPCR_{GICM}$ = 1) then addr ← $CIA_{32:63}$
    if ($MSR_{CM}$ = 1) & ($EPCR_{GICM}$ = 0) then addr ← undefined
    GSRR0 ← addr + 4
    GSRR1 ← MSR
    NIA ← $GIVPR_{0:47}$ || $GIVOR8_{48:59}$ || 0b0000
    newmsr ← $^{32}$0
    $newmsr_{CM}$ ← $EPCR_{GICM}$
    $newmsr_{PMM,UCLE}$ ← $MSR_{PMM,UCLE}$ & $MSRP_{PMMP,UCLEP}$
    $newmsr_{DE,WE,CE,ME,GS,RI}$ ← $MSR_{DE,WE,CE,ME,GS,RI}$
    MSR ← newmsr

if LEV = 1 then do
    if ($MSR_{CM}$ = 0) & ($EPCR_{ICM}$ = 0) then addr ← $^{32}$undefined || $CIA_{32:63}$
    if ($MSR_{CM}$ = 0) & ($EPCR_{ICM}$ = 1) then addr ← $^{32}$0 || $CIA_{32:63}$
    if ($MSR_{CM}$ = 1) & ($EPCR_{ICM}$ = 1) then addr ← $CIA_{32:63}$
    if ($MSR_{CM}$ = 1) & ($EPCR_{ICM}$ = 0) then addr ← undefined
    SRR0 ← addr + 4
    SRR1 ← MSR
    NIA ← $IVPR_{0:47}$ || $IVOR40_{48:59}$ || 0b0000
    newmsr ← $^{32}$0
    $newmsr_{CM}$ ← $EPCR_{ICM}$
    $newmsr_{DE,CE,ME,RI}$ ← $MSR_{DE,CE,ME,RI}$
    MSR ← newmsr
```

If category Embedded.Hypervisor is not supported:

```
    if ($MSR_{CM}$ = 0) & ($EPCR_{ICM}$ = 0) then addr ← $^{32}$undefined || $CIA_{32:63}$
    if ($MSR_{CM}$ = 0) & ($EPCR_{ICM}$ = 1) then addr ← $^{32}$0 || $CIA_{32:63}$
    if ($MSR_{CM}$ = 1) & ($EPCR_{ICM}$ = 1) then addr ← $CIA_{32:63}$
    if ($MSR_{CM}$ = 1) & ($EPCR_{ICM}$ = 0) then addr ← undefined
```

```
SRR0 ← addr + 4
SRR1 ← MSR
NIA ← $IVPR_{0:47}$ || $IVOR8_{48:59}$ || 0b0000
newmsr ← $^{32}0$
$newmsr_{CM}$ ← $EPCR_{ICM}$
$newmsr_{DE,CE,ME,RI}$ ← $MSR_{DE,CE,ME,RI}$
MSR ← newmsr
```

sc is used to request a system service.

If the Embedded.Hypervisor category is not supported then **sc** executes as follows:

- A system call interrupt is generated. The contents of the MSR are copied into SRR1 and the address of the instruction after the **sc** instruction is placed into SRR0.
- The MSR has all bits cleared except:
  - MSR[CM] is set to reflect the interrupt computation mode from EPCR[ICM] <64>.
  - MSR[DE,CE,ME,RI] are left unchanged.
- The interrupt causes the next instruction to be fetched from the system call interrupt handler: IVPR[0–47]||IVOR8[48-59]||0b0000

If the Embedded.Hypervisor category is supported then **sc** executes as follows:

- If the LEV field is 0, a system call interrupt is generated. The system call interrupt is directed to the guest supervisor state if MSR[GS] = 1:
  - The contents of the MSR are copied into GSRR1 and the address of the instruction after the **sc** instruction is placed into GSRR0.
  - The MSR has all bits cleared except:
    - MSR[CM] is set to reflect the interrupt computation mode from EPCR[GICM] <64>.
    - MSR[DE,CE,ME,RI,WE,GS] are left unchanged.
    - MSR[PMM,UCLE] are cleared if the associated protect bits in the MSRP (PMMP,UCLEP) are not set, otherwise they are left unchanged.
  - The interrupt causes the next instruction to be fetched from the guest system call interrupt handler: GIVPR[0–47]||GIVOR8[48-59]||0b0000
- If the LEV field is 0, a system call interrupt is generated. The system call interrupt is directed to the hypervisor state if MSR[GS] = 0:
  - The contents of the MSR are copied into SRR1 and the address of the instruction after the **sc** instruction is placed into SRR0.
  - The MSR has all bits cleared except:
    - MSR[CM] is set to reflect the interrupt computation mode from EPCR[GICM] <64>.
    - MSR[DE,CE,ME,RI] are left unchanged.
  - The interrupt causes the next instruction to be fetched from the system call interrupt handler: IVPR[0–47]||IVOR8[48-59]||0b0000
- If the LEV field is 1, an embedded hypervisor system call interrupt is generated.
  - The contents of the MSR are copied into SRR1 and the address of the instruction after the **sc** instruction is placed into SRR0.

— The MSR has all bits cleared except:
  - MSR[CM] is set to reflect the interrupt computation mode from EPCR[GICM] <64>.
  - MSR[DE,CE,ME,RI] are left unchanged.

— The interrupt causes the next instruction to be fetched from the embedded hypervisor system call interrupt handler: IVPR[0–47]||IVOR40[48-59]||0b0000

sc is context synchronizing. See Section 4.5.4.4, “Context Synchronization.”

Other registers altered: SRR0(GSRR0)   SRR1(GSRR1)   MSR

NOTE: Software Considerations

sc serves as both a basic and an extended mnemonic. The assembler will recognize an **sc** mnemonic with one operand as the basic form, and an **sc** mnemonic with no operand as the extended form. In the extended form, the LEV operand is assumed to be 0.

NOTE: Virtualization

sc with LEV=1 should be used for performing hypercalls to the hypervisor. Note that nothing prevents the execution of such hypercalls from user mode. If the hypervisor wishes to disallow such behavior, it should test the value of SRR1[PR] in the embedded hypervisor system call handler to determine the privilege mode of the caller.

Simplified mnemonics: See Section C.10.9, “System Call (sc).”

sld

64	User

sld

Shift Left Doubleword

sld **r**A,**r**S,**r**B (Rc=0)
sld. **r**A,**r**S,**r**B (Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 0 0 0 1 1 0 1 1	Rc

```
n ← $(rB)_{58:63}$
r ← $ROTL_{64}$((rS),n)
if $(rB)_{57}$=0 then k ← MASK(0,63-n)
else              k ← $^{64}0$
result ← r & k
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are shifted left the number of bits specified by **r**B[57–63]. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The result is placed into **r**A.

Shift amounts from 64 to 127 give a zero result.

Other registers altered: CR0 (if Rc=1)

slw

Base	User

slw

Shift Left Word

slw **r**A,**r**S,**r**B (Rc=0)

slw. **r**A,**r**S,**r**B (Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 0 0 0 1 1 0 0 0	Rc

```
n ← $(rB)_{59:63}$
r ← $ROTL_{32}((rS)_{32:63},n)$
if $(rB)_{58}$=0 then k ← MASK(32,63-n)
else             k ← $^{64}0$
result ← r & k
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S[32–63] are shifted left the number of bits specified by **r**B[58–63]. Bits shifted out of position 32 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into **r**A[32–63]. Bits **r**A[0–31] are cleared.

Shift amounts from 32 to 63 give a zero result.

Other registers altered: CR0 (if Rc=1)

srad

64	User

srad

Shift Right Algebraic Doubleword

srad **r**A,**r**S,**r**B (Rc=0)
srad. **r**A,**r**S,**r**B (Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 1 0 0 0 1 1 0 1 0	Rc

```
n ← $(rB)_{58:63}$
r ← $ROTL_{64}((rS),64-n)$
if $(rB)_{57}$=0 then k ← MASK(n,63)
else             k ← $^{64}0$
s ← $(rS)_0$
result ← r&k | ($^{64}$s)&¬k
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
CA ← s & ((r&¬k)≠0)
```

The contents of **r**S are shifted right the number of bits specified by **r**B[57–63]. Bits shifted out of position 63 are lost. Bit 0 of **r**S is replicated to fill the vacated positions on the left. The result is placed into **r**A.

CA is set if **r**S is negative and any 1 bits are shifted out of bit position 63; otherwise CA is cleared.

A shift amount of zero causes **r**A to be set to the contents of **r**S, and CA to be cleared. Shift amounts from 64 to 127 give a result of 64 sign bits, and cause CA to receive **r**S[0] (that is, sign bit of **r**S).

Other registers altered: CA CR0 (if Rc=1)

sradi

64	User

sradi

Shift Right Algebraic Doubleword Immediate

sradi	**r**A,**r**S,SH	(Rc=0)
sradi.	**r**A,**r**S,SH	(Rc=1)

0 5	6 10	11 15	16 20	21 29	30	31
0 1 1 1 1 1	**r**S	**r**A	sh	1 1 0 0 1 1 1 0 1	sh	Rc

```
n ← $sh_5$ || $sh_{0:4}$
r ← $ROTL_{64}$((rS),64-n)
k ← MASK(n,63)
s ← $(rS)_0$
result ← r&k | ($^{64}$s)&¬k
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
CA ← s & ((r&¬k)≠0)
```

The contents of **r**S are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 0 of **r**S is replicated to fill the vacated positions on the left. The result is placed into **r**A.

CA is set if **r**S is negative and any 1 bits are shifted out of bit position 63; otherwise CA is cleared.

A shift amount of zero causes **r**A to be set to the contents of **r**S, and CA to be cleared.

Other registers altered: CA CR0 (if Rc=1)

sraw sraw

Base	User

Shift Right Algebraic Word

sraw **r**A,**r**S,**r**B (Rc=0)
sraw. **r**A,**r**S,**r**B (Rc=1)

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 1 0 0 0 1 1 0 0 0	Rc

```
n ← (rB)$_{59:63}$
r ← ROTL$_{32}$((rS)$_{32:63}$,64-n)
if (rB)$_{58}$=0 then k ← MASK(n+32,63)
else           k ← $^{64}$0
s ← (rS)$_{32}$
result ← r&k | ($^{64}$s)&¬k
if Rc=1 then do
   LT  ← result$_{m:63}$ < 0
   GT  ← result$_{m:63}$ > 0
   EQ  ← result$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result$_{0:63}$
CA ← s & ((r&¬k)$_{32:63}$≠0)
```

The contents of **r**S[32–63] are shifted right the number of bits specified by **r**B[58–63]. Bits shifted out of position 63 are lost. Bit 32 of **r**S is replicated to fill the vacated positions on the left. The 32-bit result is placed into **r**A[32–63]. **r**S[32] is replicated to fill bits **r**A[0–31].

CA is set if **r**S[32–63] contain a negative value and any 1 bits are shifted out of bit position 63; otherwise CA is cleared.

A shift amount of zero causes **r**A to receive EXTS((**r**S)$_{32:63}$), and CA to be cleared. Shift amounts from 32 to 63 give a result of 64 sign bits, and cause CA to receive **r**S[32] (that is, sign bit of **r**S[32–63]).

Other registers altered: CA CR0 (if Rc=1)

srawi srawi

Base	User

Shift Right Algebraic Word Immediate

srawi **r**A,**r**S,**r**B (Rc=0)
srawi. **r**A,**r**S,**r**B (Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	SH	1 1 0 0 1 1 1 0 0 0	Rc

```
n ← SH
r ← ROTL$_{32}$((rS)$_{32:63}$,64-n)
k ← MASK(n+32,63)
s ← (rS)$_{32}$
result ← r&k | ($^{64}$s)&¬k
if Rc=1 then do
   LT  ← result$_{m:63}$ < 0
   GT  ← result$_{m:63}$ > 0
   EQ  ← result$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
CA ← s & ((r&¬k)$_{32:63}$≠0)
```

The contents of **r**S[32–63] are shifted right SH bits. Bits shifted out of position 63 are lost. Bit 32 of **r**S is replicated to fill the vacated positions on the left. The 32-bit result is placed into **r**A[32–63]. **r**S[32] is replicated to fill bits **r**A[0–31].

CA is set if **r**S[32–63] contain a negative value and any 1 bits are shifted out of bit position 63; otherwise CA is cleared.

A shift amount of zero causes **r**A to receive EXTS((**r**S)$_{32:63}$), and CA to be cleared.

Other registers altered: CA CR0 (if Rc=1)

srd srd

64	User

Shift Right Doubleword

srd **r**A,**r**S,**r**B (Rc=0)
srd. **r**A,**r**S,**r**B (Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 0 0 0 0 1 1 0 1 1	Rc

```
n ← $(rB)_{58:63}$
r ← $ROTL_{64}((rS),64-n)$
if $(rB)_{57}$=0 then k ← MASK(n,63)
else             k ← $^{64}0$
result ← r&k
if Rc=1 then do
    LT  ← $result_{m:63}$ < 0
    GT  ← $result_{m:63}$ > 0
    EQ  ← $result_{m:63}$ = 0
    CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are shifted right the number of bits specified by **r**B[57–63]. Bits shifted out of position 63 are lost. Zeros are supplied to fill the vacated positions on the left. The result is placed into **r**A.

A shift amount of zero causes **r**A to be set to the contents of **r**S. Shift amounts from 64 to 127 give a zero result.

Other registers altered: CR0 (if Rc=1)

srw srw

Base	User

Shift Right Word

srw **r**A,**r**S,**r**B (Rc=0)
srw. **r**A,**r**S,**r**B (Rc=1)

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 0 0 0 0 1 1 0 0 0	Rc

```
n ← $(rB)_{59:63}$
r ← $ROTL_{32}((rS)_{32:63},64-n)$
if $(rB)_{58}$=0 then k ← MASK(n+32,63)
else               k ← $^{64}0$
result ← r & k
if Rc=1 then do
    LT  ← $result_{m:63}$ < 0
    GT  ← $result_{m:63}$ > 0
    EQ  ← $result_{m:63}$ = 0
    CR0 ← LT || GT || EQ || SO
rA ← result
```

The shift count n is the value specified by the contents of **r**B[58–63].

The contents of **r**S[32–63] are shifted right the number of bits specified by **r**B[58–63]. Bits shifted out of position 63 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into **r**A[32–63]. Bits **r**A[0–31] are cleared.

Shift amounts from 32 to 63 give a zero result.

Other registers altered: CR0 (if Rc=1)

stb stb

Base	User

Store Byte

stb **r**S,D(**r**A)

0 5	6 10	11 15	16 31
1 0 0 1 1 0	**r**S	**r**A	D

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + EXTS(D)
MEM(EA,1) ← $(rS)_{56:63}$
```

The contents of **r**S[56–63] are stored into the byte addressed by EA.

Other registers altered: None

stbcx. stbcx.

ER	User

Store Byte Conditional Indexed

stbcx. **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 0 1 0 1 1 0 1 1 0	1

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
if RESERVE then do
   if RESERVE_LENGTH = 1 then do
       if RESERVE_ADDR = real_addr(EA) then do
               perform_store ← 0b1
               undefined_case ← 0
       else
               undefined_case ← 1
   else
       undefined_case ← 1
else
   undefined_case ← 0
   perform_store ← 0b0
if undefined_case then do
   u ← undefined 1-bit value
   if u then MEM(EA,1) ← $(rS)_{56:63}$
   CR0 ← 0b00 || u || $XER_{SO}$
else
   if perform_store then MEM(EA,1) ← $(rS)_{56:63}$
   CR0 ← 0b00 || perform_store || $XER_{SO}$
RESERVE ← 0
```

If a reservation exists, then length associated with the reservation is one byte, and the address specified by the **stbcx.** is the same as that specified by the **lbarx** instruction that established the reservation, the contents of **r**S[56–63] are stored into the byte addressed by EA and the reservation is cleared.

If a reservation exists but the address specified by **stbcx.** is not the same as that specified by the load and reserve instruction that established the reservation, or the length associated with the reservation is not one byte, the reservation is cleared, and it is undefined whether the instruction completes without altering memory.

If a reservation does not exist, the instruction completes without altering memory.

CR field 0 is set to reflect whether the store operation was performed, as follows:

CR0[LT,GT,EQ,SO] = 0b00 || store_performed || XER[SO]

Other registers altered: CR0

NOTE: Software Considerations

Store conditional instructions (**st**x**cx.**) in combination with load and reserve instructions (**l**x**arx)**, permits the programmer to write a sequence of instructions that appear to perform an atomic update operation on a memory location. This operation depends on a single reservation resource in each processor. At most one reservation exists on any given processor.

Because store conditional instructions have implementation dependencies (such as the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (such as, test and set, and compare and swap) needed by application programs. Application programs should use these library programs, rather than use store conditional instructions directly.

The granularity with which reservations are managed is implementation-dependent. Therefore, the memory to be accessed by store conditional instructions should be allocated by a system library program.

When correctly used, the load and reserve and store conditional instructions can provide an atomic update function for a single aligned datum of memory. In general, correct use requires that **lbarx** be paired with **stbcx.**, **lharx** be paired with **sthcx.**, **lwarx** be paired with **stwcx.**, and **ldarx** be paired with **stdcx.** with the same address specified by both instructions of the pair. The only exception is that an unpaired store conditional instruction to any (scratch) effective address can be used to clear any reservation held by the processor. Examples of correct uses of these instructions to emulate primitives such as fetch and add, test and set, and compare and swap can be found in Appendix D, “Programming Examples.”

A reservation is cleared if any of the following events occur:

- The processor holding the reservation executes another load and reserve instruction; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes a store conditional instruction to any address.
- Another processor executes any store instruction to the address associated with the reservation.
- Any mechanism, other than the processor holding the reservation, stores to the address associated with the reservation.

See Section 4.6.1.16, “Atomic Update Primitives Using **lwarx** and **stwcx.**,” for additional information.

stbdx

DS	User

stbdx

Store Byte with Decoration Indexed

stbdx **r**S,**r**A,**r**B

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 0 1 0 0 0 0 0 1 1	/

```
EA ← (rB)
DECORATED_MEM(EA,1,(rA)) ← (rS)$_{56:63}$
```

rS[56–63] are stored to the byte addressed by EA using the decoration supplied by rA. The decoration specified in **r**A is sent to the device corresponding to EA. The behavior of the device with respect to the decoration is implementation specific. See the integrated device manual.

Other registers altered: None

NOTE: Software Considerations

Software must be acutely aware of the effects of operations defined by the decorations supplied and must ensure that the processor only issues operations appropriately. This generally requires addresses to be marked as guarded, caching inhibited and any appropriate memory barriers.

stbepx stbepx

E.PD	Supervisor

Store Byte by External PID Indexed

stbepx **r**S,**r**A,**r**B

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1 1 0 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
MEM(EA,1) ← $(rS)_{56:63}$
```

The contents of **r**S[56–63] are stored into the byte addressed by EA.

This instruction is guest supervisor privileged.

For **stbepx**, the normal translation mechanism is not used. The EPSC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC[EPR] is used in place of MSR[PR]
- EPSC[EAS] is used in place of MSR[DS]
- EPSC[EPID] is used in place of the PID value
- EPSC[EGS] is used in place of MSR[GS] <E.HV>
- EPSC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered: ESR[EPID,ST] (GESR[EPID,ST]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

stbu stbu

Base	User

Store Byte with Update

stbu **r**S,D(**r**A)

0			5	6 10	11 15	16 31
1 0 0	1	1	1	**r**S	**r**A	D

```
EA ← (rA) + EXTS(D)
MEM(EA,1) ← $(rS)_{56:63}$
rA ← EA
```

The contents of **r**S[56–63] are stored into the byte addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered: None

stbux

Base	User

stbux

Store Byte with Update Indexed

stbux **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1 1 1 1 0 1 1 1	/

```
EA ← (rA) + (rB)
MEM(EA,1) ← (rS)$_{56:63}$
rA ← EA
```

The contents of **r**S[56–63] are stored into the byte addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered: None

stbx stbx

Base	User

Store Byte Indexed

stbx **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1 1 0 1 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
MEM(EA,1) ← (rS)$_{56:63}$
rA ← EA
```

The contents of **r**S[56–63] are stored into the byte addressed by EA.

Other registers altered: None

std

64	User

std

Store Doubleword

std **r**S,D(**r**A)

0	5	6	10	11	15	16	29	30	31
1 1 1 1 1 0		**r**S		**r**A		DS		0	0

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + EXTS(DS || 0b00))
MEM(EA,8) ← (rS)
```

The contents of **r**S are stored into the doubleword addressed by EA.

Other registers altered: None

NOTE: Software Considerations

D as specified in the assembly syntax is transformed by the assembler when encoded into the instruction such that DS is formed. The specification of the instruction limits the D field of the instruction to be divisible by 4.

stdbrx stdbrx

64	User

Store Doubleword Byte-Reversed Indexed

stdbrx **r**D**,r**A**,r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	1 0 1 0 0 1 0 1 0 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
MEM(EA,8) ← (rS)$_{56:63}$ ∥ (rS)$_{48:55}$
            ∥ (rS)$_{40:47}$ ∥ (rS)$_{32:39}$
            ∥ (rS)$_{24:31}$ ∥ (rS)$_{16:23}$
            ∥ (rS)$_{8:15}$  ∥ (rS)$_{0:7}$
```

rS[56–63] are stored into bits 0–7 of the doubleword addressed by EA. **r**S[48–55] are stored into bits 8–15 of the doubleword addressed by EA. **r**S[40–47] are stored into bits 16–23 of the doubleword addressed by EA. **r**S[32–39] are stored into bits 24–31 of the doubleword addressed by EA. **r**S[24–31] are stored into bits 32–39 of the doubleword addressed by EA. **r**S[16–23] are stored into bits 40–47 of the doubleword addressed by EA. **r**S[8–15] are stored into bits 48–55 of the doubleword addressed by EA. **r**S[0–7] are stored into bits 56–63 of the doubleword addressed by EA.

Other registers altered: None

NOTE: Software Considerations

When EA references big-endian memory, these instructions have the effect of loading data in little-endian byte order. Likewise, when EA references little-endian memory, these instructions have the effect of loading data in big-endian byte order.

NOTE: Software Considerations

In some implementations, the Store Doubleword Byte-Reverse Indexed instructions may have greater latency than other store instructions.

stdcx. stdcx.

64	User

Store Doubleword Conditional Indexed

stdcx. **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1 1 0 1 0 1 1 0	1

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
if RESERVE then do
   if RESERVE_LENGTH = 8 then do
       if RESERVE_ADDR = real_addr(EA) then do
               perform_store ← 0b1
               undefined_case ← 0
       else
               undefined_case ← 1
   else
       undefined_case ← 1
else
   undefined_case ← 0
   perform_store ← 0b0
if undefined_case then do
   u ← undefined 1-bit value
   if u then MEM(EA,8) ← (rS)
   CR0 ← 0b00 || u || XER$_{SO}$
else
   if perform_store then MEM(EA,8) ← (rS)
   CR0 ← 0b00 || perform_store || XER$_{SO}$
RESERVE ← 0
```

If a reservation exists, then length associated with the reservation is eight bytes, and the address specified by the **stdcx.** is the same as that specified by the **ldarx** instruction that established the reservation, the contents of **r**S are stored into the doubleword addressed by EA and the reservation is cleared.

If a reservation exists but the address specified by **stdcx.** is not the same as that specified by the load and reserve instruction that established the reservation, or the length associated with the reservation is not eight bytes, the reservation is cleared, and it is undefined whether the instruction completes without altering memory.

If a reservation does not exist, the instruction completes without altering memory.

CR field 0 is set to reflect whether the store operation was performed, as follows:

CR0[LT,GT,EQ,SO] = 0b00 || store_performed || XER[SO]

EA must be a multiple of 8. If it is not, an alignment exception occurs.

Other registers altered: CR0

NOTE: Software Considerations

Store conditional instructions (**st**x**cx.**) in combination with load and reserve instructions (**l**x**arx)**, permits the programmer to write a sequence of instructions that appear to perform an atomic update operation on a memory location. This operation depends on a single reservation resource in each processor. At most one reservation exists on any given processor.

Because store conditional instructions have implementation dependencies (such as the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (such as, test and set, and compare and swap) needed by application programs. Application programs should use these library programs, rather than use store conditional instructions directly.

The granularity with which reservations are managed is implementation-dependent. Therefore, the memory to be accessed by store conditional instructions should be allocated by a system library program.

When correctly used, the load and reserve and store conditional instructions can provide an atomic update function for a single aligned datum of memory. In general, correct use requires that **lbarx** be paired with **stbcx.**, **lharx** be paired with **sthcx.**, **lwarx** be paired with **stwcx.**, and **ldarx** be paired with **stdcx.** with the same address specified by both instructions of the pair. The only exception is that an unpaired store conditional instruction to any (scratch) effective address can be used to clear any reservation held by the processor. Examples of correct uses of these instructions to emulate primitives such as fetch and add, test and set, and compare and swap can be found in Appendix D, “Programming Examples.”

A reservation is cleared if any of the following events occur:

- The processor holding the reservation executes another load and reserve instruction; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes a store conditional instruction to any address.
- Another processor executes any store instruction to the address associated with the reservation.
- Any mechanism, other than the processor holding the reservation, stores to the address associated with the reservation.

See Section 4.6.1.16, “Atomic Update Primitives Using **lwarx** and **stwcx.**,” for additional information.

stddx stddx

64, DS	User

Store Doubleword with Decoration Indexed

stddx **r**S**,r**A**,r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 0 1 1 1 0 0 0 1 1	/

```
EA ← (rB)
DECORATED_MEM(EA,8,(rA)) ← (rS)
```

rS is stored to the doubleword addressed by EA using the decoration supplied by rA. The decoration specified in **r**A is sent to the device corresponding to EA. The behavior of the device with respect to the decoration is implementation specific. See the integrated device manual.

Other registers altered: None

NOTE: Software Considerations

Software must be acutely aware of the effects of operations defined by the decorations supplied and must ensure that the processor only issues operations appropriately. This generally requires addresses to be marked as guarded, caching inhibited and any appropriate memory barriers.

Software should ensure that accesses are aligned, otherwise atomic accesses are not guaranteed and may require multiple accesses to perform the operation which is not likely to produce the intended result.

stdepx stdepx

64, E.PD	Supervisor

Store Doubleword by External PID Indexed

stdepx **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1 0 0 1 1 1 0 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
MEM(EA,8) ← (rS)
```

The contents of **r**S are stored into the doubleword addressed by EA.

This instruction is guest supervisor privileged.

For **stdepx**, the normal translation mechanism is not used. The EPSC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC[EPR] is used in place of MSR[PR]
- EPSC[EAS] is used in place of MSR[DS]
- EPSC[EPID] is used in place of the PID value
- EPSC[EGS] is used in place of MSR[GS] <E.HV>
- EPSC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered: ESR[EPID,ST] (GESR[EPID,ST]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

stdu

64	User

stdu

Store Doubleword with Update

stdu **r**S,D(**r**A)

0–5	6–10	11–15	16–29	30	31
1 1 1 1 1 0	**r**S	**r**A	DS	0	1

```
EA ← (rA) + EXTS(DS || 0b00))
MEM(EA,8) ← (rS)
rA ← EA
```

The contents of **r**S are stored into the doubleword addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered: None

NOTE: Software Considerations

D as specified in the assembly syntax is transformed by the assembler when encoded into the instruction such that DS is formed. The specification of the instruction limits the D field of the instruction to be divisible by 4.

stdux stdux

64	User

Store Doubleword with Update Indexed

stdux **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1 0 1 1 0 1 0 1	/

```
EA ← (rA) + (rB)
MEM(EA,8) ← (rS)
rA ← EA
```

The contents of **r**S are stored into the doubleword addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered: None

stdx

64	User

stdx

Store Doubleword Indexed

stdx **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1 0 0 1 0 1 0 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
MEM(EA,8) ← (rS)
```

The contents of **r**S are stored into the doubleword addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered: None

stfd stfd

FP	User

Store Floating-Point Double

stfd **fr**S,D(**r**A)

0 5	6 10	11 15	16 31
1 1 0 1 1 0	**fr**S	**r**A	D

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← (a + EXTS(D))
MEM(EA,8) ← (frS)
```

The contents of **fr**S are stored into the doubleword addressed by EA.

If MSR[FP]=0, an attempt to execute **stfd** causes a floating-point unavailable interrupt.

Other registers altered: None

stfddx

FP,DS	User

stfddx

Store Floating-Point Double with Decoration Indexed

stfddx **fr**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**fr**S	**r**A	**r**B	1 1 1 0 1 0 0 0 1 1	/

```
EA ← (rB)
DECORATED_MEM(EA,8,(rA)) ← (frS)
```

frS is stored to the doubleword addressed by EA using the decoration supplied by **r**A. The decoration specified in **r**A is sent to the device corresponding to EA. The behavior of the device with respect to the decoration is implementation specific. See the integrated device manual.

If MSR[FP]=0, an attempt to execute **stfddx** causes a floating-point unavailable interrupt.

Other registers altered: None

NOTE: Software Considerations

Software must be acutely aware of the effects of operations defined by the decorations supplied and must ensure that the processor only issues operations appropriately. This generally requires addresses to be marked as guarded, caching inhibited and any appropriate memory barriers.

Software should ensure that accesses are aligned, otherwise atomic accesses are not guaranteed and may require multiple accesses to perform the operation which is not likely to produce the intended result.

stfdepx

FP, E.PD	Supervisor

stfdepx

Store Floating-Point Double by External PID Indexed

stfdepx **fr**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**fr**S	**r**A	**r**B	1 0 1 1 0 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← (a + (rB))
MEM(EA,8) ← (frS)
```

The contents of **fr**S are stored into the doubleword addressed by EA.

If MSR[FP]=0, an attempt to execute **stfdepx** causes a floating-point unavailable interrupt.

This instruction is guest supervisor privileged.

For **stfdepx**, the normal translation mechanism is not used. The EPSC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC[EPR] is used in place of MSR[PR]
- EPSC[EAS] is used in place of MSR[DS]
- EPSC[EPID] is used in place of the PID value
- EPSC[EGS] is used in place of MSR[GS] <E.HV>
- EPSC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered: ESR[EPID,ST,FP] (GESR[EPID,ST,FP]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

stfdu stfdu

FP	User

Store Floating-Point Double with Update

stfdu **fr**S,D(**r**A)

0 5	6 10	11 15	16 31
1 1 0 1 1 1	**fr**S	**r**A	D

```
EA ← ((rA) + EXTS(D))
MEM(EA,8) ← (frS)
rA ← EA
```

The contents of **fr**S are stored into the doubleword addressed by EA.

EA is placed into **r**A.

If **r**A=0, the instruction form is invalid.

If MSR[FP]=0, an attempt to execute **stfdu** causes a floating-point unavailable interrupt.

Other registers altered: None

stfdux

FP	User

stfdux

Store Floating-Point Double with Update Indexed

stfdux **fr**S,**r**A,**r**B

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	**fr**S	**r**A	**r**B	1 0 1 1 1 1 0 1 1 1	/

```
EA ← ((rA) + (rB))
MEM(EA,8) ← (frS)
rA ← EA
```

The contents of **fr**S are stored into the doubleword addressed by EA.

EA is placed into **r**A.

If **r**A=0, the instruction form is invalid.

If MSR[FP]=0, an attempt to execute **stfdux** causes a floating-point unavailable interrupt.

Other registers altered: None

stfdx stfdx

FP	User

Store Floating-Point Double Indexed

stfdx **fr**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**fr**S	**r**A	**r**B	1 0 1 1 0 1 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + (rB))
MEM(EA,8) ← (frS)
```

The contents of **fr**S are stored into the doubleword addressed by EA.

If MSR[FP]=0, an attempt to execute **stfdx** causes a floating-point unavailable interrupt.

Other registers altered: None

stfiwx stfiwx

FP	User

Store Floating-Point as Integer Word Indexed

stfiwx **fr**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**fr**S	**r**A	**r**B	1 1 1 1 0 1 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + (rB))
MEM(EA,4) ← $(frS)_{32:63}$
```

The contents of **fr**S[32–63] are stored, without conversion, into the word addressed by EA.

If the contents of **fr**S were produced, either directly or indirectly, by a load floating-point single instruction, a single-precision arithmetic instruction, or **frsp**, the value stored is undefined. (The contents of **fr**S are produced directly by such an instruction if **fr**S is the target register for the instruction. The contents of **fr**S are produced indirectly by such an instruction if **fr**S is the final target register of a sequence of one or more floating-point move instructions, with the input to the sequence having been produced directly by such an instruction.)

If MSR[FP]=0, an attempt to execute **stfiwx** causes a floating-point unavailable interrupt.

Other registers altered: None

stfs

FP	User

stfs

Store Floating-Point Single

stfs **fr**S,D(**r**A)

0 4	5	6 10	11 15	16 31
1 1 0 1 0	0	**fr**S	**r**A	D

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + EXTS(D))
MEM(EA,4) ← SINGLE((frS))
```

The contents of **fr**S are converted to single format and stored into the word addressed by EA.

If MSR[FP]=0, an attempt to execute **stfs** causes a floating-point unavailable interrupt.

Other registers altered: None

stfsu stfsu

FP	User

Store Floating-Point Single with Update

stfsu **fr**S,D(**r**A)

0 4 5	6 10	11 15	16 31
1 1 0 1 0 1	**fr**S	**r**A	D

```
EA ← ((rA) + EXTS(D))
MEM(EA,4) ← SINGLE((frS))
rA ← EA
```

The contents of **fr**S are converted to single format and stored into the word addressed by EA.

EA is placed into **r**A.

If **r**A=0, the instruction form is invalid.

If MSR[FP]=0, an attempt to execute **stfsu** causes a floating-point unavailable interrupt.

Other registers altered: None

stfsux

FP	User

stfsux

Store Floating-Point Single with Update Indexed

stfsux **fr**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**fr**S	**r**A	**r**B	1 0 1 0 1 1 0 1 1 1	/

```
EA ← ((rA) + (rB))
MEM(EA,4) ← SINGLE((frS))
rA ← EA
```

The contents of **fr**S are converted to single format and stored into the word addressed by EA.

EA is placed into **r**A.

If **r**A=0, the instruction form is invalid.

If MSR[FP]=0, an attempt to execute **stfsux** causes a floating-point unavailable interrupt.

Other registers altered: None

stfsx stfsx

FP	User

Store Floating-Point Single Indexed

stfsx **fr**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**fr**S	**r**A	**r**B	1 0 1 0 0 1 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + (rB))
MEM(EA,4) ← SINGLE((frS))
```

The contents of **fr**S are converted to single format and stored into the word addressed by EA.

If MSR[FP]=0, an attempt to execute **stfsx** causes a floating-point unavailable interrupt.

Other registers altered: None

sth sth

Base	User

Store Half Word

sth **r**S,D(**r**A)

0 4 5	6 10	11 15	16 31
1 0 1 1 0 0	**r**S	**r**A	D

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← ((rA) + EXTS(D))
MEM(EA,2) ← $(rS)_{48:63}$
```

The contents of **r**S[48–63] are stored into the half word addressed by EA.

Other registers altered: None

sthbrx

64	User

sthbrx

Store Halfword Byte-Reversed Indexed

sthbrx **r**D**,r**A**,r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 1 1 0 0 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
MEM(EA,2) ← $(rS)_{56:63}$ || $(rS)_{48:55}$
```

rS[56–63] are stored into bits 0–7 of the half word addressed by EA. Bits 48–55 of **r**S are stored into bits 8–15 of the half word addressed by EA.

Other registers altered: None

NOTE: Software Considerations

When EA references big-endian memory, these instructions have the effect of loading data in little-endian byte order. Likewise, when EA references little-endian memory, these instructions have the effect of loading data in big-endian byte order.

NOTE: Software Considerations

In some implementations, the Store Halfword Byte-Reverse Indexed instructions may have greater latency than other store instructions.

sthcx.

ER	User

sthcx.

Store Halfword Conditional Indexed

sthcx. **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 0 1 1 0 1 0 1 1 0	1

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← a + (rB)
if RESERVE then do
   if RESERVE_LENGTH = 2 then do
       if RESERVE_ADDR = real_addr(EA) then do
               perform_store ← 0b1
               undefined_case ← 0
       else
               undefined_case ← 1
   else
       undefined_case ← 1
else
   undefined_case ← 0
   perform_store ← 0b0
if undefined_case then do
   u ← undefined 1-bit value
   if u then MEM(EA,2) ← (rS)$_{48:63}$
   CR0 ← 0b00 || u || XER$_{SO}$
else
   if perform_store then MEM(EA,2) ← (rS)$_{48:63}$
   CR0 ← 0b00 || perform_store || XER$_{SO}$
RESERVE ← 0
```

If a reservation exists, then length associated with the reservation is two bytes, and the address specified by the **sthcx.** is the same as that specified by the **lharx** instruction that established the reservation, the contents of **r**S[48–63] are stored into the halfword addressed by EA and the reservation is cleared.

If a reservation exists but the address specified by **sthcx.** is not the same as that specified by the load and reserve instruction that established the reservation, or the length associated with the reservation is not two bytes, the reservation is cleared, and it is undefined whether the instruction completes without altering memory.

If a reservation does not exist, the instruction completes without altering memory.

CR field 0 is set to reflect whether the store operation was performed, as follows:

CR0[LT,GT,EQ,SO] = 0b00 || store_performed || XER[SO]

EA must be a multiple of 2. If it is not, an alignment exception occurs.

Other registers altered: CR0

NOTE: Software Considerations

Store conditional instructions (**st**x**cx.**) in combination with load and reserve instructions (**l**x**arx)**, permits the programmer to write a sequence of instructions that appear to perform an atomic update operation on a memory location. This operation depends on a single reservation resource in each processor. At most one reservation exists on any given processor.

Because store conditional instructions have implementation dependencies (such as the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (such as, test and set, and compare and swap) needed by application programs. Application programs should use these library programs, rather than use store conditional instructions directly.

The granularity with which reservations are managed is implementation-dependent. Therefore, the memory to be accessed by store conditional instructions should be allocated by a system library program.

When correctly used, the load and reserve and store conditional instructions can provide an atomic update function for a single aligned datum of memory. In general, correct use requires that **lbarx** be paired with **stbcx.**, **lharx** be paired with **sthcx.**, **lwarx** be paired with **stwcx.**, and **ldarx** be paired with **stdcx.** with the same address specified by both instructions of the pair. The only exception is that an unpaired store conditional instruction to any (scratch) effective address can be used to clear any reservation held by the processor. Examples of correct uses of these instructions to emulate primitives such as fetch and add, test and set, and compare and swap can be found in Appendix D, “Programming Examples.”

A reservation is cleared if any of the following events occur:

- The processor holding the reservation executes another load and reserve instruction; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes a store conditional instruction to any address.
- Another processor executes any store instruction to the address associated with the reservation.
- Any mechanism, other than the processor holding the reservation, stores to the address associated with the reservation.

See Section 4.6.1.16, “Atomic Update Primitives Using **lwarx** and **stwcx.**,” for additional information.

sthdx

DS	User

sthdx

Store Halfword with Decoration Indexed

sthdx **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 0 1 0 1 0 0 0 1 1	/

```
EA ←(rB)
DECORATED_MEM(EA,2,(rA)) ← (rS)$_{48:63}$
```

rS[48–63] are stored to the halfword addressed by EA using the decoration supplied by rA. The decoration specified in **r**A is sent to the device corresponding to EA. The behavior of the device with respect to the decoration is implementation specific. See the integrated device manual.

Other registers altered: None

NOTE: Software Considerations

Software must be acutely aware of the effects of operations defined by the decorations supplied and must ensure that the processor only issues operations appropriately. This generally requires addresses to be marked as guarded, caching inhibited and any appropriate memory barriers.

Software should ensure that accesses are aligned, otherwise atomic accesses are not guaranteed and may require multiple accesses to perform the operation which is not likely to produce the intended result.

sthepx sthepx

E.PD	Supervisor

Store Halfword by External PID Indexed

sthepx **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1 1 0 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
MEM(EA,2) ← $(rS)_{48:63}$
```

The contents of **r**S[48–63] are stored into the halfword addressed by EA.

This instruction is guest supervisor privileged.

For **sthepx**, the normal translation mechanism is not used. The EPSC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC[EPR] is used in place of MSR[PR]
- EPSC[EAS] is used in place of MSR[DS]
- EPSC[EPID] is used in place of the PID value
- EPSC[EGS] is used in place of MSR[GS] <E.HV>
- EPSC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered: ESR[EPID,ST] (GESR[EPID,ST]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

sthu sthu

Base	User

Store Half Word with Update

sthu **r**S,D(**r**A)

0–5	6–10	11–15	16–31
1 0 1 1 0 1	**r**S	**r**A	D

```
EA ← ((rA) + EXTS(D))
MEM(EA,2) ← $(rS)_{48:63}$
rA ← EA
```

The contents of **r**S[48–63] are stored into the half word addressed by EA.

EA is placed into **r**A.

If **r**A=0, the instruction form is invalid.

Other registers altered: None

sthux

Base	User

sthux

Store Half Word with Update Indexed

sthux **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 1 1 0 1 1 0 1 1 1	/

```
EA ← ((rA) + (rB))
MEM(EA,2) ← $(rS)_{48:63}$
rA ← EA
```

The contents of **r**S[48–63] are stored into the half word addressed by EA.

EA is placed into **r**A.

If **r**A=0, the instruction form is invalid.

Other registers altered: None

sthx

Base	User

sthx

Store Half Word Indexed

sthx **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 1 1 0 0 1 0 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
MEM(EA,2) ← $(rS)_{48:63}$
```

The contents of **r**S[48–63] are stored into the half word addressed by EA.

Other registers altered: None

sthu sthu

Base	User

Store Half Word with Update

sthu **r**S,D(**r**A)

0–5	6–10	11–15	16–31
1 0 1 1 0 1	**r**S	**r**A	D

```
EA ← ((rA) + EXTS(D))
MEM(EA,2) ← $(rS)_{48:63}$
rA ← EA
```

The contents of **r**S[48–63] are stored into the half word addressed by EA.

EA is placed into **r**A.

If **r**A=0, the instruction form is invalid.

Other registers altered: None

stmw

Base	User

Store Multiple Word

stmw **r**S,D(**r**A)

0 5	6 10	11 15	16 31
1 0 1 1 1 1	**r**S	**r**A	D

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← (a + EXTS(D))
r ← rS
do while r ≤ 31
    MEM(EA,4) ← GPR(r)$_{32:63}$
    r  ← r + 1
    EA ← EA + 4
```

Let n = (32-**r**S). n consecutive words starting at EA are stored from the low-order 32 bits of GPRs **r**S through 31.

EA must be a multiple of 4. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Other registers altered: None

stvepx stvepx

E.PD, V	Supervisor

Store Vector by External PID Indexed

stvepx **vS,r**A**,r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**vS**	**r**A	**r**B	1 1 0 0 1 0 0 1 1 1	/

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← a + (rB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0,16)← (vS)
```

The contents of **vS** are stored into the quadword addressed by EA truncated to the nearest quadword boundary. The data is stored in big-endian form regardless of the setting of the E storage attribute.

This instruction is guest supervisor privileged.

An attempt to execute **stvepx** while MSR[SPV] = 0 causes an AltiVec unavailable interrupt.

For **stvepx**, the normal translation mechanism is not used. The EPSC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC[EPR] is used in place of MSR[PR]
- EPSC[EAS] is used in place of MSR[DS]
- EPSC[EPID] is used in place of the PID value
- EPSC[EGS] is used in place of MSR[GS] <E.HV>
- EPSC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID,ST,SPV] (GESR[EPID,ST,SPV]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

stvepxl

E.PD, V	Supervisor

stvepxl

Store Vector by External PID Indexed

stvepxl **v**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**v**S	**r**A	**r**B	1 1 0 0 0 0 0 1 1 1	/

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← a + (rB)
MEM(EA & 0xFFFF_FFFF_FFFF_FFF0,16)← (vS)
```

The contents of **vSvS** are stored into the quadword addressed by EA truncated to the nearest quadword boundary. The data is stored in big-endian form regardless of the setting of the E storage attribute.

The **stvepxl** instruction provides a hint that the quadword addressed by EA will probably not be needed again by the program in the near future.

This instruction is guest supervisor privileged.

An attempt to execute **stvepxl** while MSR[SPV] = 0 causes an AltiVec unavailable interrupt.

For **stvepxl**, the normal translation mechanism is not used. The EPSC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC[EPR] is used in place of MSR[PR]
- EPSC[EAS] is used in place of MSR[DS]
- EPSC[EPID] is used in place of the PID value
- EPSC[EGS] is used in place of MSR[GS] <E.HV>
- EPSC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered:

- ESR[EPID,ST,SPV] (GESR[EPID,ST,SPV]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

stw stw

Base	User

Store Word

stw **r**S,D(**r**A)

0–5	6–10	11–15	16–31
1 0 0 1 0 0	**r**S	**r**A	D

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + EXTS(D))
MEM(EA,4) ← $(rS)_{32:63}$
```

The contents of **r**S[32–63] are stored into the word addressed by EA.

Other registers altered: None

stwbrx

Base	User

stwbrx

Store Word Byte-Reverse Indexed

stwbrx **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 0 1 0 0 1 0 1 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + (rB))
MEM(EA,4) ← $(rS)_{56:63} \| (rS)_{48:55} \| (rS)_{40:47} \| (rS)_{32:39}$
```

Bits 56–63 of **r**S are stored into bits 0–7 of the word addressed by EA. Bits 48–55 of **r**S are stored into bits 8–15 of the word addressed by EA. Bits 40–47 of **r**S are stored into bits 16–23 of the word addressed by EA. Bits 32–39 of **r**S are stored into bits 24–31 of the word addressed by EA.

Other registers altered: None

NOTE: Software Considerations

When EA references big-endian memory, these instructions have the effect of loading data in little-endian byte order. Likewise, when EA references little-endian memory, these instructions have the effect of loading data in big-endian byte order.

NOTE: Software Considerations

In some implementations, the Store Word Byte-Reverse Indexed instructions may have greater latency than other store instructions.

stwcx. stwcx.

Base	User

Store Word Conditional Indexed

stwcx. **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21		30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1	0 0 1 0	1 1 0	1

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
if RESERVE then do
   if RESERVE_LENGTH = 4 then do
       if RESERVE_ADDR = real_addr(EA) then do
               perform_store ← 0b1
               undefined_case ← 0
       else
               undefined_case ← 1
   else
       undefined_case ← 1
else
   undefined_case ← 0
   perform_store ← 0b0
if undefined_case then do
   u ← undefined 1-bit value
   if u then MEM(EA,4) ← (rS)$_{32:63}$
   CR0 ← 0b00 || u || XER$_{SO}$
else
   if perform_store then MEM(EA,4) ← (rS)$_{32:63}$
   CR0 ← 0b00 || perform_store || XER$_{SO}$
RESERVE ← 0
```

If a reservation exists, then length associated with the reservation is four bytes, and the address specified by the **stwcx.** is the same as that specified by the **lwarx** instruction that established the reservation, the contents of **r**S[32–63] are stored into the word addressed by EA and the reservation is cleared.

If a reservation exists but the address specified by **stwcx.** is not the same as that specified by the load and reserve instruction that established the reservation, or the length associated with the reservation is not four bytes, the reservation is cleared, and it is undefined whether the instruction completes without altering memory.

If a reservation does not exist, the instruction completes without altering memory.

CR field 0 is set to reflect whether the store operation was performed, as follows:

CR0[LT,GT,EQ,SO] = 0b00 || store_performed || XER[SO]

EA must be a multiple of 4. If it is not, an alignment exception occurs.

Other registers altered: CR0

NOTE: Software Considerations

Store conditional instructions (**st**x**cx.**) in combination with load and reserve instructions (**l**x**arx)**, permits the programmer to write a sequence of instructions that appear to perform an atomic update operation on a memory location. This operation depends on a single reservation resource in each processor. At most one reservation exists on any given processor.

Because store conditional instructions have implementation dependencies (such as the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (such as, test and set, and compare and swap) needed by application programs. Application programs should use these library programs, rather than use store conditional instructions directly.

The granularity with which reservations are managed is implementation-dependent. Therefore, the memory to be accessed by store conditional instructions should be allocated by a system library program.

When correctly used, the load and reserve and store conditional instructions can provide an atomic update function for a single aligned datum of memory. In general, correct use requires that **lbarx** be paired with **stbcx.**, **lharx** be paired with **sthcx.**, **lwarx** be paired with **stwcx.**, and **ldarx** be paired with **stdcx.** with the same address specified by both instructions of the pair. The only exception is that an unpaired store conditional instruction to any (scratch) effective address can be used to clear any reservation held by the processor. Examples of correct uses of these instructions to emulate primitives such as fetch and add, test and set, and compare and swap can be found in Appendix D, “Programming Examples.”

A reservation is cleared if any of the following events occur:

- The processor holding the reservation executes another load and reserve instruction; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes a store conditional instruction to any address.
- Another processor executes any store instruction to the address associated with the reservation.
- Any mechanism, other than the processor holding the reservation, stores to the address associated with the reservation.

See Section 4.6.1.16, “Atomic Update Primitives Using **lwarx** and **stwcx.**,” for additional information.

stwdx

64, DS	User

stwdx

Store Word with Decoration Indexed

stwdx **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	1 0 1 1 0 0 0 0 1 1	/

```
EA ← (rB)
DECORATED_MEM(EA,4,(rA)) ← (rS)$_{32:63}$
```

rS[32–63] is stored to the word addressed by EA using the decoration supplied by rA. The decoration specified in **r**A is sent to the device corresponding to EA. The behavior of the device with respect to the decoration is implementation specific. See the integrated device manual.

Other registers altered: None

NOTE: Software Considerations

Software must be acutely aware of the effects of operations defined by the decorations supplied and must ensure that the processor only issues operations appropriately. This generally requires addresses to be marked as guarded, caching inhibited and any appropriate memory barriers.

Software should ensure that accesses are aligned, otherwise atomic accesses are not guaranteed and may require multiple accesses to perform the operation which is not likely to produce the intended result.

stwepx

E.PD	Supervisor

stwepx

Store Word by External PID Indexed

stwepx **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1 0 0 1 1 1 1 1	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← a + (rB)
MEM(EA,4) ← $(rS)_{32:63}$
```

The contents of **r**S[32–63] are stored into the word addressed by EA.

This instruction is guest supervisor privileged.

For **stwepx**, the normal translation mechanism is not used. The EPSC contents are used to provide the context in which translation occurs. The following substitutions are made for just the translation and access control process:

- EPSC[EPR] is used in place of MSR[PR]
- EPSC[EAS] is used in place of MSR[DS]
- EPSC[EPID] is used in place of the PID value
- EPSC[EGS] is used in place of MSR[GS] <E.HV>
- EPSC[ELPID] is used in place of LPIDR <E.HV>

Other registers altered: ESR[EPID,ST] (GESR[EPID,ST]<E.HV>) if a data TLB error interrupt or a data storage interrupt occurs

stwu stwu

Base	User

Store Word with Update

stwu **r**S,D(**r**A)

0–5	6–10	11–15	16–31
1 0 0 1 0 1	**r**S	**r**A	D

```
EA ← ((rA) + EXTS(D))
MEM(EA,4) ← (rS)$_{32:63}$
rA ← EA
```

The contents of **r**S[32–63] are stored into the word addressed by EA.

EA is placed into **r**A.

If **r**A=0, the instruction form is invalid.

Other registers altered: None

stwux

Base	User

stwux

Store Word with Update Indexed

stwux **r**S,**r**A,**r**B

0 … 5	6 … 10	11 … 15	16 … 20	21 … 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1 0 1 1 0 1 1 1	/

```
EA ← ((rA) + (rB))
MEM(EA,4) ← $(rS)_{32:63}$
rA ← EA
```

The contents of **r**S[32–63] are stored into the word addressed by EA.

EA is placed into **r**A.

If **r**A=0, the instruction form is invalid.

Other registers altered: None

stwx stwx

Base	User

Store Word Indexed

stwx **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 0 1 0 0 1 0 1 1 1	/

```
if rA=0 then a ← $^{64}$0 else a ← (rA)
EA ← (a + (rB)))
MEM(EA,4) ← (rS)$_{32:63}$
```

The contents of **r**S[32–63] are stored into the word addressed by EA.

Other registers altered: None

subf subf

Base	User

Subtract From

subf **r**D,**r**A,**r**B (OE=0, Rc=0)
subf. **r**D,**r**A,**r**B (OE=0, Rc=1)
subfo **r**D,**r**A,**r**B (OE=1, Rc=0)
subfo. **r**D,**r**A,**r**B (OE=1, Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	OE	0 0 0 1 0 1 0 0 0	Rc

```
carry$_{0:63}$ ← Carry(¬(rA) + (rB) + 1)
sum$_{0:63}$   ←       ¬(rA) + (rB) + 1
if OE=1 then do
   OV   ← carry$_{m}$ ⊕ carry$_{m+1}$
   SO   ← SO | OV
if Rc=1 then do
   LT  ← sum$_{m:63}$ < 0
   GT  ← sum$_{m:63}$ > 0
   EQ  ← sum$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rD ← sum
```

The sum of the one’s complement of the contents of **r**A, the contents of **r**B, and 1 is placed into **r**D.

Other registers altered:

- CR0 (if Rc=1)
  SO OV (if OE=1)

Simplified mnemonics: See Section C.2.2, “Subtract.”

subfc

Base	User

subfc

Subtract From Carrying

subfc	**r**D,**r**A,**r**B	(OE=0, Rc=0)
subfc.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
subfco	**r**D,**r**A,**r**B	(OE=1, Rc=0)
subfco.	**r**D,**r**A,**r**B	(OE=1, Rc=1)

0 ... 5	6 ... 10	11 ... 15	16 ... 20	21	22 ... 30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	OE	0 0 0 0 0 1 0 0 0	Rc

```
carry$_{0:63}$ ← Carry(¬(rA) + (rB) + 1)
sum$_{0:63}$   ←       ¬(rA) + (rB) + 1
if OE=1 then do
   OV   ← carry$_m$ ⊕ carry$_{m+1}$
   SO   ← SO | OV
if Rc=1 then do
   LT  ← sum$_{m:63}$ < 0
   GT  ← sum$_{m:63}$ > 0
   EQ  ← sum$_{m:63}$ = 0
   CR0 ← LT ‖ GT ‖ EQ ‖ SO
rD ← sum
CA ← carry$_m$
```

The sum of the one’s complement of the contents of **r**A, the contents of **r**B, and 1 is placed into **r**D.

Other registers altered:

- CA
  CR0 (if Rc=1)
  SO OV (if OE=1)

Simplified mnemonics: See Section C.2.2, “Subtract.”

subfe subfe

Base	User

Subtract From Extended

subfe	**r**D,**r**A,**r**B	(OE=0, Rc=0)
subfe.	**r**D,**r**A,**r**B	(OE=0, Rc=1)
subfeo	**r**D,**r**A,**r**B	(OE=1, Rc=0)
subfeo.	**r**D,**r**A,**r**B	(OE=1, Rc=1)

0–5	6–10	11–15	16–20	21	22–30	31
0 1 1 1 1 1	**r**D	**r**A	**r**B	OE	0 1 0 0 0 1 0 0 0	Rc

```
$carry_{0:63}$ ← Carry(¬(rA) + (rB) + CA)
$sum_{0:63}$   ←       ¬(rA) + (rB) + CA
if OE=1 then do
   OV   ← $carry_m$ ⊕ $carry_{m+1}$
   SO   ← SO | OV
if Rc=1 then do
   LT  ← $sum_{m:63}$ < 0
   GT  ← $sum_{m:63}$ > 0
   EQ  ← $sum_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rD ← sum
CA ← $carry_m$
```

The sum of the one’s complement of the contents of **r**A, the contents of **r**B, and CA is placed into **r**D.

Other registers altered:

- CA
  CR0 (if Rc=1)
  SO OV (if OE=1)

subfic

Base	User

Subtract From Immediate Carrying

subfic **r**D,**r**A,SIMM

0 5	6 10	11 15	16 31
0 0 1 0 0 0	**r**D	**r**A	SIMM

```
$carry_{0:63} \gets Carry(\neg(rA) + EXTS(SIMM) + 1)$
$sum_{0:63}   \gets       \neg(rA) + EXTS(SIMM) + 1$
$rD \gets sum$
$CA \gets carry_{m}$
```

The sum of the one’s complement of the contents of **r**A, the sign-extended value of the SIMM field, and 1 is placed into **r**D.

Other registers altered: CA

subfme

Base	User

subfme

Subtract From Minus One Extended

subfme	**r**D,**r**A	(OE=0, Rc=0)
subfme.	**r**D,**r**A	(OE=0, Rc=1)
subfmeo	**r**D,**r**A	(OE=1, Rc=0)
subfmeo.	**r**D,**r**A	(OE=1, Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	///	OE	0 1 1 1 0 1 0 0 0	Rc

```
carry$_{0:63}$ ← Carry(¬(rA) + CA + 0xFFFF_FFFF_FFFF_FFFF)
sum$_{0:63}$   ←       ¬(rA) + CA + 0xFFFF_FFFF_FFFF_FFFF
if OE=1 then do
   OV   ← carry$_{m}$ ⊕ carry$_{m+1}$
   SO   ← SO | OV
if Rc=1 then do
   LT  ← sum$_{m:63}$ < 0
   GT  ← sum$_{m:63}$ > 0
   EQ  ← sum$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rD ← sum
CA ← carry$_{m}$
```

The sum of CA, $^{64}1$, and the one’s complement of the contents of **r**A is placed into **r**D.

Other registers altered:

- CA
  CR0 (if Rc=1)
  SO OV (if OE=1)

subfze

Base	User

subfze

Subtract From Zero Extended

subfze	**r**D,**r**A	(OE=0, Rc=0)
subfze.	**r**D,**r**A	(OE=0, Rc=1)
subfzeo	**r**D,**r**A	(OE=1, Rc=0)
subfzeo.	**r**D,**r**A	(OE=1, Rc=1)

0 5	6 10	11 15	16 20	21	22 30	31
0 1 1 1 1 1	**r**D	**r**A	///	OE	0 1 1 0 0 1 0 0 0	Rc

```
carry$_{0:63}$ ← Carry(¬(rA) + CA)
sum$_{0:63}$   ←       ¬(rA) + CA
if OE=1 then do
   OV   ← carry$_m$ ⊕ carry$_{m+1}$
   SO   ← SO | OV
if Rc=1 then do
   LT  ← sum$_{m:63}$ < 0
   GT  ← sum$_{m:63}$ > 0
   EQ  ← sum$_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rD ← sum
CA ← carry$_m$
```

The sum of the one’s complement of the contents of **r**A and CA is placed into **r**D.

Other registers altered:

- CA
  CR0 (if Rc=1)
  SO OV (if OE=1)

sync

Base	User

sync

Synchronize

sync L

0 5	6 8	9 10	20	21 30	31
0 1 1 1 1 1	///	L	///	1 0 0 1 0 1 0 1 1 0	/

The **sync** instruction creates a memory barrier. The set of storage accesses that is ordered by the memory barrier depends on the value of the L field.

L=0 (“heavyweight sync”)

> The memory barrier provides an ordering function for the storage accesses associated with all instructions that are executed by the processor executing the **sync** instruction. The applicable pairs are all pairs a_i,b_j in which b_j is a data access, except that if a_i is the storage access caused by an **icbi** instruction then b_j may be performed with respect to the processor executing the sync instruction before a_i is performed with respect to that processor.

L=1 (“lightweight sync”)

> The memory barrier provides an ordering function for the storage accesses caused by load, store, **dcbz**, **dcbzl** <DEO>, **dcbzep** <E.PD>, and **dcbzlep** <E.PD,DEO> instructions that are executed by the processor executing the **sync** instruction and for which the specified storage location is in storage that is neither Write Through Required nor Caching Inhibited. The applicable pairs are all pairs a_i,b_j of such accesses except those in which a_i is an access caused by a store, **dcbz**, **dcbzl** <DEO>, **dcbzep** <E.PD>, or **dcbzlep** <E.PD,DEO> instruction and b_j is an access caused by a load instruction.
>
> The memory barrier orders accesses described by the applicable pairs above to the local caches of the processor such that a_i is performed in all caches local to the processor prior to any b_j access.

The ordering done by all memory barriers created by the **sync** instruction is cumulative. If L=0, the **sync** instruction has the following additional properties.

- Executing the **sync** instruction ensures that all instructions preceding the **sync** instruction have completed before the **sync** instruction completes, and that no subsequent instructions are initiated until after the **sync** instruction completes.
- The **sync** instruction is execution synchronizing (See Section 4.5.4.5, “Execution Synchronization.”).
- The memory barrier provides the additional ordering function such that if a given instruction that is the result of a store in set B is executed, all applicable storage accesses in set A have been performed with respect to the processor executing the instruction to the extent required by the associated memory coherence properties. The single exception is that any storage access in set A that is caused by an **icbi** instruction executed by the processor executing the **sync** instruction (P1) may not have been performed with respect to P1.

The cumulative properties of the barrier apply to the execution of the given instruction as they would to a load that returned a value that was the result of a store in set B.

- The **sync** instruction provides an ordering function between **tlbivax** and **tlbsync** instructions and related storage accesses as described in the **tlbsync** instruction. (see page 5-321)
- The **sync** instruction also orders store operations and **msgsnd** delivery similar to a memory barrier such that all stores executed prior to a **sync** instruction will have been performed prior to a **msgsnd** executed on the same processor after the **sync** instruction.

The values L=2 and L=3 are reserved. The **sync** instruction may complete before storage accesses associated with instructions preceding the **sync** instruction have been performed. The **sync** instruction may complete before operations caused by **dcbt** and **dcbtst** instructions preceding the **sync** instruction have been performed.

Other registers altered: None

Extended mnemonics for **sync** are shown in this table. For more information, see Section C.10.7, “Sync (sync).

Table 5-15. Extended Mnemonics for sync

Extended Mnemonic	Equivalent To
msync	sync 0,0
sync	sync 0,0
lwsync	sync 1,0

Except in the **sync** instruction description in this section, references to “**sync**” in EREF unless otherwise stated or obvious from context; the appropriate extended mnemonics are used when other L values are intended.

NOTE: Software Considerations

Some older implementations define an HID0[ABE] field that is used to enable the broadcast of the sync operation (along with broadcasting other multiprocessing operations); see the core reference manual for more information.

NOTE: Software Considerations

sync can be used to ensure that all stores into a data structure, caused by store instructions executed in a “critical section” of a program, are performed with respect to another processor before the store that releases the lock is performed with respect to that processor.

The memory barrier created by a **sync** instruction does not order implicit storage accesses or instruction fetches.

(The memory barrier created by a **sync** instruction appears to order instruction fetches for instructions preceding the **sync** instruction with respect to data accesses caused by instructions following the **sync** instruction. However, this ordering is a consequence of the first “additional property” of **sync** with L=0, not a property of the memory barrier.)

NOTE: Software Considerations

The functions provided by **lwsync** (**sync** with L=1) are a strict subset of those provided by **sync** with L=0.

In order to obtain best performance for memory barriers, the programmer should use **lwsync** wherever possible. If **lwsync** is insufficient, then **mbar** or **sync** with L=0 should be used. See also Section 6.4.8.1, “Memory Access Ordering.”

NOTE: Software Considerations

Some older implementations do not provide **lwsync** (**sync** with L=1) and attempting to execute **lwsync** may cause an illegal instruction exception.

td

64	User

td

Trap Doubleword

td TO**,r**A**,r**B

0–5	6–10	11–15	16–20	21–30	31
0 1 1 1 1 1	TO	**r**A	**r**B	0 0 0 1 0 0 0 1 0 0	/

```
a ← (rA))
b ← (rB)
if (a <  b) & $TO_0$ then TRAP
if (a >  b) & $TO_1$ then TRAP
if (a =  b) & $TO_2$ then TRAP
if (a $<_u$ b) & $TO_3$ then TRAP
if (a $>_u$ b) & $TO_4$ then TRAP
```

The contents of **r**A are compared with the contents of **r**B. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered: None

Simplified mnemonics: See Section C.8, “Trap Instructions Simplified Mnemonics.”

tdi

64	User

tdi

Trap Doubleword Immediate

tdi TO,**r**A,SIMM

0 5	6 10	11 15	16 31
0 0 0 0 1 0	TO	**r**A	SIMM

```
a ← EXTS(rA)
b ← EXTS(SIMM)
if (a <  b) & $TO_0$ then TRAP
if (a >  b) & $TO_1$ then TRAP
if (a =  b) & $TO_2$ then TRAP
if (a $<_u$ b) & $TO_3$ then TRAP
if (a $>_u$ b) & $TO_4$ then TRAP
```

The contents of **r**A are compared with the sign-extended value of the SIMM field.

If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered: None

Simplified mnemonics: See Section C.8, “Trap Instructions Simplified Mnemonics.”

tlbilx

E.HV	Supervisor

tlbilx

TLB Invalidate Local Indexed

tlbilx T,**r**A,**r**B

0 … 5	6	7 8	9 10	11 … 15	16 … 20	21 … 30	31
0 1 1 1 1 1	**0**	///	T	**r**A	**r**B	0 0 0 0 0 1 0 0 1 0	/

```
if rA = 0 then a ← 0 else a ← (rA)
EA ← (a + (rB))
for each TLB array do
   for each TLB entry do
      if (entry$_{IPROT}$ = 0) & (MAS5$_{SLPID}$ = entry$_{TLPID}$) then do
         k ← ¬((1 << (2 × (entry$_{SIZE}$-1))) - 1)
         if T = 0 then entry$_V$ ← 0
         if T = 1 & (MAS6$_{SPID}$ = entry$_{TID}$) then entry$_V$ ← 0
         if T = 3 & (MAS6$_{SPID}$ = entry$_{TID}$) & ((EA$_{0:51}$ & k)=(entry$_{EPN}$ & k)) &
                    (MAS5$_{SGS}$ = entry$_{TGS}$) & (MAS6$_{SAS}$ = entry$_{TS}$) then entry$_V$ ← 0
```

tlbilx invalidates TLB entries in the processor which executes the **tlbilx** instruction. TLB entries protected by the IPROT attribute (entry[IPROT]=1) are not invalidated.

If T = 0, all TLB entries for which entry[TLPID] = MAS5[SLPID] are invalidated.

If T = 1, all TLB entries for which entry[TLPID] = MAS5[SLPID] and entry$_{TID}$ = MAS6[SPID] are invalidated.

If T = 3, all TLB entries for which the following are all true are invalidated:

- entry[TLPID] = MAS5[SLPID]
- entry[TID] = MAS6[SPID]
- (entry[EPN]&*m*) = (EA$_{0:51}$&*m*), where *m* is an appropriate mask based on page size
- entry[TS] = MAS6[SAS]
- entry[TGS] = MAS5[SGS]

If T = 2, the instruction form is invalid.

The effects of the invalidation are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation.

Invalidations may occur for other TLB entries on the processor executing the **tlbilx** instruction, but in no case are any TLB entries with the IPROT attribute set made invalid.

The preferred form of **tlbilx** has rA = 0. Some implementations may take an illegal instruction exception if rA $\neq$ 0.

This instruction is guest supervisor privileged. If guest execution of TLB Management instructions is disabled (EPCR[DGTMI]=1), this instruction is hypervisor privileged.

Extended mnemonics for **tlbilx** are shown in this table.

Table 5-16. Extended Mnemonics for tlbilx

Extended Mnemonic	Equivalent To
tlbilxlpid	**tlbilx** 0,0,0
tlbilxpid	**tlbilx** 1,0,0
tlbilxva rA,rB	**tlbilx** 3,rA,rB
tlbilxva rB	**tlbilx** 3,0,rB

NOTE: Software Considerations

tlbilx is the preferred way of performing TLB invalidations, especially for operating systems running as a guest to the hypervisor since the invalidations are partitioned and do not require hypervisor privilege.

The preferred form of **tlbilx** has **r**A = 0. Forms where **r**A ≠ 0 takes an illegal instruction exception on some processors.

Executing **tlbilx** with T = 0 or T = 1 may take many cycles to perform. Software should only issue these operations when a logical partition ID or process ID value is reused or taken out of use.

NOTE: Virtualization

Hypervisor should always ensure that MAS5[SLPID] is set to LPIDR and MAS5[SGS] is set to 1 when dispatching to a guest if guests are allowed to execute **tlbilx**.

Simplified mnemonics: See Section C.10.10, “TLB Invalidate Local Indexed.”

tlbivax

Embedded	Hypervisor

TLB Invalidate Virtual Address Indexed

tlbivax **r**A,**r**B

0–5	6	7–10	11–15	16–20	21–30	31
0 1 1 1 1 1	0	///	**r**A	**r**B	1 1 0 0 0 1 0 0 1 0	/

```
if rA=0 then a ← $^{64}0$ else a ← (rA)
EA ← (a + (rB))
if embedded hypervisor implemented then target_lpid ← $MAS5_{SLPID}$
if $MMUCFG_{MAVN}$ = 0 then
   inval_all ← $EA_{61}$ else inval_all ← 0
   target_array ← $EA_{59:60}$
else
   inval_all ← 0
for each processor do
   for each TLB array
       if ($MMUCFG_{MAVN}$ ≠ 0) | (array = target_array) then do
           for each entry do
               k ← ¬((1 << (2 × ($entry_{SIZE}$-1))) - 1)
               if embedded hypervisor implemented then do
                   if target_lpid = $entry_{TLPID}$ &
                      target_gs = $entry_{TGS}$ then lpid_match ← 1
                                              else lpid_match ← 0
               else lpid_match ← 1
               if ($entry_{IPROT}$ = 0) &
                  (inval_all |
                  ( (($EA_{0:51}$ & k)=($entry_{EPN}$ & k)) &
                    ($MAS6_{SPID}$ = $entry_{TID}$) & ($MAS6_{SAS}$ = $entry_{TS}$) &
                    lpid_match )then $entry_{V}$ ← 0
```

tlbivax invalidates any TLB entry that corresponds to a virtual address and TLB array calculated by this instruction if IPROT is not set; this includes invalidating TLB entries on other devices as well as on the processor executing **tlbivax**. The virtual address is defined by the following:

EA[0-51]	The effective page number (entry[EPN]) to compare
MAS5[SLPID]	LPID value (entry[TLPID]) to compare <E.HV>
MAS5[SGS]	GS value (entry[TGS]) to compare <E.HV>
MAS6[SPID]	PID value (entry[TID]) to compare
MAS6[SAS]	AS value (entry[TS]) to compare
MAS6[SIND]	AS value (entry[IND]) to compare <E.PT>

In addition, for MMU V1, other bits of EA are used to further define the invalidation:

EA[59-60]	TLB array selector 00=TLB0 01=TLB1 10=TLB2 11=TLB3

EA[61] Invalidate all entries in selected TLB array

If EA[61] = 1, then all TLB entries in the TLB array specified by EA[59-60] in all processors in the coherence domain, except for entries protected by the IPROT attribute, are made invalid.

For MMU V1, if EA[61] = 0, then the TLB entries in the TLB array specified by EA[59-60] in all processors in the coherence domain except for entries protected by the IPROT attribute, are subject to invalidation if they meet the matching criteria. The matching criteria is as follows:

- (EA[0-51] & k) = (entry[EPN] & k), where *k* is an appropriate mask based on the pages size of the TLB entry. This selects the page address.
- MAS5[SLPID] = entry[TLPID] and MAS5[SGS] = entry[TGS]. This selects the logical partition and hypervisor/guest address spaces. <E.HV>
- MAS6[SPID] = entry[TID] and MAS6[SAS] = entry[TS]. This selects the process ID and address space.

TLB entries other than the TLB entries specified may also be invalidated. In no case will any TLB entry that is protected with the IPROT attribute be invalidated.

The operation performed by this instruction is ordered by **mbar** (or **sync**) with respect to a subsequent **tlbsync** executed by the processor executing **tlbivax**. Operations caused by **tlbivax** and **tlbsync** are ordered by **mbar** as a set of operations independent of the other sets that **mbar** orders.

The effects of the invalidation are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation. See Section 4.5.4.4, "Context Synchronization."

This instruction is hypervisor privileged.

Other registers altered: None

NOTE: Software Considerations

Some older implementations define an HID0[ABE] field that is used to enable the broadcast of the **tlbivax** operation (along with broadcasting other multiprocessing operations); see the core reference manual for more information.

NOTE: Software Considerations

The preferred form of **tlbivax** contains the entire EA in **r**B and zero in **r**A. Some implementations may take an unimplemented instruction exception if **r**A is nonzero.

NOTE: Software Considerations

System software must take care that TLB entries that map interrupt vectors are not invalidated. Such entries should be protected with the IPROT attribute.

NOTE: Software Considerations

The preferred method of invalidation is to use **tlbilx**. Future versions of the architecture may not include all **tlbivax** capabilities.

NOTE: Virtualization

Hypervisors may wish to emulate **tlbivax** for guests. It is recommended that such emulation be performed by executing the appropriate **tlbilx** instructions on each processor in the logical partition.

tlbre

Embedded	Hypervisor

tlbre

TLB Read Entry

tlbre

0					5	6	7 … 20	21									30	31
0	1	1	1	1	1	0	///	1	1	1	0	1	1	0	0	1	0	/

```
entry ← SelectTLB($MAS0_{TLBSEL}$, $MAS0_{ESEL}$, $MAS2_{EPN}$)
if entry was not found then
    $MAS1_{V}$ ← 0
    $MAS1_{IPROT\ TID\ TS\ TSIZE\ IND}$ ← undefined
    MAS2 ← undefined
    MAS3 ← undefined
    MAS7 ← undefined
    if category E.HV not implemented then MAS8 ← undefined
else
    $rpn_{0:51}$ ← $entry_{RPN}$
    if TLB array supports Next Victim then
        $MAS0_{NV}$ ← hint
    else
        $MAS0_{NV}$ ← undefined
    $MAS1_{V\ IPROT\ TID\ TS\ TSIZE}$ ← $entry_{V\ IPROT\ TID\ TS\ SIZE}$
    $MAS3_{UX\ SX\ UW\ SW\ UR\ SR}$ ← $entry_{UX\ SX\ UW\ SW\ UR\ SR}$
    $MAS2_{EPN\ VLE\ W\ I\ M\ G\ E\ ACM}$ ← $entry_{EPN\ VLE\ W\ I\ M\ G\ E\ ACM}$
    $MAS3_{RPNL}$ ← $rpn_{32:51}$
    $MAS3_{U0:U3}$ ← $entry_{U0:U3}$
    $MAS7_{RPNU}$ ← $rpn_{0:31}$
    if category E.HV implemented then $MAS8_{TGS\ VF\ TLPID}$ ← $entry_{TGS\ VF\ TLPID}$
```

The contents of the TLB entry specified by MAS0[TLBSEL], MAS0[ESEL], and MAS2[EPN] are read and placed into the MAS registers.

If the TLB array supports MAS0[NV], then an implementation defined value, hint, specifying the index for the next entry to be replaced is loaded into MAS0[NV]; otherwise MAS0[NV] is set to an undefined value.

If the specified entry does not exist, MAS1[V] is set to 0 and all other MAS register fields are undefined.

This instruction is hypervisor privileged.

Other Registers Altered: MAS0 MAS1 MAS2 MAS3 MAS7 MAS8<E.HV>

NOTE: Virtualization

Hypervisors should emulate **tlbre** for guests.

tlbsx tlbsx

Embedded	Hypervisor

TLB Search Indexed

tlbsx **r**A,**r**B

0 … 5	6	… 10	11 … 15	16 … 20	21 … 30	31
0 1 1 1 1 1	0	///	**r**A	**r**B	1 1 1 0 0 1 0 0 1 0	/

```
if rA = 0 then a ← 0 else a ← (rA)
EA ← (a + (rB))
pid ← $MAS6_{SPID}$
if embedded hypervisor implemented then
    lpid ← $MAS5_{SLPID}$
    gs ← $MAS5_{SGS}$
else
    lpid ← 0
    gs ← 0
as ← $MAS6_{SAS}$
va ← gs || lpid || as || pid || EA
if Valid_matching_entry_exists(va) then
    entry ← matching entry found
    array ← TLB array number where TLB entry found
    index ← index into TLB array of TLB entry found
    if TLB array supports Next Victim then
        hint ← hardware hint for Next Victim
    else
        hint ← undefined
    $rpn_{0:51}$ ← $entry_{RPN}$
    $MAS0_{TLBSEL}$ ← array
    $MAS0_{ESEL}$ ← index
    $MAS0_{NV}$ ← hint
    $MAS1_{V}$ ← 1
    $MAS1_{IPROT\ TID\ TS\ TSIZE}$ ← $entry_{IPROT\ TID\ TS\ SIZE}$

    $MAS2_{EPN\ W\ I\ M\ G\ E\ ACM}$ ← $entry_{EPN\ W\ I\ M\ G\ E\ ACM}$
    if category VLE implemented then $MAS2_{VLE}$ ← $entry_{VLE}$
    $MAS3_{RPNL}$ ← $rpn_{32:51}$
$MAS3_{U0:U3}$ ← $entry_{U0:U3}$
    $MAS7_{RPNU}$ ← $rpn_{0:31}$
    if embedded hypervisor implemented then $MAS8_{TGS\ VF\ TLPID}$ ← $entry_{TGS\ VF\ TLPID}$
else
    $MAS0_{TLBSEL}$ ← $MAS4_{TLBSELD}$
    $MAS0_{ESEL}$ ← hint
    $MAS0_{NV}$ ← hint
    $MAS1_{V\ IPROT}$ ← 0
    $MAS1_{TID\ TS}$ ← $MAS6_{SPID\ SAS}$
    $MAS1_{TSIZE}$ ← $MAS4_{TSIZED}$
    $MAS2_{W\ I\ M\ G\ E\ ACM}$ ← $MAS4_{WD\ ID\ MD\ GD\ ED\ ACMD}$
    $MAS3_{RPNL}$ ← 0
    $MAS7_{RPNU}$ ← 0
```

If any valid TLB array contains an entry corresponding to the virtual address formed by MAS6[SAS,SPID], MAS5[SGS,SLPID] <E.HV>, and EA, that entry as well as the index and array are read into the MAS registers. If no valid matching translation exists, MAS1[V] is set to 0 and the MAS registers are loaded with defaults to facilitate a TLB replacement.

If the TLB array supports MAS0[NV], an implementation defined value, hint, specifying the index for the next entry to be replaced is loaded into MAS0[NV] regardless of whether a match occurs; otherwise MAS0[NV] is set to an undefined value. It is also loaded into MAS0[ESEL] if no match occurs.

This instruction is hypervisor privileged.

Special registers altered: MAS0 MAS1 MAS2 MAS3 MAS7 MAS8<E.HV>

NOTE: Software Considerations

The preferred form of **tlbsx** contains the entire EA in **r**B and zero in **r**A. Some implementations may take an unimplemented instruction exception if **r**A is nonzero.

tlbsync tlbsync

Embedded	Hypervisor

TLB Synchronize

tlbsync

0 5	6	7 20	21 30	31
0 1 1 1 1 1	0	///	1 0 0 0 1 1 0 1 1 0	/

tlbsync provides an ordering function for the effects of all **tlbivax** instructions executed by the processor executing **tlbsync**, with respect to the memory barrier created by a subsequent **msync** instruction executed by the same processor. Executing **tlbsync** ensures that all of the following occur.

- All TLB invalidations caused by **tlbivax** instructions preceding the **tlbsync** will have completed on any other processor before any memory accesses associated with data accesses caused by instructions following the **sync** instruction are performed with respect to that processor.
- All memory accesses by other processors for which the address was translated using the translations being invalidated, will have been performed with respect to the processor executing the **sync** instruction, to the extent required by the associated memory-coherence required attributes, before the **mbar** or **sync** instruction's memory barrier is created.

The operation performed by this instruction is ordered by the **mbar** and **sync** instructions with respect to preceding **tlbivax** instructions executed by the processor executing the **tlbsync** instruction. The operations caused by **tlbivax** and **tlbsync** are ordered by **mbar** as a set of operations that is independent of the other sets that **mbar** orders.

The **tlbsync** instruction may complete before operations caused by **tlbivax** instructions preceding the **tlbsync** instruction have been performed.

Only one **tlbsync** may be active at a time. A second **tlbsync** executed (from any processor in the coherence domain) before the first one is completed, can cause processors to hang or a machine check interrupt to occur.

This instruction is hypervisor privileged.

Other registers altered: None

NOTE: Software Considerations

Software must ensure that only one **tlbsync** operation is active at a given time. Software should make sure the **tlbsync** and its associated synchronization are contained within a mutual exclusion lock that all processors must acquire before executing **tlbsync**.

NOTE: Software Considerations

Example code sequence for performing **tlbivax** and **tlbsync**:

```
// lock already acquired..
    tlbivax
    sync
    tlbsync
```

```
        sync
        isync
    // release lock..
```

tlbwe

Embedded	Supervisor

tlbwe

TLB Write Entry

tlbwe

0 5	6	7 20	21 30	31
0 1 1 1 1 1	0	///	1 1 1 1 0 1 0 0 1 0	/

```
if $MSR_{PR}$ = 1 then
   privilege operation exception; exit
entry ← SelectTLB($MAS0_{TLBSEL}$, $MAS0_{ESEL}$, $MAS0_{HES}$, $MAS2_{EPN}$)
hint ← $MAS0_{NV}$
$entry_{V\ IPROT\ TID\ TS\ SIZE}$ ← $MAS1_{V\ IPROT\ TID\ TS\ TSIZE}$
$entry_{U0:U3\ UX\ SX\ UW\ SW\ UR\ SR}$ ← $MAS3_{U0:U3\ UX\ SX\ UW\ SW\ UR\ SR}$
$entry_{EPN\ W\ I\ M\ G\ E\ ACM}$ ← $MAS2_{EPN\ W\ I\ M\ G\ E\ ACM}$
$entry_{RPN}$ ← rpn
if category E.HV implemented then $entry_{TGS\ VF\ TLPID}$ ← $MAS8_{TGS\ VF\ TLPID}$
if category VLE implemented then $entry_{VLE}$ ← $MAS2_{VLE}$
```

The contents of the MAS registers are written to the TLB entry specified by MAS0[TLBSEL], MAS0[ESEL], and MAS2[EPN].

MAS0[NV] provides a suggestion to hardware of where the next hardware hint for replacement should be given when the next data or instruction TLB error interrupt, **tlbsx**, or **tlbre** instruction occurs.

If the specified entry does not exist, the results are undefined.

A context synchronizing instruction is required after a **tlbwe** instruction to ensure any subsequent instructions that will use the updated TLB values execute in the new context.

This instruction is hypervisor privileged. This instruction is hypervisor privileged if EPCR[DGMTI] = 1.

Special registers altered: None

NOTE: Software Considerations

The effects of the TLB update are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation. See Section 4.5.4.4, “Context Synchronization.”

NOTE: Virtualization

tlbwe must be emulated by the hypervisor if the guest is not allowed to write TLB entries.

tw

Base	User

tw

Trap Word

tw TO,**r**A,**r**B

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	TO	**r**A	**r**B	0 0 0 0 0 0 0 1 0 0	/

```
a ← EXTS($(rA)_{32:63}$)
b ← EXTS($(rB)_{32:63}$)
if (a <  b) & $TO_0$ then TRAP
if (a >  b) & $TO_1$ then TRAP
if (a =  b) & $TO_2$ then TRAP
if (a $<_u$ b) & $TO_3$ then TRAP
if (a $>_u$ b) & $TO_4$ then TRAP
```

The contents of **r**A[32–63] are compared with the contents of **r**B[32–63]. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered: None

Simplified mnemonics: See Section C.8, “Trap Instructions Simplified Mnemonics.”

twi

Base	User

twi

Trap Word Immediate

twi TO,**r**A,SIMM

0 5	6 10	11 15	16 31
0 0 0 0 1 1	TO	**r**A	SIMM

```
a ← EXTS($(rA)_{32:63}$)
b ← EXTS(SIMM)
if (a <  b) & $TO_0$ then TRAP
if (a >  b) & $TO_1$ then TRAP
if (a =  b) & $TO_2$ then TRAP
if (a $<_u$ b) & $TO_3$ then TRAP
if (a $>_u$ b) & $TO_4$ then TRAP
```

The contents of **r**A[32–63] are compared with the sign-extended value of the SIMM field.

If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered: None

Simplified mnemonics: See Section C.8, “Trap Instructions Simplified Mnemonics.”

wait wait

Wait	User

Wait for Interrupt

wait WC,WH

0 5	6 8	9 10	11	12 20	21 30	31
0 1 1 1 1 1	///	WC	WH	///	0 0 0 0 1 1 1 1 1 0	/

The **wait** instruction allows instruction fetching and execution to be suspended under certain conditions, depending on the value of the WC field. A **wait** instruction without any operands is treated as a **wait** instruction with WC = 0 and WH = 0.

The defined values of the WC field are as follows:

0b00 Resume instruction fetching and execution when an interrupt occurs.

0b01 Resume instruction fetching and execution when an interrupt occurs or when a reservation does not exist.

0b10 Reserved.

0b11 Reserved.

If WC = 0, or if WC = 1 and a reservation exists for the processor then the following occurs:

- Instruction fetching and execution is suspended.
- Once the wait instruction has completed, the NIA will point to the next sequential instruction.

Execution and fetching are suspended until one of the following occurs:

- An asynchronous interrupt occurs or a debug postcompletion (e.g. ICMP) event and subsequent interrupt occurs
- If WC=1 and the reservation is lost
- If NIA is written when the processor is halted

An implementation may support other conditions that cause instruction execution and fetching to resume. Software should always be aware that such conditions could exist.

Executing **wait** is a hint to the processor that no further synchronous processor activity will occur until the appropriate condition occurs. The processor may use this to reduce power consumption.

If WH = 1, this is a hint to the processor that the wait is likely to be of a longer duration and the processor may use this to enter a lower power state more quickly.

Executing **wait** with reserved values for WC either causes **wait** with WC = 0 to be executed, or produces a no-op.

Not all Freescale processors implement **wait** with WC = 1. Those processors will ignore the WC field and always implement it as WC = 0.

Other registers altered: None

Simplified mnemonics:

- **wait** equivalent to: **wait** 0,0
- **waitrsv** equivalent to: **wait** 1,0

NOTE: Software Considerations

The **wait** instruction may be useful as the completion of processing for a cooperative thread or the primary instruction of an idle process. However, overall system performance may be better served if the wait instruction is used by applications only for idle times that are expected to be short.

Note that **wait** updates the NIA so that an interrupt that awakens a **wait** instruction will naturally return to the instruction after the **wait**.

NOTE: Software Considerations

If debugger software using the ICMP debug event to single-step instruction execution wishes to have the **wait** instruction wait only for asynchronous interrupts (as it would do in the absence of debugging events), software should examine each instruction that is to be executed next in the debug instruction complete handler, and if it was a **wait** instruction, turn off ICMP, and enable events for interrupts taken or interrupt returns depending on what behavior is desired. When the interrupt taken or interrupt return events produce debug interrupts, software can then turn ICMP back on.

NOTE: Software Considerations

wait with WC = 1, can be used by multiprocessing locking code that is waiting for a lock to reduce power consumption.

wrtee

Embedded	Supervisor

wrtee

Write MSR External Enable

wrtee **r**S

0 5	6 10	11 20	21 30	31
0 1 1 1 1 1	**r**S	///	0 0 1 0 0 0 0 0 1 1	/

```
$MSR_{EE} \leftarrow (rS)_{48}$
```

rS[48] is placed into MSR[EE].

In addition, changes to MSR[EE] are effective as soon as the instruction completes. Thus if MSR[EE]=0 and an external interrupt is pending, executing a **wrtee** that sets MSR[EE] causes the external interrupt to be taken before the next instruction is executed, if no higher priority exception exists.

This instruction is guest supervisor privileged.

Other registers altered: MSR

NOTE: Software Considerations

wrtee and **wrteei** are used to update of MSR[EE] without affecting other MSR bits. Typical usage is as follows:

```
mfmsr   Rn  #save EE in GPR(Rn)48
wrteei  0   #turn off EE
:   :       :
:   :       #code with EE disabled
:   :       :
wrtee   Rn  #restore EE without altering other MSR bits that may have changed
```

wrteei

Embedded	Supervisor

Write MSR External Enable Immediate

wrteei **E**

0 5	6 15	16	17 20	21 30	31
0 1 1 1 1 1	///	E	///	0 0 1 0 1 0 0 0 1 1	/

$\mathtt{MSR}_{\mathtt{EE}} \leftarrow \mathtt{E}$

E is placed into MSR[EE].

In addition, changes to MSR[EE] are effective as soon as the instruction completes. Thus if MSR[EE]=0 and an external interrupt is pending, executing a **wrteei** that sets MSR[EE] causes the external interrupt to be taken before the next instruction is executed, if no higher priority exception exists.

This instruction is guest supervisor privileged.

Other registers altered: MSR

NOTE: Software Considerations

wrtee and **wrteei** are used to update of MSR[EE] without affecting other MSR bits. Typical usage is as follows:

```
mfmsr   Rn  #save EE in GPR(Rn)48
wrteei  0   #turn off EE
:   :       :
:   :       #code with EE disabled
:   :       :
wrtee   Rn  #restore EE without altering other MSR bits that may have changed
```

xor xor

Base	User

XOR

xor	**r**A,**r**S,**r**B	(Rc=0)
xor.	**r**A,**r**S,**r**B	(Rc=1)

0 — 5	6 — 10	11 — 15	16 — 20	21 — 30	31
0 1 1 1 1 1	**r**S	**r**A	**r**B	0 1 0 0 1 1 1 1 0 0	Rc

```
result ← (rS) ⊕ (rB)
if Rc=1 then do
   LT  ← $result_{m:63}$ < 0
   GT  ← $result_{m:63}$ > 0
   EQ  ← $result_{m:63}$ = 0
   CR0 ← LT || GT || EQ || SO
rA ← result
```

The contents of **r**S are XORed with the contents of **r**B.

The result is placed into **r**A.

Other registers altered: CR0 (if Rc=1)

xori

Base	User

xori

XOR Immediate

xori **r**A,**r**S,UIMM

0 4 5	6 10	11 15	16 31
0 1 1 0 1 0	**r**S	**r**A	UIMM

```
rA ← (rS) ⊕ ($^{48}0$ || UIMM)
```

The contents of **r**S are XORed with $^{48}0$ || UIMM.

The result is placed into **r**A.

Other registers altered: None

xoris

Base	User

XOR Immediate Shifted

xoris **r**A,**r**S,UIMM

0 4 5	6 10	11 15	16 31
0 1 1 0 1 1	**r**S	**r**A	UIMM

```
rA ← (rS) ⊕ ($^{32}$0 || UIMM || $^{16}$0)
```

The contents of **r**S are XORed with ^{32}0 || UIMM || ^{16}0.

The result is placed into **r**A.

Other registers altered: None

Chapter 6
Cache and MMU Architecture

This chapter describes the cache and MMU architecture for Freescale embedded devices. It is organized into the following sections:

- Section 6.1, "Cache and MMU Architecture Overview"
- Section 6.2, "Cache Model"
- Section 6.3, "Special Cache Management Functionality"
- Section 6.4, "Memory and Cache Coherency"
- Section 6.5, "MMU Architecture"

It focuses on how Freescale devices manage cache implementations and memory translation and protection. Note, however, that cache and MMU implementations vary from device to device, particularly with respect to the implementation of the caches and TLBs. For more information, refer to the core reference manuals.

This chapter also describes additional functionality that is defined by the architecture and may be implemented on Freescale cores, such as the following:

- Cache Line Locking <E.CL>
- Cache Way Partitioning <CWP>
- External PID load and store <E.PD>
- Decorated storage <DS>
- Direct Cache Flushing <DCF>
- Data cache block extended operations <DEO>

This functionality may be implemented independently of each other; however, an implementation is likely to include more than one. Consult the core documentation.

NOTE: Software Considerations

This manual describes some architected features in a very general way that does not include some details that are important to the programmer. There are also small differences in how some features are defined and how they are implemented. For implementation-specific details, see the user documentation.

Throughout this chapter, references to load instructions include cache management and other instructions that are stated in the instruction descriptions to be treated as a load, and references to store instructions include the cache management and other instructions that are treated as a store.

6.1 Cache and MMU Architecture Overview

The architecture cache and memory models support a wide variety of implementations. To provide such flexibility, some features are defined very generally, leaving specific details up to the implementation.

To ensure consistency among Freescale devices, EIS defines more specific implementation standards for cache and memory models. However, these standards still leave some details up to individual implementations. Programmers should still consult the core reference manual for additional implementation specific details.

The architectural definition of the cache model consists primarily of a framework that assumes weakly ordered memory and defines instructions and register resources that support Harvard or unified cache designs in configurations that may include both types, typically a Harvard-style L1 cache structure often with a unified L2 cache.

For example, the architecture provides separate instructions for invalidating instruction and data caches, and likewise separate cache touch instructions that can be used to hint that an instruction or data cache block is likely to be used soon. See Section 6.2.3, “Cache Control Instructions.”

EIS provides registers for configuring and managing L1 and L2 caches, which are described in detail in Section 3.10, “L1 Cache Registers,” and Section 3.11, “L2 Cache Registers.”

Many aspects of the cache implementation are affected by the MMU model, in particular the memory coherency attributes (described in Section 6.4.1, “Memory/Cache Access Attributes (WIMGE Bits)”).

The MMU model for embedded devices assumes that translation is always enabled (there is no real mode) and that memory is configured into pages, for which translation and permission attributes are configured by software through TLBs using instructions and registers defined for that purpose by the architecture.

The architecture supports demand-paged virtual memory as well as a variety of other management schemes that depend on precise control of effective-to-real address translation and flexible memory protection. Address translation misses and protection faults cause precise exceptions. Sufficient information is available to correct the fault and restart the faulting instruction.

Each program on a 64-bit or 32-bit implementation can access 2^{64} or 2^{32} bytes of effective address (EA) space, subject to limitations imposed by the operating system. Typically, each program’s EA space is a subset of a larger virtual address (VA) space managed by the operating system. The architecture also provides support for addressing real address (RA) space between 32 and 64 bits. The size of the real address space is independent of whether the processor is a 32-bit or 64-bit implementation. Many 32-bit implementations provide access to more than 32 bits of real address space. Consult the user documentation.

<Embedded.Hypervisor>:
Implementations that provide category E.HV also provide logical addresses (LA). The LA is translated to a physical address by hypervisor software when a TLB entry is written in guest state. Logical addresses are what the guest OS views as real addresses, but are translated to true real addresses.

Each effective address is translated to a real address before being used to access real memory or an I/O device. Hardware does this by using the address translation mechanism described in Section 6.5.4, "Address Translation." The operating system (and hypervisor <E.HV>) manage the physically addressed resources of the system by setting up the tables used by the address translation mechanism.

The architecture divides the effective address space into pages. The page represents the granularity of effective address translation, permission control, and memory/cache attributes. Multiple page sizes may be simultaneously supported. For an effective-to-real address translation to exist, a valid entry for the page containing the effective address must be in a translation lookaside buffer (TLB). Addresses for which no TLB entry exists cause TLB miss exceptions (instruction or data TLB error interrupts).

NOTE: Software Considerations

> Implementations may support only a subset of the available page sizes or may have TLB array that only support a fixed size page. See the core reference manual.

6.2 Cache Model

The cache model defines resources for L1 and L2 caches, which are described as follows:

- L1 caches may separate instruction and data caches into two separate structures (commonly known as Harvard architecture), or they may provide a unified cache combining instructions and data. Caches are physically tagged.
  Some L1 caches may be implemented as write-through in conjunction with a secondary cache. In this case, modified data exists only in the secondary cache.
- Secondary (L2) caches are generally organized as a unified cache, that is, combining instructions and data although they may behave as a Harvard cache by tracking which cache lines are allocated by instruction or data references. Caches are physically tagged. Secondary caches are required to support the coherency mechanisms of the core and the primary caches.
  Secondary caches are defined to be strictly managed by the processor core and are implemented as a direct part of the core programming model. These may include, but are not limited to, backside and inline caches.
  An integrated device may have other caches or cache-like objects that may generate processor core accesses to the memory subsystem. Such caches are not considered part of the core cache programming model and are architected as system-level blocks. These may include, but are not limited to, front lookaside caches.
  Secondary caches may have many different features and capabilities. The architecture does not explicitly state which features are required, but provides a register programming model that must be followed if certain features are implemented. Thus, an implementation may provide additional features and controls beyond what is described here.
  The L2 cache programming model defines a set of SPRs for configuration and control, described in Section 3.11, "L2 Cache Registers." Note that some Freescale cores use device control registers, DCRs, for this purpose. Section 3.18, "Device Control Registers (DCRs)," provides a general description of these registers, which the architecture defines only as a general resource. Specific information is provided in the core documentation.

6.2.1 Storage Attributes and Coherency

L1 data caches support storage attributes defined with the following advisory:

- The primary data cache may be implemented not to snoop (that is, not coherent with transactions outside the processor). System software is then responsible to maintain coherency. Thus the setting of the M attribute is meaningless. The preferred implementation provides snooping for primary data caches.

L1 instruction caches must support the architecture-define storage attributes with the following advisory:

- The guarded attribute should be ignored for instruction fetch accesses. To prevent speculative fetch accesses to guarded memory, software should mark those pages as no-execute.
- The cache may be implemented not to snoop (that is, not coherent with transactions outside the processor). System software is then responsible to maintain coherency. The preferred implementation does not provide snooping for primary instruction caches.

6.2.2 Cache Identifiers and CT Values

Cache identifiers and CT values are used to identify caches to perform cache operations. CT values are the values specified in the CT or TH instruction fields which are used to identify a particular cache in the cache hierarchy. Cache identifiers are used by external devices to identify a particular cache in the SoC which may or may not be a part of a core’s cache hierarchy. Logically, cache identifiers and CT values do not share the same exact namespace. CT values are defined here by EIS. CT values that identify a particular cache are not necessarily the same as the cache identifier for the same cache.

Caches in a system may be specific targets of “cache stashing.” Cache stashing is an operation initiated by another device in the system, specifying a hint that specified addresses should be prefetched into a target cache. The target cache for such an operation is called a “cache identifier.” Each cache in the system that responds to such requests has a cache identifier set by system software or predefined by hardware. For the primary data cache of a processor, the identifier is defined in L1CSR2[DCSTASHID]. For the secondary data cache of the processor (the backside or L2 cache), the identifier is defined in L2CSR1[L2STASHID]. A value of 0 set for a cache identifier indicates that cache stashing operations will not be accepted or performed by the cache.

Instructions having a CT (Cache Target) or TH field for specifying a specific cache hierarchy such as **dcbt**, **dcbtst**, **dcbtls** <E.CL>, **dcbtstls** <E.CL>, **dcblc** <E.CL>, **icbtls** <E.CL>, **icblc** <E.CL>, and **icbt** use the values in the following table to identify target caches:

Table 6-1. CT Values and Target Cache

CT Value	Target Cache
0	Primary data cache (or unified primary cache) of the processor executing the instruction. This value may not be used as a cache identifier in an external cache operation (stash) as it is implicit.
1	The I/O cache (platform cache). Defined by the SoC architecture.
2	Secondary (L2) data cache of the processor executing the instruction. This value may not be used as a cache identifier in an external cache operation as it is implicit.
3,5,7	Available for use as a cache identifier in an external cache operation.
4,6	Reserved. Not available for use as a cache identifier in an external cache operation.
8-31	Reserved. Not available for use as a cache identifier in an external cache operation. Defined by PowerISA for streaming prefetch control.

NOTE: Software Considerations

Software should avoid assigning cache identifiers 1, 2, 4, and 6 as such values already have predefined targets or may be assigned by the architecture in the future.

6.2.3 Cache Control Instructions

The architecture defines instructions for controlling both the instruction and data caches (when they exist).

- Data Cache Block Touch (**dcbt**)
- Data Cache Block Touch for Store (**dcbtst**)
- Data Cache Block Zero (**dcbz**, **dcbzl** <DEO>, **dcbzep** <E.PD>, **dcbzlep** <E.PD,DEO>)
- Data Cache Block Store (**dcbst**, **dcbstep** <E.PD>)
- Data Cache Block Flush (**dcbf**)
- Data Cache Block Allocate (**dcba**, **dcbal** <DEO>)
- Data Cache Block Invalidate (**dcbi**)
- Instruction Cache Block Invalidate (**icbi**, **icbiep** <E.PD>)
- Instruction Cache Block Touch (**icbt**)

These instructions are described in Section 4.6.1.17.2, "User-Level Cache Management Instructions," and Section 4.6.2.2.1, "Supervisor-Level Cache Management Instructions."

Cache locking adds the following instructions <E.CL>:

- Data Cache Block Lock Clear (**dcblc**)
- Data Cache Block Touch and Lock Set (**dcbtls**)
- Data Cache Block Touch for Store and Lock Set (**dcbtstls**)
- Data Cache Block Lock Query (**dcblq.**)
- Instruction Cache Block Lock Clear (**icblc**)
- Instruction Cache Block Touch and Lock Set (**icbtls**)
- Instruction Cache Block Lock Query (**icblq.**)

Cache locking is described in Section 6.3.1, "Cache Line Locking <E.CL>."

6.3 Special Cache Management Functionality

The following sections describe special cache management features. Consult the user documentation to determine whether these features are implemented.

6.3.1 Cache Line Locking <E.CL>

The cache line locking facility defines instructions and methods for locking cache lines for frequently used instructions and data. Cache locking allows software to instruct the cache to keep latency sensitive data readily available for fast access. This is accomplished by marking individual cache lines (or blocks) as locked.

A locked line differs from a normal line in that lines that are locked in the cache do not participate in the normal replacement policy when a line must be victimized for replacement.

6.3.1.1 Cache Line Lock Set, Query, and Clear Instructions

This section describes the cache locking instructions, which are summarized in Section 4.6.1.17.3, "Cache Locking Instructions <E.CL>" and Chapter 5, "Instruction Set." Lines are locked into the cache by software using a series of touch and lock set instructions.

MSR[UCLE], the user-mode cache lock enable bit, determines whether cache line locking instructions can be executed in user mode. If UCLE = 0, any cache lock instruction executed in user-mode causes a cache-locking exception and data storage interrupt and sets either ESR[DLK] or ESR[ILK]. This allows the operating system to manage and track the locking/unlocking of cache lines by user-mode tasks. See Section 6.3.1.2.3, "Mode Restricted Cache Locking Exceptions."

If UCLE = 1, cache-locking instructions can be executed in user-mode and they do not take a DSI for cache-locking (although they may still take a data storage interrupt for access violations).

<Embedded.Hypervisor>:
MSRP[UCLEP], the MSR[UCLE] protect bit, determines whether the guest operating system can execute cache locking instruction or change the value of MSR[UCLE]. When MSRP[UCLEP] is set, an attempt to execute a cache locking instruction while in guest supervisor state causes an embedded hypervisor privilege exception. MSR[UCLEP] does not affect whether unprivileged software can execute cache locking instructions which is still controlled by MSR[UCLE].

The following instructions are provided to lock data items into the data and instruction cache:

- **dcbtls**—Data cache block touch and lock set
- **dcbtstls**—Data cache block touch for store and lock set
- **icbtls**—Instruction cache block touch and lock set

The **r**A and **r**B operands to these instructions form an effective address identifying the line to be locked. The CT field indicates which cache in the cache hierarchy should be targeted. These instructions are similar to the **dcbt**, **dcbtst**, and **icbt** instructions, but locking instructions cannot execute speculatively and may cause additional exceptions. For unified caches, both the instruction lock set and the data lock set target the same cache.

Similarly, lines are unlocked from the cache by software using a series of lock clear instructions. The following instructions are provided to clear locks in the instruction and data cache:

- **dcblc**—Data cache block lock clear.
- **icblc**—Instruction cache block lock clear.

The **r**A and **r**B operands to these instructions form an effective address identifying the line to be unlocked. The CT field indicates which cache in the cache hierarchy should be targeted.

The status of whether a line is locked or not can be queried on some processors using a series of lock query instructions. The following instructions are provided to determine whether a line is locked in the instruction and data cache:

- **dcblq.**—Data cache block lock query.

- **icblq.**—Instruction cache block lock query.

The **r**A and **r**B operands to these instructions form an effective address identifying the line to be queried. The CT field indicates which cache in the cache hierarchy should be targeted. Processors that implement lock query instructions do not post status of locking operations in L2CSR0 or L2CSR1.

Additionally, software may clear all the locks in the cache. For the primary cache, this is accomplished by setting the lock flash clear bits. See Section 3.10.1, "L1 Cache Control and Status Register 0 (L1CSR0)," and Section 3.10.2, "L1 Cache Control and Status Register 1 (L1CSR1)."

Cache lines may also be implicitly unlocked anytime the data in the line is invalidated. The following ways illustrate how such an invalidation may occur:

- A locked line is invalidated by an **dcbi**, **dcbf**, **icbi**, or **icbiep** <E.PD> instruction that targets the line.
- A snoop hit on a locked line can require the line to be invalidated if the data has been modified external to the processor or if another processor explicitly invalidated the line.
- The entire cache containing the locked line is invalidated.
- The cores coherency implementation requires that the line be invalidated.

An implementation is not required to unlock lines if data is invalidated in the cache. Although the data may be invalidated (and thus not be present in the cache), it is still possible for the cache to keep the lock associated with that cache line present and when the next access occurs to fill the line from the memory subsystem. If the implementation does not clear locks when the associated line is invalidated, the method of locking is said to be *persistent*.

6.3.1.2 Error Conditions

Setting locks in the cache can fail for a variety of reasons. An address specified with a lock set instruction that does not have the proper permission will cause a data storage interrupt. Cache locking addresses are always translated as data references, therefore **icbtls** instructions that fail to translate or fail permissions cause DTLB and DSI exceptions respectively. Additionally, cache locking and clearing operations can fail due to restricted user mode (or guest supervisor mode <E.HV>) access. See Section 6.3.1.2.3, "Mode Restricted Cache Locking Exceptions."

6.3.1.2.1 Overlocking

If no exceptions occur for the execution of an **dcbtls**, **dcbtstls**, or **icbtls** instruction, an attempt is made to lock the corresponding cache line. If all of the available ways are locked in the given cache set, the requested line is not locked. This is considered an overlocking situation, and if the lock was targeted for the primary cache (CT = 0), L1CSR0[DCLO] (or L1CSR1[ICLO] if **icbtls**) is set appropriately. If the lock targeted the secondary cache (CT = 2), L2CSR[L2LO] is set.

A processor may optionally allow victimizing a locked line in an overlocking situation. If L1CSR0[DCLOA] (L1CSR0[ICLOA] for the primary instruction cache, L2CSR[L2CLOA] for the secondary cache) is set an overlocking condition will result in the replacement of an existing locked line with the requested line. Selection of the line to replace in an overlocking situation is implementation dependent. The overlocking condition is still said to exist and is appropriately reflected in the status bits for lock overflow.

An attempt to lock a line which is already established and valid does not cause an overlocking condition.

A non–lock setting cache line fill or line replacement request to a cache that has all ways locked for a given set, does not cause a lock to be cleared.

6.3.1.2.2 Unable to Lock Conditions

If no exceptions occur and no overlocking condition exists, an attempt to set a lock can fail if any of the following is true:

- The target address is marked cache-inhibited, or the storage attributes of the address uses a coherency protocol that does not support locking.
- The target cache is disabled or not present.
- The CT field of the instructions contains a value not supported by the implementation.
- Any other implementation-specific error condition.

If an unable-to-lock condition occurs, the lock set instruction is treated as a no-op. If the lock targets the data cache (**dcbtls**, **dcbtstls**) in processors that do not implement lock query **(icblq.** and **dcblq.**) instructions, L1CSR0[DCUL] is set to indicate the unable-to-lock condition. If the lock is targeted for the instruction cache (**icbtls**) in processors that do not implement lock query instructions, L1CSR1[ICUL] is set to indicate the unable-to-lock condition. L1CSR0[DCUL] or L1CSR1[ICUL] is set regardless of the CT value in the lock setting instruction.

Processors that implement lock query instruction (**dcblq.** and **icblq.**) do not report unable-to-lock conditions. Instead software should query the lock status after performing the lock set operation.

6.3.1.2.3 Mode Restricted Cache Locking Exceptions

Cache locking can be restricted to supervisor mode only access. Setting MSR[UCLE] allows cache locking operations to be performed in user mode. If MSR[UCLE] = 0 and MSR[PR] = 1 and execution of a cache lock or cache lock clear instruction occurs, a cache locking exception occurs, and the processor suppresses execution of the instruction causing the exception. If the exception is the highest priority exception, a data storage interrupt is taken and ESR is set as follows:

- ESR[DLK] is set if the instruction was a **dcbtls**, **dcbtstls**, **dcblq.**, or a **dcblc**.
- ESR[ILK] is set if the instruction was a **icbtls**, **icblq.**, or a **icblc**.
- All other ESR bits are cleared.

<Embedded.Hypervisor>:
Cache locking can be restricted to hypervisor mode only access. Setting MSRP[UCLEP] prevents guest supervisor mode from executing cache locking instructions. If MSRP[UCLEP] = 1 and MSR[PR] = 0 and execution of a cache lock or cache lock clear instruction occurs, an embedded hypervisor privilege exception occurs, and the processor suppresses execution of the instruction causing the exception. If the exception is the highest priority exception, an embedded hypervisor privilege interrupt is taken. The setting of MSRP[UCLEP] has no effect on user mode execution of cache locking instructions.

NOTE: Virtualization

> Since MSRP[UCLEP] has no effect on user mode execution of cache locking instructions, the hypervisor should ensure that MSR[UCLE] is not set in the guest to prevent user mode from executing cache locking instructions.

6.3.2 Direct Cache Flushing <DCF>

To assist in software flush of the L1 cache, the direct cache flushing allows the programmer to flush and/or invalidate the cache by specifying the cache set and cache way. Without such a feature, the programmer must do one of the following:

- Know the virtual addresses of the lines that need to be flushed and issue **dcbst** or **dcbf** instructions to those addresses.
- Flush the entire cache by causing all the lines to be replaced. This requires a virtual address range that is mapped as a contiguous physical address range, that the programmer knows and can manipulate the replacement policy of the cache, and the size and organization of the cache.

With direct cache flushing, the program needs only to specify the way and set of the cache to flush.

The direct cache flushing available bit, L1CFG0[CFISWA], is set for implementations that support this functionality.

The direct cache flush facility defines the L1 flush and invalidate control register 0 (L1FINV0), described in Section 3.10.8, “L1 Flush and Invalidate Control Register 0 (L1FINV0) <DCF>.” To address a specific physical block of the cache, the L1 flush and invalidate control register 0 (L1FINV0) is written with the cache set (L1FINV0[CSET]) and cache way (L1FINV0[CWAY]) of the line that is to be flushed. L1FINV0 is written using a **mtspr** instruction specifying the L1FINV0 register. No tag match in the cache is required. An additional field, L1FINV0[CCMD], is used to specify the type of flush to be performed on the line addressed by L1FINV0[CWAY] and L1FINV0[CSET].

The available L1FINV0[CCMD] encodings are described in Section 3.10.8, “L1 Flush and Invalidate Control Register 0 (L1FINV0) <DCF>.”

Direct cache flushing affects only the L1 data cache (or unified cache); the L1 instruction cache and any other caches in the cache hierarchy are not explicitly targeted.

6.3.3 Cache Way Partitioning <CWP>

Cache way partitioning allows ways in a unified L1 cache to be configured to accept either data or instruction miss line-fill replacements and is implemented with bits in L1CSR0 and L1CFG0:

- Way instruction disable field (L1CSR0[WID]) is a 4-bit field that determines which of ways 0–3 are available for replacement by instruction miss line refills.

  The additional ways instruction disable bit (L1CSR0[AWID]) determines whether ways 4 and above are available for replacement by instruction miss line refills.
- Way data disable field (L1CSR0[WDD]) is a 4-bit field that determines which of ways 0–3 are available for replacement by data miss line refills.

The additional ways data disable bit (L1CSR0[AWDD]) determines whether ways 4 and above are available for replacement by instruction miss line refills.

- Way access mode bit, L1CSR0[WAM], Determines whether all ways are available for access or only ways partitioned for the specific type of access are used for a fetch or read operation.
- Cache way partitioning available bit, L1CFG0[CWPA], indicates whether the cache way partitioning is available.

These fields are described in detail in Section 3.10.1, “L1 Cache Control and Status Register 0 (L1CSR0).” and in Section 3.10.6, “L1 Cache Configuration Register 0 (L1CFG0).”

6.3.3.1 Interaction with Cache Locking Operations

Cache way partitioning can affect the ability of cache line locking to control replacement of lines. If any cache line locking instruction (**icbtls**, **dcbtls**, **dcbtstls**) is allowed to execute and finds a matching line in the cache, the line’s lock bit is set regardless of the L1CSR0[WID,AWID,WDD,AWDD] settings. In this case, no replacement has been made.

However, for cache misses that occur while executing a cache line lock set instruction, the only candidate lines available for locking are those that correspond to ways of the cache that have not been disabled for the particular type of line locking instruction (controlled by WDD and AWDD for **dcbtls** and **dcbtstls**, controlled by WID and AWID for **icbtls**). Thus, an overlocking condition may result even though fewer than eight lines with the same index are locked.

6.4 Memory and Cache Coherency

The primary objective of a coherent memory system is to provide the same image of memory to all devices using the system. Enforcing coherency allows synchronization and cooperative use of shared resources. Otherwise, multiple copies of data corresponding to a memory location, some containing outdated values, could exist in a system and could cause errors when the outdated values are used. Each memory-sharing device must follow rules for managing cache state. This section describes the coherency mechanisms of the architecture and the cache coherency protocols that the Freescale devices support.

Unless specifically noted, the discussion of coherency in this section applies to the core data cache only. The instruction cache is not snooped for general coherency with other caches; however, it is snooped when the Instruction Cache Block Invalidate (**icbi**, **icbiep** <E.PD>) instruction is executed by this processor or any processor in the system.

6.4.1 Memory/Cache Access Attributes (WIMGE Bits)

Some memory characteristics can be set on a page basis by using the WIMGE bits in the translation lookaside buffer (TLB) entries. These bits allow both uniprocessor and multiprocessor system designs to exploit numerous system-level performance optimizations. WIMGE bits control the following:

- Write-through required (W bit)
- Caching-inhibited (I bit)
- Memory-coherency-required (M bit)
- Guarded (G bit)

- Endianness (E bit)

In addition to the WIMGE bits, the MMU model defines the following attributes on a page basis:

- User-definable (U0, U1, U2, U3). These bits, progammable through MAS3[54–57], are associated with a TLB entry and can be used by system software. For example, these bits may be used to hold information useful to a page scanning algorithm or be used to mark more abstract page attributes.

NOTE: Software Considerations

> Although U2 and U3 may be available in some implementations, software should avoid use of these because some implementations may only support U0 and U1.

The following optional attributes, are programmable through MMU assist register 2 (MAS2):

- Alternate coherency mode, programmed through MAS2[ACM], allows an implementation to employ multiple coherency methods and to participate in multiple coherency protocols on a per page basis. If the M attribute (memory coherence required) is not set (M = 0), the page has no associated coherency and the ACM attribute is ignored. If M is set, the ACM attribute determines the coherency domain (or protocol) used for the page. ACM values are implementation dependent.
- <VLE>:
  Variable-length encoding (VLE). The VLE attribute, MAS2[VLE], identifies pages that contain instructions from the VLE instruction set. If VLE = 0, instructions fetched from the page are decoded and executed as non-VLE instructions. If VLE = 1, instructions fetched from the page are decoded and executed as VLE instructions. For more information, see the VLE PEM.

Consult the user documentation to determine whether these attributes are implemented.

The WIMGE attributes are programmed by the operating system for each page. The W and I attributes control how the processor performing an access uses its own cache. The M attribute ensures that coherency is maintained for all copies of the addressed memory location. The G attribute prevents speculative loading from the addressed memory location. (An operation is said to be performed speculatively if, at the time that it is performed, it is not known to be required by the sequential execution model.) The E attribute defines the order in which the bytes that consist of a multiple-byte data object are stored in memory (big- or little-endian).

The WIMGE attributes occupy 5 bits in the TLB entries for page address translation. The operating system writes the WIMGE bits for each page into the TLB entries as it maps translations. For more information, see Section 6.5.3.1, “TLB Entries.”

All combinations of these attributes are supported except:

- Those that simultaneously specify a region as write-through-required and caching-inhibited. Write-through and caching-inhibited attributes are mutually exclusive because the write-through attribute permits the data to be in the data cache while the caching-inhibited attribute does not.
- Those that simultaneously specify a region as little-endian and VLE mode. <VLE>

In general, aliasing pages (creating TLB mappings to the same physical page using 2 or more different virtual addresses) for which the WIMGE attribute differ may cause coherency to be lost. Additionally, changing WIMGE attributes on a page may require additional steps by software to ensure coherency. More specific cases are described in the following sections that describe the individual page attributes.

Memory that is write-through-required or caching-inhibited is not intended for general-purpose programming. For example, load and reserve and store conditional instructions may cause the system data storage interrupt handler to be invoked if they specify a location in memory having either of these attributes. Some implementations take a data storage interrupt if the location is write-through but does not take the interrupt if the location is cache-inhibited.

6.4.2 Write-Through-Required Attribute (W)

A page marked W = 0 is generally considered to be write-back. Some implementations may implement certain caches in the cache hierarchy such as the L1 cache, to always write-through all cacheable stores. Such write-through characteristics may be limited to a specific portion of the cache hierarchy and the stores may be only written through to another cache which is the point of coherency for the processor. Processors that implement category Write Shadow Mode operate in this manner by writing any store data through the L1 data cache to the L2 cache such that no modified data is present in the L1 data cache.

A store to a write-through-required (W = 1) memory location is performed in main memory and may cause additional memory locations to be accessed. If a copy of the block containing the specified location is retained in the data cache, the store is also performed in the data cache. A store to write-through-required memory cannot cause a block to be put in a modified state in the data cache.

Also, if a store instruction that accesses a block in a location marked as write-through-required is executed when the block is already considered to be modified in the data cache, the block may continue to be considered to be modified in the data cache even if the store causes all modified locations in the block to be written to main memory. In some processors, accesses caused by separate store instructions that specify locations in write-through-required memory may be combined into one access. This is called store-gathering. Such combining does not occur if the store instructions are separated by a memory barrier (**sync** or an **mbar**).

Aliasing a page as both W = 0 and W = 1 is coherent when all accesses are performed by one processor. If some store instructions executed by a given processor access locations in a block as write-through-required and other store instructions executed by the same processor access locations in that block as write-back, software must ensure that the block cannot be accessed by another processor or mechanism in the system.

6.4.3 Caching-Inhibited Attribute (I)

A load instruction that specifies a location in caching-inhibited (I = 1) memory is performed to main memory and may cause additional locations in main memory to be accessed unless the specified location is also guarded. An instruction fetch from caching-inhibited memory may cause additional words in main memory to be accessed. No copy of the accessed locations is placed into the caches.

Aliasing caching-inhibited and cached memory operations should generally be avoided. If a load occurs to a caching-inhibited location for which a copy of the location exists in the cache hierarchy of the processor executing the memory access, it is implementation dependent whether the cached location is used to satisfy the load or whether the cache copy is ignored and the load is satisfied from main memory. If a store occurs to a caching inhibited location, the store is always performed in main memory even if a copy of the location exists in the cache hierarchy of the processor executing the store. It is implementation dependent whether the cache is updated with the stored value. If software must perform caching-inhibited accesses to a page that has been accessed as cached, it should flush the page (or lines that are intended to be accessed) from the cache prior to accessing them as caching-inhibited.

In some processors, nonoverlapping accesses caused by separate load instructions that specify locations in caching-inhibited memory may be combined into one access, as may nonoverlapping accesses caused by separate store instructions to caching-inhibited memory (that is, store-gathering). Such combining does not occur if the load or store instructions are separated by a memory barrier instruction or if the memory is also guarded.

6.4.4 Memory Coherence–Required Attribute (M)

Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are coherent if they are serialized, and no processor or mechanism can observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence: the physical storage location need not assume each of the values written to it. For example, a processor may update a location several times before the value is written to physical storage.

The result of a store operation is not available to every processor or mechanism at the same instant, and some mechanisms may observe only some of the values written to a location. However, when a location is accessed atomically and coherently by all processors and mechanisms, the sequence loaded from the location by any processor or mechanism during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a mechanism can never load an older value after it loads a newer one.

Memory coherence is managed in blocks called coherence blocks, whose size is implementation dependent but is usually the size of a cache block.

For storage that is not memory coherence required (M = 0), software must explicitly manage memory coherence to the extent required by program correctness; operations required for this may be system dependent.

Because the memory coherence required attribute for a given location is of little use unless all processors that access the location do so coherently, in this document, statements about memory coherence–required storage generally assume that the storage memory coherence is required for all processors that access it.

NOTE: Software Considerations

> Operating systems that allow programs to request that storage not be memory coherence required should provide services to assist in managing memory coherence for such storage, including all system-dependent aspects thereof.
>
> In most systems, the default is that all storage requires coherence. For some applications in some systems, software management of coherence may yield better performance. In such cases, a program can request that a given unit of storage not be memory coherence required and can manage the coherence of that storage by using the **sync** instruction, the cache management instructions, and services provided by the operating system.

6.4.5 Guarded Attribute (G)

Storage is said to be well-behaved if the corresponding physical storage exists and is not defective, and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

Storage is said to be guarded if the G bit is set in the TLB entry that translates the effective address. This setting can protect certain memory areas from read accesses made by the processor that are not dictated directly by the program. If areas of physical memory are not fully populated (in other words, there are holes in the physical memory map within this area), this setting can protect the system from undesired accesses caused by speculative load operations (referred to as ‘out of order’ accesses in Power ISA and described in Section 6.4.5.1, “Definition of Speculative and Out-of-Order Memory Accesses”) that could lead to a machine check exception. Also, the guarded bit can be used to prevent speculative load operations from occurring to certain peripheral devices that produce side effects.

In general, storage that is not well-behaved should be guarded. Because such storage may represent a control register on an I/O device or may include locations that do not exist, an out-of-order access to such storage may cause an I/O device to perform unintended operations or may result in a machine check.

Locations accessed by decorated storage instructions that are in caching-inhibited memory should always be marked as guarded.

Instruction fetching is not affected by the G bit. Software must set guarded pages to no execute (that is UX=0 and SX=0) to prevent instruction fetching from guarded storage.

Accesses to caching inhibited memory that is both guarded and caching-inhibited, aligned, with an atomic access will only be executed once. If the access is misaligned or not an atomic access, multiple accesses may be performed including accessing the same byte more than once.

The following rules apply to in-order execution of load and store instructions for which the first byte of the storage operand is in storage that is both caching inhibited and guarded.

- Load or store instruction that causes an atomic access

If any portion of the storage operand has been accessed, the instruction completes before the interrupt occurs if any of the following exceptions is pending.

- Any asynchronous exceptions that would cause an asynchronous interrupt.
- Load or store instruction that causes an alignment exception, a data TLB error exception, or that causes a data storage exception.

The portion of the storage operand that is in caching inhibited and guarded storage is not accessed.

6.4.5.1 Definition of Speculative and Out-of-Order Memory Accesses

In the Power ISA definition, the term ‘out of order’ replaced the term ‘speculative’ with respect to memory accesses to avoid a conflict between the word’s meaning in the context of execution of instructions past unresolved branches. Power ISA’s use of ‘out of order’ in this context could in turn be confused with the notion of loads and stores being reordered in a weakly ordered memory system.

In the context of memory accesses, this document uses the terms ‘speculative’ and ‘out of order’ as follows:

- Speculative memory access—An access that occurs before it is known to be required by the sequential execution model
- Out-of-order memory access—An access performed ahead of one that may have preceded it in the sequential model, such as is allowed by a weakly ordered memory model

6.4.5.2 Performing Operations Speculatively

An operation is said to be nonspeculative if it is guaranteed to be required by the sequential execution model. Any other operation is said to be performed speculatively, (out of order).

The processor performs operations speculatively on the expectation that the results will be needed by an instruction required by the sequential execution model. Whether they are needed depends on whether control flow is diverted away from the instruction by any event that changes the context in which the instruction is executed, such as an exception, branch, trap, system call, return from interrupt instruction.

Typically, the hardware performs operations speculatively when it has resources that would otherwise be idle, so the operation incurs little or no cost. If subsequent events such as branches or exceptions indicate that the operation would not have been performed, the processor abandons any results of the operation except as described below.

Most operations can be performed speculatively, as long as the machine appears to follow the sequential execution model. Speculative operations have the following restrictions:

- Stores—A store instruction cannot execute speculatively in a manner such that the alteration of the target location can be observed by other processors or mechanisms. EIS requires that stores are not executed speculatively.
- Accessing guarded memory—The restrictions for this case are given in Section 6.4.5.2.1, “Speculative Accesses to Guarded Memory.”

Only asynchronous machine check exceptions may be reported due to an operation that is performed speculatively, until such time as it is known that the operation is required by the sequential execution model. Other than asynchronous machine check, the only other permitted side effect of performing an operation speculatively is that nonguarded memory locations that could be fetched into a cache by non speculative execution may be fetched speculatively into that cache.

6.4.5.2.1 Speculative Accesses to Guarded Memory

Load accesses from guarded memory may be performed speculatively if a copy of the target location is in a cache; the location may be accessed from the cache or from main memory.

Note that software should ensure that only well behaved memory is loaded into a cache, either by marking as caching-inhibited (and guarded) all memory that may not be well behaved or by marking such memory caching-allowed (and guarded) and referring only to well behaved cache blocks.

6.4.5.2.2 Instruction Accesses: Guarded Memory and No-Execute Memory

The G bit is ignored for instruction fetches, and instructions are speculatively fetched from guarded pages. To prevent speculative fetches from pages that contain no instructions and are not well behaved, the page should be designated as no-execute (with the UX/SX page permission bits cleared). If the effective address of the current instruction is mapped to no-execute memory, an instruction storage exception is generated.

6.4.6 Byte-Ordering (Endianness) Attribute (E)

Objects may be loaded from or stored to memory in byte, half word, word, double word, or quad word units. For a particular data length, load and store operations are symmetrical; a store followed by a load of the same data object yields an unchanged value. The architecture does not guarantee the order in which the bytes that consist of multiple-byte data objects are stored into memory. The endianness (E) page attribute distinguishes between memory that is big or little endian, as described in the following subsections.

Except for instruction fetches, it is always permitted to access the same location using two effective addresses with different E bit settings. Instruction pages must be flushed from any caches before the E bit can be changed for those addresses. See Section 4.5.3.4, “Byte Ordering,” for more information about endianness.

Misaligned accesses which cross a page boundary for which the two pages have differing endian attributes produces a byte ordering exception.

6.4.6.1 Big-Endian Pages and Multiple-Byte Scalars

If a stored multiple-byte object is probed by reading its component bytes one at a time using load-byte instructions, the store order may be perceived. If such probing shows that the lowest memory address contains the highest-order byte of the multiple-byte scalar, the next-higher sequential address the next-least-significant byte, and so on, the multiple-byte object is stored in big-endian form. Big-endian memory is defined on a page basis by the memory/cache attribute, E = 0.

6.4.6.2 Little-Endian Pages and Multiple-Byte Scalars

If the probing described above shows that the lowest memory address contains the lowest-order byte of the multiple-byte scalar, the next-higher sequential address the next-most-significant byte, and so on, the multiple-byte object is stored in little-endian form. Setting E selects little-endian byte ordering for that page. For Power ISA embedded devices it is defined as true little-endian memory, not the version of little endian defined by PowerPC architecture–compliant devices.

6.4.6.3 Structure Mapping Examples

The following C programming example defines the data structure *S* used in this section to demonstrate how the bytes that consist of each element (a, b, c, d, e, and f) are mapped into memory. The structure contains scalars (shown in hexadecimal in the comments) and a sequence of characters, shown in single quotation marks.

```
struct {
   int a;  /* 0x1112_1314word                        */
   doubleb;/* 0x2122_2324_2526_2728double word      */
   char *c;/* 0x3132_3334word                        */
   chard[7];/* 'L','M','N','O','P','Q','R'array of bytes*/
   shorte; /* 0x5152   half word                     */
   int f;  /* 0x6162_6364word                        */
} S;
```

This figure shows the big-endian mapping of the structure.

Contents	11	12	13	14	(x)	(x)	(x)	(x)
Address	00	01	02	03	04	05	06	07
Contents	21	22	23	24	25	26	27	28
Address	08	09	0A	0B	0C	0D	0E	0F
Contents	31	32	33	34	‘L’	‘M’	‘N’	‘O’
Address	10	11	12	13	14	15	16	17
Contents	‘P’	‘Q’	‘R’	(x)	51	52	(x)	(x)
Address	18	19	1A	1B	1C	1D	1E	1F
Contents	61	62	63	64	(x)	(x)	(x)	(x)
Address	20	21	22	23	24	25	26	27

Figure 6-1. Big-Endian Mapping of Structure *S*

NOTE

The MSB of each scalar is at the lowest address. The mapping uses padding (indicated by (x)) to align the scalars—4 bytes between elements a and b, 1 byte between d and e, and 2 bytes between e and f. Note that the padding is determined by the compiler, not the architecture.

This figure shows the structure using little-endian mapping, showing double words laid out with addresses increasing from right to left.

Contents	(x)	(x)	(x)	(x)	11	12	13	14
Address	07	06	05	04	03	02	01	00
Contents	21	22	23	24	25	26	27	28
Address	0F	0E	0D	0C	0B	0A	09	08
Contents	‘O’	‘N’	‘M’	‘L’	31	32	33	34
Address	17	16	15	14	13	12	11	10
Contents	(x)	(x)	51	52	(x)	‘R’	‘Q’	‘P’
Address	1F	1E	1D	1C	1B	1A	19	18
Contents	(x)	(x)	(x)	(x)	61	62	63	64
Address	27	26	25	24	23	22	21	20

Figure 6-2. Little-Endian Mapping of Structure *S*—Alternate View

6.4.7 Mismatched Memory Cache Attributes

Aliased accesses to the same memory location using two effective addresses for which the write-through required attribute (W bit) differs meet the memory coherence requirements described in Section 6.4.2, "Write-Through-Required Attribute (W)," if the accesses are performed by a single processor. If the accesses are performed by two or more processors, coherence is enforced by the hardware only if the write-through required attribute is the same for all the accesses.

Loads, stores, **dcbz**, **dcbzl** <DEO>, **dcbzep** <E.PD>, and **dcbzlep** <E.PD, DEO> instructions, and instruction fetches to the same memory location using two effective addresses for which the caching-inhibited attribute (I bit) differs must meet the requirement that a copy of the target location of an access to caching-inhibited memory not be in the cache. Violation of this requirement is considered a programming error; software must ensure that the location has not previously been brought into the cache or, if it has, that it has been flushed from the cache. If the programming error occurs, the result of the access is boundedly undefined and it is implementation dependent whether the access bypasses the cache or is performed in the cache. It is not considered a programming error if the target location of any other cache management instruction to caching-inhibited memory is in the cache.

Accesses to the same memory location using two effective addresses for which the memory coherence attribute (M bit) differs may require explicit software synchronization before accessing the location with M = 1 if the location has previously been accessed with M = 0. Any such requirement is implementation and system-dependent. For example, in some systems that use bus snooping, no software synchronization may be required. In some directory-based systems, software may be required to execute **dcbf** instructions on each processor to flush all cache entries accessed with M = 0 before accessing those locations with M = 1.

Accesses to the same memory location using two effective addresses for which the guarded attribute (G bit) differs are always permitted.

Except for instruction fetches, accesses to the same memory location using two effective addresses for which the endian storage attribute (E bit) differs are always permitted as described in Section 6.4.6, "Byte-Ordering (Endianness) Attribute (E)." Instruction memory locations must be flushed before the endian attribute can be changed for those addresses.

The requirements on mismatched user-defined memory attributes (U0–U3) is implementation-dependent.

6.4.7.1 Coherency Paradoxes and WIMGE

Careless programming of the WIMGE bits may create situations that present coherency paradoxes to the processor. These paradoxes can occur within a single processor or across several processors. It is important to note that, in the presence of a paradox, the operating system software is responsible for correctness.

In particular, a coherency paradox can occur when the state of these bits is changed without appropriate precautions (such as flushing the pages that correspond to the changed bits from the caches of all processors in the system) or when the address translations of aliased real addresses specify different values for certain WIMGE bit values. For more information, see Section 6.4.7, "Mismatched Memory Cache Attributes."

Support for M = 1 memory is optional. Cache attribute settings where both W = 1 and I = 1 are not supported. For all supported combinations of the W, I, and M bits, both G and E may be 0 or 1.

6.4.7.2 Self-Modifying Code

When a processor modifies any memory location that can contain an instruction, software must ensure that the instruction cache is made consistent with data memory and that the modifications are made visible to the instruction fetching mechanism. This must be done even if the cache is disabled or if the page is marked caching inhibited.

The following instruction sequence can be used to accomplish this when the instructions being modified are in memory that is memory-coherence required and one processor both modifies the instructions and executes them. (Additional synchronization is needed when one processor modifies instructions that another will execute.)

The following sequence synchronizes the instruction stream (using either **dcbst** or **dcbf**):

```
dcbst (or dcbf)    // force modified instructions to memory from data cache
sync               // insure dcbst completed
icbi               // invalidate old copies of modified instruction in icache
sync               // ensure icbi completed
isync              // remove any old copies of modified instructions that may have been
                   // fetched, but not completed (force refetch to get new
                   // modified instructions)
```

6.4.8 Shared Memory

The architecture supports sharing memory between programs, between different instances of the same program, and between processors and other mechanisms. It also supports access to a memory location by one or more programs using different effective addresses. In these cases, memory is shared in blocks that are an integral number of pages. When one physical memory location has different effective addresses, the addresses are said to be aliases. Each application can be granted separate access privileges to aliased pages.

Section 6.4.8.6, “Lock Acquisition and Import Barriers,” gives examples of how **sync** and **mbar** are used to control memory access ordering when memory is shared among programs.

6.4.8.1 Memory Access Ordering

Memory access ordering is weakly consistent. This provides an opportunity for improved performance over a model with stronger consistency rules but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed for correct execution of the program.

Strongly-consistent ordering ensures that all programs that share a data resource observe updates to that memory in the same order in which updates occur.

In a weakly ordered system, the order in which a processor accesses memory, the order in which those accesses are performed with respect to other processors or mechanisms, and the order in which they are performed in main memory may all be different. Although such a model creates opportunities for processors to reschedule memory transactions based on the dynamics of the instruction stream and memory traffic, it shifts the burden onto the programmer to force synchronization when memory transactions must occur in a specified order, and the architecture provides instructions for efficient handling of synchronization specific to the need.

In almost all applications, concern over memory transaction order is normally isolated to a few specialized OS routines such as interprocessor communication routines, library routines for locking and unlocking shared memory data structures, and I/O device drivers. Because such areas of increased programming difficulty are characterized easily enough, the resulting performance improvement easily justifies the use of a weakly ordered storage model.

While a weakly ordered memory model appears extraordinarily complicated to the programmer, in fact several restrictions placed by EIS simplify this greatly. In practice this requires that the programmer only really be aware of the ordering of memory accesses that are used by another processor or another device and the other processor or device care about the order. In general, this reduces even further to the following three scenarios:

1. The shared memory multiprocessor (SMP) case
   - Code is running on more than one processor.
   - Data being manipulated is accessed from more than one processor.
   - Software is designed, in general, with some sort of mutual exclusion or locking mechanism regardless of the architecture (because software running on one processor must make several updates to data structures atomically).
2. The device driver case
   - Code is running that controls a device through memory-mapped addresses.
   - Accesses to these memory-mapped registers usually need to occur in a specific order because the accesses have side effects (for example a store to an address causes the device to perform some action and the order these actions are performed must be explicitly controlled in order for the device to perform correctly).
   - Addresses are usually marked as caching-inhibited and guarded because the memory is not "well behaved."
3. The processor synchronization case
   - Some registers within the processor, such as the MSR, have special synchronization requirements associated with them to guarantee when changes which may effect memory accesses, occur. (see Section 4.5.4.3, "Synchronization Requirements" for the specific registers and their synchronization requirements).
   - Only system programmers modifying these special registers need be aware of these cases.

6.4.8.2 Architecture Ordering Requirements

EIS implicitly requires processors to order certain types of memory accesses. Other accesses require software to execute memory barrier instructions to enforce ordering. The implicit memory access rules are as follows:

1. All loads and stores appear to execute in-order on the same processor. That is, each memory access a processor performs, if that memory location is not stored to by another processor or device, it will appear to be performed in order to the processor. For example, a processor executes the following sequence:

```
lwz     r3,0(r4)
lwz     r5,100(r4)
```

Because there is no way for a single processor to distinguish which order these loads occurred in (because the memory is “well behaved”), the loads can be performed in any order. Similarly the sequence:

```
stw     r3,0(r4)
stw     r5,100(r4)
```

may also be performed out of order because a single processor cannot distinguish which order the stores are performed in. However, the sequence:

```
stw     r3,0(r4)
lwz     r5,0(r4)
```

must be performed in order on the same processor because the processor can distinguish a difference depending on whether the store or the load is performed first.

In general this means that the processor performs memory accesses in order between any two accesses to overlapping bytes. The processor may decide that accesses overlap if they touch a larger subset and not merely a common byte, but such ordering is not required by EIS.

Such accesses to overlapping bytes may not occur in order external to the performing processor and the processor that performs these may actually satisfy the load from a store queue or other implementation dependent buffer.

2. Any load or store that depends on data from a previous load or store must be performed in order. For example, a load retrieves the address that is used in a subsequent load:

   ```
   lwz     r3,0(r4)
   lwz     r5,0(r3)
   ```

   Because the second load’s address depends on the first load being performed and providing data, the processor must ensure that the first load occurs before the second is attempted (and in fact must be sure the first load has returned data before even attempting translation).
3. Guarded caching-inhibited stores must be performed in order with respect to other guarded caching-inhibited stores and guarded caching-inhibited loads must be performed in order with respect to other guarded caching-inhibited loads. This generally only applies to writing device drivers that control memory mapped devices with side effects through store operations.
4. A store operation cannot be performed before a previous load operation regardless of the addresses. That is if a load is followed by a store, then the load will always be performed before the store is. This is an EIS requirement of Freescale processors. It is unlikely, but possible, that other Power Architecture® cores may not require this.

This table describes how the architecture defines requirements for ordering of loads and stores.

Table 6-2. Load and Store Ordering

Access Type	Architecture Definition
Load ordering with respect to other loads	The architecture guarantees that loads that are both caching-inhibited (I = 1) and guarded (G = 1) are not reordered with respect to one another.
	If a load instruction depends on the value returned by a preceding load (because the value is used to compute the EA specified by the second load), the corresponding memory accesses are performed in program order with respect to any processor or mechanism to the extent required by the associated memory coherence required attributes (that is, the memory coherence required attribute, if any, associated with each access). This applies even if the dependency does not affect program logic (for example, the value returned by the first load is ANDed with zero and then added to the EA specified by the second load).
Store ordering with respect to other stores	If two store instructions specify memory locations that are both caching inhibited and guarded, the corresponding memory accesses are performed in program order with respect to any processor or mechanism. Otherwise, stores are weakly ordered with respect to one another.
Load followed by store	Any store instruction will not be performed until all previous load instructions on the same processor have been performed.instructions specify memory locations that are both caching inhibited and guarded, the corresponding memory accesses are performed in program order with respect to any processor or mechanism. EIS does not allow the implementation of speculative stores.
Store ordering with respect to loads	The architecture specifies that a memory barrier must be used to ensure sequential ordering of loads with respect to stores.

6.4.8.3 Forcing Load and Store Ordering (Memory Barriers)

The implicit ordering requirements enforced by the processor handle the vast majority of all the programming cases when accessing memory locations from a single core. Normal software should only be concerned about ordering when the memory locations being accessed are done so in an SMP environment or the memory locations are part of a device’s memory mapped locations (that is, non-well behaved memory). If these cases occur, then software must place explicit memory barriers to control the order of memory accesses. A memory barrier causes ordering between memory accesses that occur before the barrier in the instruction stream and memory accesses that occur after the barrier in the instruction stream.

There are four memory barriers that can be used to order memory accesses, depending on the type of memory (the WIMGE attributes) being accessed and the level of performance desired. Memory barriers, by definition, can slow down the processor because they prevent the processor from performing loads and stores in their most efficient order. The barriers from strongest (that is, enforces the most ordering between different types of accesses) to the weakest are:

- **sync** (or **sync** 0 or **msync**)—**sync** creates a barrier such that *all* (regardless of WIMGE attributes) memory accesses that occur before the **sync** are performed before any accesses after the **sync**. **sync** also ensures that no other instructions after the **sync** are initiated until the instructions before the **sync** and the **sync** itself, have performed their operations. **sync** also has the most negative effect on performance. **sync** can be used regardless of the memory attributes of the access and can be used in the place of any of the other barriers. However, it should only be used when performance isn’t an issue, or if no other barrier orders the memory accesses. **sync** also orders stores and the effects of **msgsnd** instructions.

- **mbar** (or **mbar** 0)—**mbar** creates the same barrier **sync** does, however it does not restrict instructions following **mbar** from being initiated. It does prevent memory accesses following the **mbar** from being performed until all the memory accesses prior to the **mbar** have been performed. **mbar** affects performance almost as much as **sync** does.
- **mbar** 1—**mbar** 1 creates a memory barrier that is the same as the **eieio** instruction from the original PowerPC architecture. It creates two different barriers:
  - Loads and stores that are both caching-inhibited and guarded (WIMGE = 0b01*x*1*x*) as well as stores that are write-through required (WIMGE = 0b10*xxx*). This is useful for the device driver case which would be doing loads and stores to caching-inhibited memory.
  - Stores that have the following attributes: not caching-inhibited, not write-through required, and memory coherence required (WIMGE = 0b001*xx*). These are stores to normal cacheable coherent memory.

  mbar 1 is a better performing memory barrier than **sync** or **mbar**.
- **lwsync** (or **sync** 1)—**lwsync** (lightweight sync) creates a barrier for normal cacheable memory accesses (WIMGE = 0b001*xx*). It orders all combinations of the loads and stores *except* for a store followed by a load.

  lwsync is a better performing memory barrier than **sync**, **mbar**, or **mbar** 1.

Another method also exists for ordering all caching-inhibited loads and stores which are guarded. The HID0[CIGLSO] bit can be set to force all caching-inhibited loads and stores which are guarded to be performed in order. This is not a barrier, per se, but a system attribute that causes the processor to always order these accesses and also ensures that the system interconnect does not reorder them as well. Setting this bit is a good way to deal with the device driver case over a broad range of code if the memory accesses to the device are caching-inhibited and guarded which is normally the case. This is likely to perform better than inserting an **mbar** in specific places since load-load, store-store, and load-store are already ordered by the architecture for guarded, caching inhibited loads and stores. For a guarded, caching-inhibited store followed by a guarded caching-inhibited load many implementations can handle this efficiently by simply ensuring that the processor performs the store prior to starting the load.

NOTE

Not all implementations support all barriers. See the core reference manual.

6.4.8.4 Architectural Memory Access Ordering

Table 6-3 displays the Power ISA and *EIS* memory access ordering requirements based on the WIMG attributes and access type. For access where the attributes differ, ordering between these types of access generally requires **mbar** 0 (or **sync**) except that write-through required and guarded caching inhibited loads or stores may be ordered with **mbar** 1. For Table 6-3, entries list the barriers in order from the likely most efficient to least efficient. Efficiency between different types of barriers may differ between implementations. Some older implementations may not provide all forms of barriers, but all implementations provide sync (**sync** 0 or **msync**) and **mbar** 0. ‘Yes’ means that the given ordering is already guaranteed by the architecture and no barrier is required. Not all possible barriers are listed and **sync** 0 or **mbar** 0 will enforce all barriers.

Table 6-3. Architectural Memory Access Ordering

Memory Access Attributes	WIMGE	Store-Store Ordered	Load-Load Ordered	Store-Load Ordered	Load-Store Ordered
Caching-inhibited and Guarded	0b01x1x	Yes	Yes	HID0[CIGLSO] mbar 1 mbar 0 sync	Yes
Caching-inhibited and non Guarded	0b01x0x	mbar 0 sync	mbar 0 sync	mbar 0 sync	Yes
Write-through	0b10xxx	mbar 1 mbar 0 sync	mbar 0 sync	mbar 0 sync	Yes
Write-back	0b00xxx	lwsync mbar 0 sync	lwsync mbar 0 sync	mbar 0 sync	Yes

6.4.8.5 Memory Barrier Example

When a processor (P1) executes **sync**, a memory barrier is created that separates applicable memory accesses into two groups, G1 and G2. G1 includes all applicable memory accesses associated with instructions preceding the barrier-creating instruction, and G2 includes all applicable memory accesses associated with instructions following the barrier-creating instruction.

This figure shows an example using a two-processor system.

Processor 1 (P1)	Memory Access Groups G1 and G2	Processor 2 (P2)
Instruction 1	G1: Memory accesses generated by P1 before the memory barrier	When memory coherence is required, G1 accesses that affect P2 are also performed before the memory barrier.
Instruction 2		
Instruction 3		
Instruction 4		
Instruction 5 (**sync**)—Memory barrier		Barrier generated by P1 does not order P2 instructions or associated accesses with respect to other P2 instructions and associated accesses.
Instruction 6	G2: Memory accesses generated by P1 after the memory barrier	When memory coherence is required, G2 accesses that affect P2 are also performed after the memory barrier.
Instruction 7		
Instruction 8		
Instruction 9		
Instruction 10		

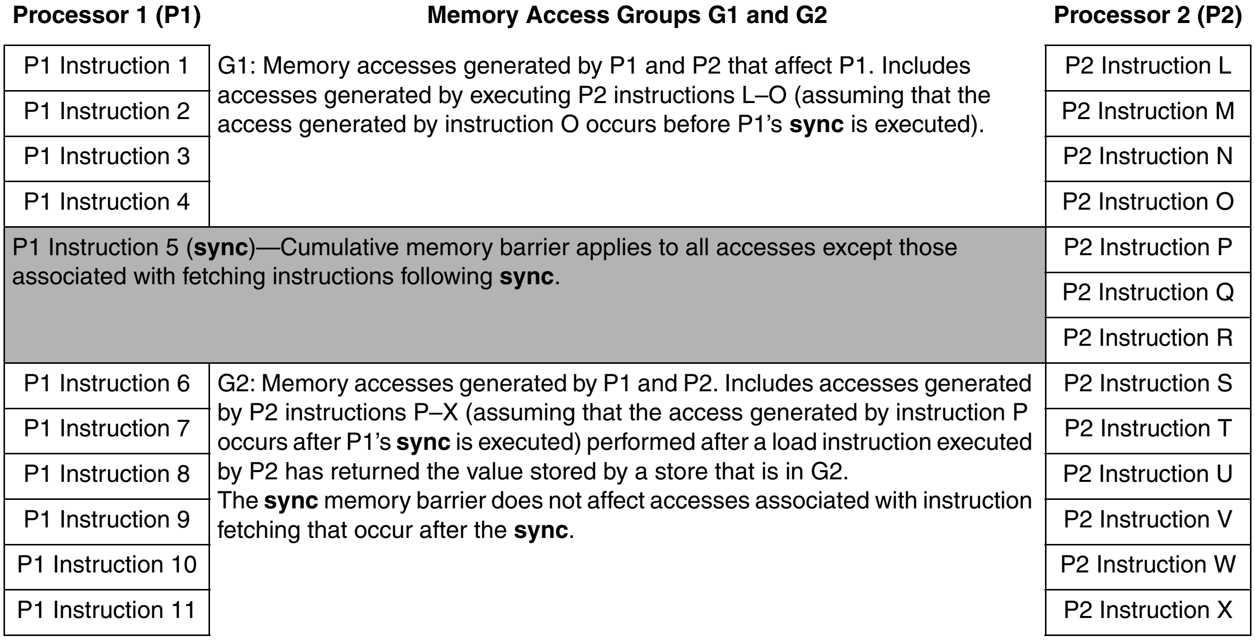
Figure 6-3. Memory Barrier when Coherency is Required (M = 1)

The memory barrier ensures that all memory accesses in G1 are performed with respect to any processor or mechanism, to the extent required by the associated memory coherence required attributes (that is, the memory-coherence required attribute, if any, associated with each access), before any memory accesses in G2 are performed with respect to that processor or mechanism.

The ordering enforced by a memory barrier is said to be cumulative if it also orders memory accesses that are performed by processors and mechanisms other than P1, as follows:

- G1 includes all applicable memory accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
- G2 includes all applicable memory accesses by any such processor or mechanism that are performed after a load instruction executed by that processor or mechanism has returned the value stored by a store that is in G2.

This figure shows an example of a cumulative memory barrier in a two-processor system.



Processor 1 (P1)	Memory Access Groups G1 and G2	Processor 2 (P2)
P1 Instruction 1	G1: Memory accesses generated by P1 and P2 that affect P1. Includes accesses generated by executing P2 instructions L–O (assuming that the access generated by instruction O occurs before P1’s **sync** is executed).	P2 Instruction L
P1 Instruction 2		P2 Instruction M
P1 Instruction 3		P2 Instruction N
P1 Instruction 4		P2 Instruction O
P1 Instruction 5 (**sync**)—Cumulative memory barrier applies to all accesses except those associated with fetching instructions following **sync**.		P2 Instruction P
		P2 Instruction Q
		P2 Instruction R
P1 Instruction 6	G2: Memory accesses generated by P1 and P2. Includes accesses generated by P2 instructions P–X (assuming that the access generated by instruction P occurs after P1’s **sync** is executed) performed after a load instruction executed by P2 has returned the value stored by a store that is in G2. The **sync** memory barrier does not affect accesses associated with instruction fetching that occur after the **sync**.	P2 Instruction S
P1 Instruction 7		P2 Instruction T
P1 Instruction 8		P2 Instruction U
P1 Instruction 9		P2 Instruction V
P1 Instruction 10		P2 Instruction W
P1 Instruction 11		P2 Instruction X

Figure 6-4. Cumulative Memory Barrier

A memory barrier created by **sync** is cumulative and applies to all accesses except those associated with fetching instructions following the **sync**.

6.4.8.5.1 Programming Considerations

Because instructions following an **isync** cannot execute until all instructions preceding **isync** have completed, if an **isync** follows a conditional branch instruction that depends on the value returned by a preceding load instruction, that load is performed before any loads caused by instructions following the **isync**. This is true even if the effects of the dependency are independent of the value loaded (for example, the value is compared to itself and the branch tests CR*n*[EQ]), and even if the branch target is the next sequential instruction.

Except for the cases described above and earlier in this section, data and control dependencies do not order memory accesses. Examples include the following:

- If a load specifies the same memory location as a preceding store and the location is not caching inhibited, the load may be satisfied from a store queue (a buffer into which the processor places stored values before presenting them to the memory subsystem) and not be visible to other processors and mechanisms. As a result, if a subsequent store depends on the value returned by the load, the two stores need not be performed in program order with respect to other processors and mechanisms.
- Because a store conditional instruction may complete before its store is performed, a conditional branch instruction that depends on the CR0 value set by a store conditional instruction does not order that store with respect to memory accesses caused by instructions that follow the branch. For example, in the following sequence, the **stw** is the **bc** instruction’s target:

  ```
  stwcx.
  bc
  stw
  ```

  To complete, the **stwcx.** must update the architected CR0 value, even though its store may not have been performed. The architecture does not require that the store generated by the **stwcx.** must be performed before the store generated by the **stw**.
- Because processors may predict branch target addresses and branch condition resolution, control dependencies (branches, for example) do not order memory accesses except as described above. For example, when a subroutine returns to its caller, the return address may be predicted, with the result that loads caused by instructions at or after the return address may be performed before the load that obtains the return address is performed.

Some processors implement nonarchitected duplicates of architected resources such as GPRs, CR fields, and the LR, so resource dependencies (for example, specification of the same target register for two load instructions) do not force ordering of memory accesses.

Examples of correct uses of dependencies, **sync**, and **mbar** to order memory accesses can be found in Section D.1, “Synchronization.”

Because the memory model is weakly consistent, the sequential execution model as applied to instructions that cause memory accesses guarantees only that those accesses appear to be performed in program order with respect to the processor executing the instructions. For example, an instruction may complete, and subsequent instructions may be executed, before memory accesses caused by the first instruction have been performed. However, for a sequence of atomic accesses to the same memory location for which memory coherence is required, the definition of coherence guarantees that the accesses are performed in program order with respect to any processor or mechanism that accesses the location coherently, and similarly if the location is one for which caching is inhibited.

Because caching-inhibited memory accesses are performed in main memory, memory barriers and dependencies on load instructions order such accesses with respect to any processor or mechanism even if the memory is not marked as requiring memory coherence.

6.4.8.5.2 Programming Examples

Example 1 shows cumulative ordering of memory accesses preceding a memory barrier, Example 2 shows cumulative ordering of memory accesses following a memory barrier. In both examples, assume that locations X, Y, and Z initially contain the value 0. In both, cumulative ordering dictates that the value loaded from location X by processor C is 1.

Programming Example 1:

- Processor A stores the value 1 to location X.
- Processor B loads from location X obtaining the value 1, executes a **sync**, then stores the value 2 to location Y.
- Processor C loads from location Y obtaining the value 2, executes a **sync**, then loads from location X.

Programming Example 2:

- Processor A stores the value 1 to location X, executes a **sync**, then stores the value 2 to location Y.
- Processor B loops, loading from location Y until the value 2 is obtained, then stores the value 3 to location Z.
- Processor C loads from location Z obtaining the value 3, executes a **sync**, then loads from location X.

The following subsections show examples of shared memory usage. These examples show the usage of memory barrier instructions and load and reserve and store conditional instructions. In addition, these examples assume that the memory used is well behaved cacheable and coherent memory (WIMGE = 0b001xx). Some older processors may not support the memory barriers shown in this example, however other appropriate memory barriers can be substituted.

In the examples, **lwarx** and **stwcx.** are used, however, these can be replaced as by other load and reserve or store conditional instructions that offer different data sizes. In each case, the substitution is straightforward although the same data size should be used for both the load and reserve and store conditional instructions. For example **ldarx** <64> and **stdcx.** <64> can be used in place of **lwarx** and **stwcx.**.

6.4.8.6 Lock Acquisition and Import Barriers

An import barrier is an instruction or instruction sequence that prevents memory accesses caused by instructions following the barrier from being performed before memory accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock has been acquired. A **sync** can always be used as an import barrier, but the approaches shown below generally yield better performance because they order only the relevant memory accesses.

6.4.8.6.1 Acquire Lock and Import Shared Memory

lwarx and **stwcx.** are used to obtain the lock, and an import barrier can be constructed by placing an **lwsync** immediately following the loop containing the **lwarx** and **stwcx.**. The following example uses the compare and swap primitive (see Section D.1.1, “Synchronization Primitives”) to acquire the lock.

This example assumes that all memory is well behaved cacheable memory (WIMGE = 0b001xx), the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```
loop:
    lwarx   r6,0,r3     # load lock and reserve
    cmpw    r4,r6       # skip ahead if
    bne     wait        # lock not free
    stwcx.  r5,0,r3     # try to set lock
    bne     loop        # loop if lost reservation
    lwsync              # import barrier
    lwz     r7,data1(r9)# load shared data
.
.
wait: ...               #wait for lock to free
    lwzx    r6,0,r3     # load lock
    cmpw    r4,r6       # keep waiting if
    bne     wait        # lock not free

    b       loop        # try lock again
```

This example continues to load and reserve the lock word and performs a **stwcx**. to set the lock. If the lock is successful, then the load of the lock word must be barriered from the load of data1 protected by the lock. The lwsync performs a ‘load with load’ barrier which insures that load of data1 occurs after the load of the lock (which was free). An alternate way to perform the barrier is to use **isync** although that may perform slower in some implementations. Using **isync** for the barrier, the second **bne** does not complete until CR0 has been set by the **stwcx.**. The **stwcx.** does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the **stwcx.** completes successfully. Together, the second **bne** and the subsequent **isync** create an import barrier that prevents the load from data1 from being performed until the branch is resolved to be not taken.

6.4.8.6.2 Acquire Lock and Import Shared Memory Using sync and mbar

A **sync** instruction can always be used as an import barrier, independent of the memory control attributes (for example, presence or absence of the caching inhibited attribute) of the memory containing the lock and the shared data structure. When either the lock or the shared data structure is in memory that is caching inhibited or write through required a **sync** or **mbar 0** instruction is generally required as an import barrier. In general, an **mbar 0** will perform better than **sync** on most implementations.

This example assumes that all memory is not well behaved memory, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```
loop:
    lwarx   r6,0,r3     # load lock and reserve
    cmpw    r4,r6       # skip ahead if
    bne     wait        # lock not free
    stwcx.  r5,0,r3     # try to set lock
    bne     loop        # loop if lost reservation
    sync    (or mbar 0) # import barrier
    lwz     r7,data1(r9)# load shared data
.
.
wait: ...               #wait for lock to free
```

```
      lwzx    r6,0,r3      # load lock
      cmpw    r4,r6        # keep waiting if
      bne     wait         # lock not free

      b       loop         # try lock again
```

This example continues to load and reserve the lock word and performs a **stwcx**. to set the lock. If the lock is successful, then the load of the lock word must be barriered from the load of data1 protected by the lock. The **sync** (or **mbar 0**) performs a barrier which insures that load of data1 occurs after the load of the lock (which was free). An alternate way to perform the barrier is to use **isync** which may perform faster in some implementations. Using **isync** for the barrier, the second **bne** does not complete until CR0 has been set by the **stwcx.**. The **stwcx.** does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the **stwcx.** completes successfully. Together, the second **bne** and the subsequent **isync** create an import barrier that prevents the load from data1 from being performed until the branch is resolved to be not taken.

Most processors do not allow load and reserve or store conditional instructions to target memory which is caching inhibited or write through required. For this reason, the lock word will normally be in memory that is well behaved memory, although the shared data will be in memory that is either write through required or caching inhibited.

6.4.8.6.3 Obtain Pointer and Import Shared Memory

If **lwarx** and **stwcx.** are used to obtain a pointer into a shared data structure, an import barrier is not needed if all the accesses to the shared data structure depend on the value obtained for the pointer. The following example uses the fetch and add primitive (see Section D.1.1, “Synchronization Primitives”) to obtain and increment the pointer.

In this example, it is assumed that the address of the pointer is in GPR 3, the value to be added to the pointer is in GPR 4, and the old value of the pointer is returned in GPR 5.

```
loop:   lwarx    r5,0,r3          # load pointer and reserve
        add      r0,r4,r5         # increment the pointer
        stwcx.   r0,0,r3          # try to store new value
        bne      loop             # loop if lost reservation
        lwz      r7,data1(r5)     # load shared data
```

The load from data1 cannot be performed until the **lwarx** loads the pointer value into GPR 5. The load from data1 may be performed out of order before the **stwcx.**. But if the **stwcx.** fails, the branch is taken and the value returned by the load from data1 is discarded. If the **stwcx.** succeeds, the value returned by the load from data1 is valid even if the load is performed out of order, because the load uses the pointer value returned by the instance of the **lwarx** that created the reservation used by the successful **stwcx.**.

An **isync** could be placed between the **bne** and the subsequent **lwz**, but no **isync** is needed if all accesses to the shared data structure depend on the value returned by the **lwarx**.

6.4.8.7 Lock Release and Export Barriers

An export barrier is an instruction or sequence of instructions that prevents the store that releases a lock from being performed before stores caused by instructions preceding the barrier have been performed. An export barrier can be used to ensure that all stores to a shared data structure protected by a lock be performed with respect to any other processor (to the extent required by the associated memory coherence required attributes) before the store that releases the lock is performed with respect to that processor. A **sync** can always be used as an export barrier, but the approaches shown below generally yield better performance because they order only the relevant memory accesses.

6.4.8.7.1 Export Shared Memory and Release Lock

The export barrier can be constructed by placing an **lwsync** between the store of the shared data and the store which frees the lock which protects the shared data..

This example assumes that all memory is well behaved cacheable memory (WIMGE = 0b001xx), the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw      r7,data1(r9)      # store shared data (last)
lwsync                     # export barrier
stw      r4,lock(r3)       # release lock
```

The **lwsync** ensures that the store that releases the lock is not performed with respect to any other processor until all stores caused by instructions preceding the **lwsync** have been performed with respect to that processor.

6.4.8.7.2 Export Shared Memory and Release Lock Using sync or mbar

A **sync** instruction can always be used as an export barrier, independent of the memory control attributes (for example, presence or absence of the caching inhibited attribute) of the memory containing the lock and the shared data structure. When either the lock or the shared data structure is in memory that is caching inhibited or write through required a **sync** or **mbar 0** instruction is generally required as an export barrier. In general, an **mbar 0** will perform better than **sync** on most implementations.

In this example it is assumed that the lock is in memory that is caching inhibited, the shared data structure is in memory that is not caching inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw      r7,data1(r9)      # store shared data (last)
sync (or mbar 0)           # export barrier
stw      r4,lock(r3)       # release lock
```

The **sync** or **mbar 0** ensures that the store that releases the lock is not performed with respect to any other processor until all stores caused by instructions preceding the **sync** or **mbar 0** have been performed with respect to that processor.

If both the lock word and the shared data structure are in memory that is in write through required, **mbar 1** may be used as an export barrier.

6.5 MMU Architecture

This section describes the memory management structure defined by the architecture. EIS supports demand-paged virtual memory. Address translation misses and protection faults cause precise exceptions. Sufficient information is available to correct the fault and restart the faulting instruction.

The architecture defines four different types of addresses:

Effective address (EA) An EA is an address that is specified by software, either as the address of an instruction (for example, the address saved in the link register when a branch and link instruction is executed), or the data address computed by a storage or cache management instruction (for example, the value of the contents of rA added to the contents of rB for a load byte indexed instruction). EAs are addresses for which a program normally uses to reference storage. In 32-bit mode, EAs only use the lower 32 bits of the calculated address on 64-bit implementations.

Virtual address (VA) A VA is an address that is formed from using the EA and supplying state from the current address space context. The address space context is managed by the operating system and hypervisor <E.HV>. In particular the following state is part of the VA:

- EA
- MSR[IS] or MSR[DS] depending on whether the EA was an instruction fetch (MSR[IS]) or a data reference (MSR[DS])
- PID*n*
- LPIDR <E.HV>
- MSR[GS] <E.HV>

If category E.PD is implemented, then external PID instructions substitute values from fields in EPLC or EPSC registers for all of the address space context on a data access. See Section 6.5.4, “Address Translation.” A VA is translated to a real address through a translation specified by a TLB entry.

<E.HV>:

Logical address (LA) An LA is an address which a guest operating system uses as a real address. The hypervisor translates a logical address to a real address when the guest writes a TLB entry. The guest written TLB entry represents what a guest operating system defines as a translation. The LA also includes the LPID value.

Real address (RA) An RA is a real or physical address. The RA is what the processor sends in transactions with the rest of the system. The system may provide other methods of translating RAs, but such translations occur outside the domain of the processor and are system specific.

Each program on a 64-bit or 32-bit implementation can access 2^{64} or 2^{32} bytes of effective address (EA) space, subject to limitations imposed by the operating system. Some devices also support physical addresses larger than 32 bits but not as large as 64 bits. Typically, each program’s EA space is a subset of a larger virtual address (VA) space managed by the operating system.

Each effective address is translated to a real address before being used to access physical memory or an I/O device. Hardware does this by using the address translation mechanism described in Section 6.5.4, “Address Translation.” The operating system (and hypervisor <E.HV>) manages the physically addressed resources of the system by setting up the tables used by the address translation mechanism.

The architecture divides the effective address space into pages. The page represents the granularity of effective address translation, permission control, and memory/cache attributes. Differing page sizes may be simultaneously supported. For an effective-to-real address translation to exist, a valid entry for the page containing the effective address must be in a translation lookaside buffer (TLB). Addresses for which no TLB entry exists cause exceptions.

The instruction addresses generated by a program and the addresses used by load, store, and cache management instructions are effective addresses. However, in general, the physical memory space may not be large enough to map all the virtual pages used by the currently active applications. With support provided by hardware, the operating system can attempt to use the available real pages to map enough virtual pages for an application. If a sufficient set is maintained, paging activity is minimized, therefore maximizing performance.

The operating system can restrict access to virtual pages by selectively granting permissions for user-state read, write, and execute, and supervisor-state read, write, and execute on a per-page basis. These permissions can be set up for a particular system (for example, program code might be execute-only, data structures may be mapped as read/write/no-execute) and can also be changed by the operating system based on application requests and operating system policies.

6.5.1 MMU Programming Model

This figure shows the registers that support the MMU.

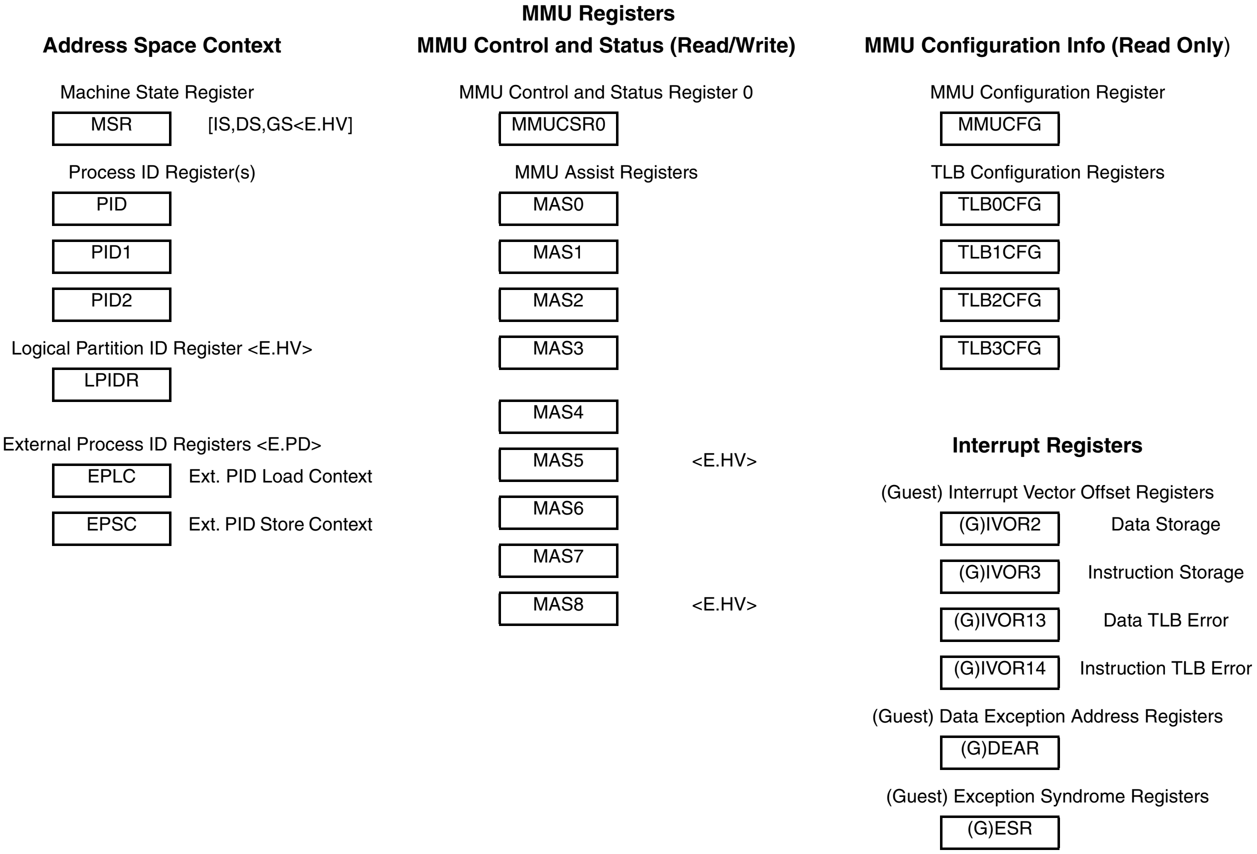


Figure 6-5. MMU Registers

These registers function as follows:

- Machine state register (MSR). The MSR contains instruction and data space bits (MSR[DS,IS]), which are used to defined the address space and are part of the virtual address in address translation. The guest state bit (MSR[GS]), is used to indicate the processor is in guest state and is also part of the virtual address. <E.HV>
- Process ID registers (PIDs). The PIDs each contain a multi-bit value which further identifies the address space. The PIDs are part of the virtual address. Values in PID registers are used by the operating system to assign virtual address spaces. Note that individual processors may not implement all 14 bits of the PID registers. Early versions of the architecture allowed for more than one PID register to be implemented. Newer processors only implement one PID register. The number of PIDs implemented is indicated by the value of MMUCFG[NPIDS].
- Logical partition ID register (LPIDR). <E.HV>
  The LPIDR contains a logical partition ID value. The LPIDR identifies the partition which is in execution. The LPIDR is used by the hypervisor to manage the address spaces between logical partitions in a manner similar to how an operating system uses PID registers to manage address spaces between processes. The LIPDR is part of the virtual address.
- External PID registers (EPLC, EPSC). <E.PD>
  External PID registers are used by the operating system and hypervisor to substitute different values for the virtual address space when performing external PID instructions. This allows the operating system or hypervisor to perform storage operations to a different virtual address space from the address space in which instructions are executing.
- MMU control and status register 0 (MMUCSR0). MMUCSR0 is used for general control of the MMU. It provides the following fields:
  - TLBn array page size (TLB*n*_PS). Specifies the page size for TLBn array if the array supports multiple fixed page sizes.
  - TLB invalidate all bit for the TLB*n* array (TLB*n*_FI). Allows the specified TLB array to have all its entries not protected with the IPROT entry invalidated
- MMU assist registers (MAS*n*). The MMU uses MAS registers configure and manage TLBs. MAS register functionality is summarized in Section 3.12.6, "MMU Assist Registers (MASn)." Section 6.5.8, "MMU Exception Handling," Section 6.5.8.3, "MAS Register Updates for Exceptions, tlbsx, and tlbre," and Section 6.5.3.4, "TLB Management using the MAS Registers," provide extensive information about how the MAS registers are used.
- MMU configuration register (MMUCFG). Gives configuration information about the implementation's MMU, including the following:
  - Number of bits in the LPID register supported by the implementation <E.HV>
  - Number of bits in a physical address supported by the implementation
  - Number of PID registers
  - Number of bits in a PID register supported by the implementation
  - Number of TLB arrays
  - MMU architecture version number
- TLB configuration registers (TLB*n*CFG), provide information about each specific TLB that is visible to the programming model. For example, TLB0CFG corresponds to TLB0.
  - Number of ways of associativity of TLB*n*

  - — Number of entries in TLB*n*
  - — Invalidate protect capability of TLB*n*
  - — Other TLB*n* characteristics:
    - – Minimum page size of TLB*n*
    - – Maximum page size of TLB*n*
    - – Page size availability of TLB*n*
- (Guest) Interrupt vector offset registers. (G)IVOR2 provides the vector offset for data storage interrupts.(G)IVOR3 provides the vector offset for instruction storage interrupts.(G)IVOR13 provides the vector offset for data TLB error interrupts. (G)IVOR14 provides the vector offset for instruction TLB error interrupts.
- IVOR42 provides the vector for the LRAT error interrupt. <E.HV.LRAT>
- (Guest) Data exception address registers ((G)DEAR). (G)DEAR provides the effective address that was use din a data access when a data storage interrupt or data TLB error interrupt occurs.
- (Guest) Exception syndrome registers ((G)ESR). (G)ESR provides information about errors which occur from a variety interrupts. For MMU related errors, it contains bits which identify whether the operation was a load or a store.

6.5.2 Virtual Address (VA)

As shown in Figure 6-6, the virtual address space is composed of the following:

- the effective address (EA), 32 bits in 32-bit mode and 64 bits in 64-bit mode
- the address space (AS), 1 bit taken from the MSR[IS], MSR[DS], EPLC[EAS] <E.PD>, or EPSC[EAS] <E.PD> bit depending on the type of access (instruction, non-external PID data, external PID load, or external PID store)
- the PID value, up to 14 bits from the PID, PID1, PID2, EPLC[EPID] <E.PD>, or EPSC[EPID] <E.PD> register depending on the type of access (instruction or non-external PID data, external PID load, or external PID store)
- the logical partition ID value (LPID value), up to 12 bits from the LPIDR, EPLC[ELPID] <E.PD>, or EPSC[ELPID] <E.PD> register depending on the type of access (instruction or non-external PID data, external PID load, or external PID store) <E.HV>
- the guest state value (GS), 1 bit taken from the MSR[GS], EPLC[EGS] <E.PD>, or EPSC[EGS] <E.PD> bit depending on the type of access (instruction or non-external PID data, external PID load, or external PID store) <E.HV>

The AS value, PID value, LPID value, and GS value determine the address space. These values are managed by system software. When instructions are fetched or instructions execute and access storage, they produce EAs that are combined with the address space and produce a virtual address. System software uses the address space components to provide isolated and protected ranges of addresses that a program may access. The total number of address spaces is $2^{1+1+\text{PID bits}+\text{LPID bits}}$ for implementations that support category Embedded.Hypervisor and $2^{1+\text{PID bits}}$ for implementations that do not support category Embedded.Hypervisor.

The following subsections describe how each of these values is selected as part of the virtual address based on the type of access.

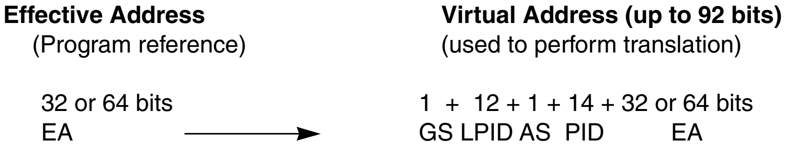

Effective Address
(Program reference)
32 or 64 bits
EA
Virtual Address (up to 92 bits)
(used to perform translation)
1 + 12 + 1 + 14 + 32 or 64 bits
GS LPID AS PID EA

Figure 6-6. Virtual Address Space

6.5.2.1 Address Space (AS) Value

The architecture defines two different virtual address spaces based on the setting of the MSR[IS] or MSR[DS] bit. These can be called address space 0 and address space 1. The 2 different virtual address spaces can be used to have completely separated address spaces for non-privileged and privileged software. This is possible because when an interrupt occurs, the MSR[IS] and MSR[DS] bits are always set to 0. Since an interrupt is required to transition from non-privileged to privileged modes, this provides an automatic and simple means of separating the address spaces. Thus, in general address space 0 can be used for privileged software and address space 1 can be used for non-privileged software. However, the usage model is not enforced by the processor (except that it always sets MSR[IS] and MSR[DS] to 0 on an interrupt), and software can be designed such that both privileged and non-privileged software share address space 0 and memory pages are appropriately protected. The sharing of address space between privileged can limit the amount of effective address space that a process can reference since it must share some of its address space with the privileged software. This can be particularly limiting for 32-bit implementations since EAs are limited to 32 bits.

The address space (AS) bit of the virtual address is formed based on the following criteria:

- If the access is an instruction fetch generated by sequential instruction fetches or due to a change in program flow (from a branch or an interrupt), the AS bit in the virtual address is taken from the MSR[IS] bit.
- If the access is not due to an instruction fetch (that is, is a data or cache management access), and the instruction which is attempting to perform the access is not an external PID instruction, the AS bit in the virtual address is taken from MSR[DS].
- If the access is not due to an instruction fetch (that is, it is a data or cache management access), and the instruction which is attempting to perform the access is an external PID instruction, the AS bit in the virtual address is taken from EPLC[EAS] if the instruction is considered a load or EPSC[EAS] if the instruction is considered a store. <E.PD>

The AS value of a virtual address is compared to the TS field in TLB entries to determine a matching translation.

Changing a source of the AS bit (MSR[IS], MSR[DS], EPLC[EAS], or EPSC[EAS]) is considered a context-altering operation and requires a context synchronizing operation to ensure that the results of the changes are visible in the current context.

NOTE: Software Considerations

When changing any of the register fields associated with forming virtual addresses from effective addresses, software must perform a context synchronizing operation to ensure that the results are visible to the programming model. Since the change can occur at an time after the register field is changed up until the context synchronizing operation is performed, it is unwise to change state that is associated with the current execution stream (particularly state which affects the virtual address of instruction fetches) by using **mtmsr**. This applies to MSR[IS], MSR[GS], and MSR[CM] (although MSR[CM] is not part of the virtual address is does control whether 32 or 64 bits are used from the effective address).

If such a change is made with **mtmsr**, software must ensure that all permutations of these bits have valid translations which contain the same permissions and translate to the same physical address. This avoids an implicit branch which is not supported by the architecture. For this reason, **mtmsr** should not be used to change these bits, but a return from interrupt instruction should be used instead since it changes the virtual address state and is context synchronizing.

NOTE: Software Considerations

If software uses disjoint address spaces for privileged and non-privileged software, external PID instructions can be used by privileged software to perform storage accesses to the non-privileged address space.

6.5.2.2 Process ID (PID) Value

A unique PID value is assigned by the operating system when executing a particular task or process. This identifies a particular address space for which EAs produced by the task or process will be part of. When an instruction fetch occurs, the PID value is taken from the PID register (see Section 3.12.2, “Process ID Register (PID)”) and is used to form the VA. When a data storage access occurs, the PID value is also taken from the PID register unless the instruction is an external PID instruction. If the data storage access is an external PID operation, the PID value is taken from EPLC[EPID] for loads or EPSC[EPID] for stores (see Section 3.12.7.1, “External PID Load Context Register (EPLC) <E.PD>” and Section 3.12.7.2, “External PID Store Context (EPSC) Register <E.PD>”).

The PID value 0, is a special value. When a 0 PID value is present in the TID field of a TLB entry, that TLB entry will match all PID values of a virtual address. In general, software will not store a 0 PID value in the PID register or EPLC[EPID] <E.PD>, EPSC[EPID] <E.PD> fields. TLB entries with 0 PID values in the TID field are generally used to map pages that are considered to be global among many address spaces. The operating system will most likely do this to avoid having to provide replicated TLB entries that map its address space when AS = 0 for every PID value.

The PID value of a virtual address is compared to the TID field in TLB entries to determine a matching translation.

Changing PID values in the PID register or in the EPLC[EPID] <E.PD>, EPSC[EPID] <E.PD> fields is a context-altering operation and requires a context synchronizing operation to ensure that the results of the changes are visible in the current context.

Some implementations provide more than one PID register. For those implementations, each PID register forms an additional virtual address. The multiple virtual addresses are discussed in Section 6.5.4, “Address Translation.”

NOTE: Software Considerations

> In general, the operating system need not flush the TLB between context switches since each unique context will have a unique PID value associated with it and the TLB entries for the different contexts can co-exist in the TLB. When the operating system must associate a PID value to a new context, the operating system must ensure that all the existing mappings of this particular PID value are flushed from the TLBs. For example, assume the operating system had assigned a PID value of 50 to a process. That process terminates and another process is created. When that process is created, the operating system may want to associate the PID value of 50 with that process since that PID value is no longer associated with any context. When the PID value is associated with that context (and will be used in TLB entries for that process), the TLB must be flushed of all the old entries which used the PID value 50.
>
> If the number of PID values available ($2^{\text{PID bits-1}}$) is larger than the number of simultaneous active contexts, flushing will only occur when a new process is created. However, if there are more active contexts than there are PID values, the operating system will need to reassign PID values more frequently, which will require the TLB to be flushed more frequently, possibly affecting performance.

6.5.2.3 GS Value <E.HV>

The GS value is used to provide separate address spaces for hypervisor and guest software.

The GS bit of the virtual address is formed based on the following criteria:

- If the access is an instruction fetch generated by sequential instruction fetches or due to a change in program flow (from a branch or an interrupt) the GS bit in the virtual address is taken from the MSR[GS] bit.
- If the access is not due to an instruction fetch (that is, it is a data or cache management access), and the instruction which is attempting to perform the access is not an external PID instruction, the GS bit in the virtual address is taken from MSR[GS].
- If the access is not due to an instruction fetch (that is, it is a data or cache management access), and the instruction which is attempting to perform the access is an external PID instruction, the GS bit in the virtual address is taken from EPLC[EGS] if the instruction is considered a load or EPSC[EGS] if the instruction is considered a store. <E.PD>

The GS value of a virtual address is compared to the TGS field in TLB entries to determine a matching translation.

Changing a source of the GS bit (MSR[GS], EPLC[EGS] <E.PD>, or EPSC[EGS] <E.PD>) is considered a context-altering operation and requires a context synchronizing operation to ensure that the results of the changes are visible in the current context.

6.5.2.4 LPID Value <E.HV>

A unique LPID value is assigned by the hypervisor when executing a particular logical partition. This identifies a unique address space for EAs, PID values, AS values, and GS = 0 values produced by the logical partition. The hypervisor uses LPID values in a manner similar to how an operating system uses PID values. In this case the assignment is for a particular logical partition as opposed to a process. When an instruction fetch occurs, the LPID value is taken from the LPIDR register (see Section 3.12.1, “Logical Partition ID Register (LPIDR) <E.HV>”) and is used to form the VA. When a data storage access occurs, the LPID value is also taken from the LPIDR register unless the instruction is an external PID instruction. If the data storage access is an external PID operation, the LPID value is taken from EPLC[ELPID] for loads or EPSC[ELPID] for stores (see Section 3.12.7.1, “External PID Load Context Register (EPLC) <E.PD>” and Section 3.12.7.2, “External PID Store Context (EPSC) Register <E.PD>”).

The LPID value 0, is a special value. When a 0 LPID value is present in the TLPID field of a TLB entry, that TLB entry will match all LPID values of a virtual address. In general, software will not store a 0 LPID value in the LPIDR register or EPLC[ELPID] <E.PD>, EPSC[ELPID] <E.PD> fields. TLB entries with 0 LPID values in the TID field are generally used to map pages that are considered to be global among all logical partitions. The hypervisor will most likely do this to avoid having to provide replicated TLB entries that map its address space when AS = 0 for every LPID value.

The LPID value of a virtual address is compared to the TLPID field in TLB entries to determine a matching translation.

Changing LPID values in the LPIDR register or in the EPLC[ELPID] <E.PD>, EPSC[ELPID] <E.PD> fields is a context-altering operation and requires a context synchronizing operation to ensure that the results of the changes are visible in the current context.

6.5.3 TLB Concept

A TLB is defined as the hardware resource that contains address translations, implements protections, and maintains the memory and cache attributes for pages. TLBs are defined by the architecture to be unified between instruction and data accesses. That is, a separate set of TLB structures between instruction fetch and data access are not permitted. TLB organization (for example number of TLB array structures, associativity, and number of entries) is implementation dependent, but is reflected in the MMUCFG, TLB*n*CFG registers.

The architecture defines the following general attributes associated with a TLB entry:

- Guest space (TGS) <E.HV>
- Logical Process ID (TLPID) <E.HV>
- Address space (TS)

- Process ID (TID)
- Effective page number (EPN)
- Real page number (RPN)
- Permission bits (UR,UW,UX,SR,SW,SX)
- Page attributes (WIMGE, VLE <VLE>, ACM, VF <E.HV>, IPROT, V)
- Page size (TSIZE)
- User bits (U0,U1,U2,U3)

Some implementations may provide additional page attributes, but such attributes are implementation dependent.

TLBs are software managed for MMU V1. Managing TLBs in software allows many different operating systems to be ported quickly because memory management data structures may be in any form, and requiring specific porting only at points where hardware intervention is required (such as a TLB entry replacement operation).

TLB read, write, search, and invalidate instructions are defined as follows:

- TLB Write Entry—**tlbwe**
- TLB Read Entry—**tlbre**
- TLB Search**—tlbsx r**A,**r**B
- TLB Invalidate**—tlbivax r**A,**r**B
- TLB Invalidate Local—**tlbilx** T,**r**A,**r**B

In addition, the architecture defines the TLB Synchronize instruction (**tlbsync**), which provides ordering for the effects of all **tlbivax** instructions.

The TLB entry fields defined by the architecture are described in this table.

Table 6-4. TLB Fields

Field	Description
V	Valid 0 This TLB entry is not valid for translation. 1 This TLB entry is valid for translation.
TID	Translation ID. Identifies the process ID (PID) value for this TLB entry. Translation IDs are compared with PID values during translation to help select a TLB entry. If a TLB entry’s TID field is zero, it globally matches all PID values.
TLPID <E.HV>	Translation logical partition ID value. Identifies the logical partition ID (LPID) value for this TLB entry. Translation LPID values are compared with LPID values during translation to help select a TLB entry. If a TLB entry’s TLPID field is zero, it globally matches all LPID values.
TS	Translation space. Identifies which address space (AS) value is valid for this TLB entry. TS is compared with the AS value (from MSR[IS] for instruction accesses and MSR[DS] for non-external PID data accesses) for selecting a TLB entry during translation.
TGS <E.HV>	Translation guest state value. Identifies which guest state (GS) value is valid for this TLB entry. TGS is compared with the GS value (from MSR[GS] for instruction accesses and MSR[GS] for non-external PID data accesses) for selecting a TLB entry during translation.

Table 6-4. TLB Fields (continued)

Field	Description
TSIZE	Page size. Describes the size of the page. Not all implementations support varying page sizes. Some TLB arrays support a range of page sizes, while other TLB arrays only support a fixed size.
EPN	Effective page number. Describes the effective address of the start of the page. The number of bits that are valid (used in translation) depends on the size of the page. EPN is compared with the effective address being translated selecting a TLB entry.
WIMGE	Memory and cache access attributes. Describes the characteristics of memory accesses to the page and the subsequent treatment of that data with respect to the memory subsystem (caches and bus transactions). See Section 6.4.1, “Memory/Cache Access Attributes (WIMGE Bits).” W Write-through required (1 = write-through required) I Caching inhibited (1 = page is caching-inhibited) M Memory coherence required (1 = page is memory-coherence required) G Guarded (1 = page is guarded) E Endianness (0 = page is big endian, 1 = page is little endian) Note: Little-endian is true little endian, which differs slightly from the little endian capability defined by the original PowerPC architecture.
IPROT	Invalidate protect. If set, this entry is protected against invalidation due to **tlbivax**, **tlbilx** <E.HV>, or an invalidate all operation from writing MMUCSR0. Not all TLB arrays support this attribute.
VLE <VLE>	VLE mode. Identifies pages which contain instructions from the VLE instruction set.
VF <E.HV>	Virtualization fault. Identifies a page which will always take a data storage interrupt directed to the hypervisor, regardless of the setting of any other page attributes. Set by the hypervisor for pages associated with a device for which the hypervisor is providing a “virtual” device through emulation.
ACM	Alternate coherency mode. The ACM attribute allows an implementation to employ more than a single coherency method. If the M attribute is not also set for this TLB entry, then the ACM attribute is ignored. This field is implementation specific.
RPN	Real page number. Describes the physical starting address of the page. The real page number replaces the effective page address resulting in a translated address. The number of bits used in translation depend on the size of the page (that is, the same number of bits as the EPN for a given page translation).
UX	User execute permission 0 Instructions may not be executed out of this page if MSR[PR] = 1 (user mode). When a user program attempts to execute instructions out of this page, an instruction storage interrupt occurs. 1 Instructions may be executed out of this page if MSR[PR] = 1 (user mode).
SX	Supervisor execute permission 0 Instructions may not be executed out of this page if MSR[PR] = 0 (supervisor mode). When a supervisor program attempts to execute instructions out of this page, an instruction storage interrupt occurs. 1 Instructions may be executed out of this page when the processor if MSR[PR] = 0 (supervisor mode).
UW	User write permission 0 Stores may not be performed to this page if MSR[PR] = 1 (user mode). If a user program attempts to store instructions to this page, a data storage interrupt occurs. 1 Stores may be performed to this page if MSR[PR] = 1 (user mode).

Table 6-4. TLB Fields (continued)

Field	Description
SW	Supervisor write permission 0 Stores may not be performed to this page when the processor if MSR[PR] = 0 (supervisor mode). If a supervisor program attempts to store to this page, a data storage interrupt occurs. 1 Stores may be performed to this page if MSR[PR] = 0 (supervisor mode).
UR	User read permission 0 Loads may not be performed from this page if MSR[PR] = 1 (user mode). If a user program attempts to load from this page, a data storage interrupt occurs. 1 Loads may be performed from this page if MSR[PR] = 1 (user mode).
SR	Supervisor read permission 0 Loads may not be performed from this page if MSR[PR] = 0 (supervisor mode). If a supervisor program attempts to load from this page, a data storage occurs. 1 Loads may be performed from this page if MSR[PR] = 0 (supervisor mode).
U0–U3	User page attribute bits . In general, These are extra memory bits associated with a TLB entry to be used by system software.

The Power ISA includes MMU features defined for Freescale embedded processors. This establishes a MMU model that is consistent across all Freescale Power ISA devices, and because the programming model is less likely to change, it reduces the software efforts required in porting to a new processor. Thus, the standard defines configuration information for features such as TLBs, caches, and other entities that have standard forms, but differing attributes (like cache sizes and associativity) such that a single software implementation can be created that works efficiently for all implementations of a class.

The architecture defines the following functions and structures visible to the execution model of the processor:

- The TLB, from a programming point of view, consists of zero or more TLB arrays, each of which may have differing characteristics.
- The effective-to-physical address translation mechanism
- Methods and effects of changing and manipulating TLB arrays
- Configuration information available to the operating system that describes the structure and form of the TLB arrays and translation mechanism

To assist or accelerate translation, implementations may employ other TLB structures not visible to the programming model. Methods for using such structures are not defined explicitly, but they may be considered at the operating-system level because they may affect an implementation's performance.

6.5.3.1 TLB Entries

The TLB is subdivided into zero or more TLB arrays. Each array must contain TLB entries that share the same characteristics. Each array contains one or more TLB entries. Each entry has specific fields that correspond to fields in the MMU assist registers, described in Section 3.12.6, “MMU Assist Registers (MASn).” Note that in each implementation, architected fields may have restrictions (such as implementing fewer bits) or enhancements.

6.5.3.2 TLB Entry Page Size

Each TLB entry has a page size associated with it. This defines the how many bytes this particular TLB entry supports. The possible sizes that are supported by an implementation depends on the TLB array and is reflected in the associated TLB*n*CFG register. A TLB entry page size is established by writing a TLB entry with MAS1[TSIZE] set to a value that represents a page size. Page sizes are defined as power of 4 KB values, thus the size of a page is 4^{TSIZE} KB . Certain bits in the EPN and RPN fields associated with page offsets should be 0 based on page size. Table 6-5 shows the valid page sizes defined by the architecture.

NOTE: Software Considerations

EIS does not support 1KB or 2KB page sizes.

Table 6-5. TLB Page Sizes

TSIZE (TLB_{SIZE})	Page Size	EPN and RPN Bits Required to be Zero	Real Address after Translation
0b0001	4KB	none	$RPN_{0:51}$ ll $EA_{52:63}$
0b0010	16KB	50:51	$RPN_{0:49}$ ll $EA_{50:63}$
0b0011	64KB	48:51	$RPN_{0:47}$ ll $EA_{48:63}$
0b0100	256KB	46:51	$RPN_{0:45}$ ll $EA_{46:63}$
0b0101	1MB	44:51	$RPN_{0:43}$ ll $EA_{44:63}$
0b0110	4MB	42:51	$RPN_{0:41}$ ll $EA_{42:63}$
0b0111	16MB	40:51	$RPN_{0:39}$ ll $EA_{40:63}$
0b1000	64MB	38:51	$RPN_{0:37}$ ll $EA_{38:63}$
0b1001	256MB	36:51	$RPN_{0:35}$ ll $EA_{36:63}$
0b1010	1GB	34:51	$RPN_{0:33}$ ll $EA_{34:63}$
0b1011	4GB	32:51	$RPN_{0:31}$ ll $EA_{32:63}$
0b1100	16GB	30:51	$RPN_{0:29}$ ll $EA_{30:63}$
0b1101	64GB	28:51	$RPN_{0:27}$ ll $EA_{28:63}$
0b1110	256GB	26:51	$RPN_{0:25}$ ll $EA_{26:63}$
0b1111	1TB	24:51	$RPN_{0:23}$ ll $EA_{24:63}$

6.5.3.3 Reading and Writing TLB Entries

TLB entries written by software are updated by executing **tlbwe** instructions. At the time of **tlbwe** execution, the MMU assist registers (MAS0–MAS8), are used to select a specific TLB entry. The MAS registers also contain the information that is written to the selected entry, such that they serve as the ports into the TLBs, as shown in Figure 6-7. The contents of the MAS registers are described in Section 3.12.6, “MMU Assist Registers (MASn).”

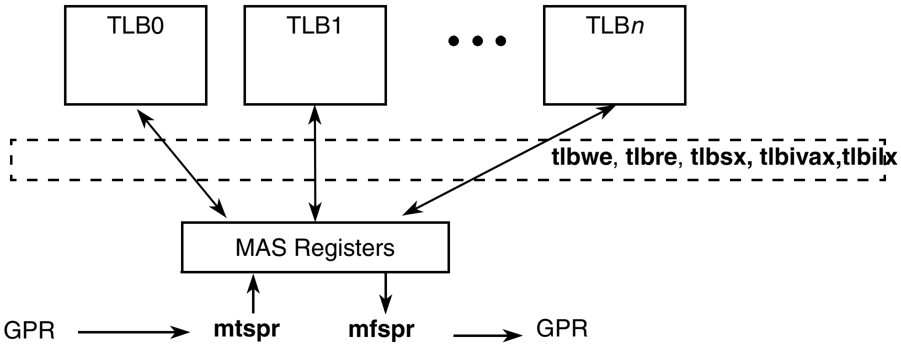


Figure 6-7. TLBs Accessed Through MAS Registers and TLB Instructions

Similarly, TLB entries are read by executing **tlbre** instructions. At the time of **tlbre** execution, the MAS registers are used to select a specific TLB entry and upon completion of the **tlbre** instruction, the MAS registers contain the contents of the indexed TLB entry. To read or write TLB entries, the MAS registers are first loaded by system software using **mtspr** instructions and then the desired **tlbre** or **tlbwe** instructions must be executed.

6.5.3.4 TLB Management using the MAS Registers

This section describes how TLB instructions and MAS registers are used to configure and update the TLBs.

6.5.3.4.1 MAS Registers

TLB entries are managed by software using a set of SPRs called the MMU assist (MAS) registers. MAS registers are used to move data between software and the TLB entries. MAS registers are also used to identify TLB entries and provide default values when translation or protection faults occur. There are 8 MAS registers that are more fully described in Section 3.12.6, “MMU Assist Registers (MASn).”

<64-bit>:
64-bit implementations also provide access to MAS register “pairs” which allows 2 MAS registers to be accessed with a single mfspr/mtspr instruction. See Section 3.12.6.10, “64-bit Access to MAS Register Pairs <64-bit, E.HV>.”

<Embedded.Hypervisor>:
MAS5 and MAS8 are only accessible by the hypervisor. MAS5 contains GS and LPID values for searching TLB entries and MAS8 contains entries that identify the LPID value and GS value associated with a TLB entry. Hypervisor software should set these registers to values appropriate to the partition which is being executed.

6.5.3.5 Reading TLB Entries

TLB entries can be read by executing **tlbre** instructions. At the time of **tlbre** execution, the MAS registers are used to select a specific TLB entry and upon completion of the **tlbre**, the MAS registers contain the contents of the indexed TLB entry.

Selection of the TLB entry to read is performed by setting MAS0[TLBSEL], MAS0[ESEL] and MAS2[EPN] to indicate the entry to read. MAS0[TLBSEL] selects which TLB array the entry should be read from (0 to 3) and MAS2[EPN] selects the set of entries from which MAS0[ESEL] selects an entry. For fully associative TLBs, MAS2[EPN] is not required since the value in MAS0[ESEL] fully identifies the TLB entry. Valid values for MAS0[ESEL] are from 0 to associativity - 1.

When reading a TLB entry, MAS0[ATSEL] must be set to 0. <E.HV.LRAT>

The selected TLB entry is then used to update the following fields of the MAS registers: V, IPROT, TLPID <E.HV>, TID, TS, TGS <E.HV>, TSIZE, EPN, ACM, VLE <VLE>, WIMGE, VF <E.HV>, RPNU, RPNL, U0—U3, and permissions. If the TLB array supports NV, it is used to update the NV field in the MAS registers, otherwise the contents of NV field are undefined. The update of MAS registers as a result of a **tlbre** instruction is summarized in Table 6-11.

No operands are given for the **tlbre** instruction.

Specifying invalid values for MAS0[TLBSEL] and MAS0[ESEL] produce boundedly undefined results.

tlbre is hypervisor privileged if category Embedded.Hypervisor is implemented, otherwise it is supervisor privileged.

NOTE: Virtualization

Hypervisor software must emulate guest executions of **tlbre** and provide a mapping of the RPN from the TLB entry to a logical address.

NOTE: Architecture

tlbre is hypervisor privileged because allowing read access to TLB entries would allow the guest to see the TLB entries of other guests as well as their own true real addresses. The embedded hypervisor privilege interrupt allows the hypervisor to intervene and perform the real-to-logical address translation of the RPN field.

6.5.3.6 Writing TLB Entries

TLB entries can be written by executing **tlbwe** instructions. At the time of **tlbwe** execution, the MAS registers are used to select a specific TLB entry and contain the contents to be written to the selected TLB entry. Upon completion of the **tlbwe** instruction, the TLB entry contents of the MAS registers are written to the selected TLB entry.

Selection of the TLB entry to write is performed by setting MAS0[TLBSEL], MAS0[ESEL] and MAS2[EPN] to indicate the entry to write. MAS0[TLBSEL] selects which TLB array the entry should be written to (0 to 3) and MAS2[EPN] selects the set of entries from which MAS0[ESEL] selects an entry. For fully associative TLBs, MAS2[EPN] is not used to identify a TLB entry since the value in MAS0[ESEL] fully identifies the TLB entry. Valid values for MAS0[ESEL] are from 0 to associativity minus 1.

When writing a TLB entry, MAS0[ATSEL] must be set to 0. <E.HV.LRAT>

The selected TLB entry is then written with following fields of the MAS registers: V, IPROT, TLPID <E.HV>, TID, TS, TGS <E.HV>, TSIZE, EPN, ACM, VLE <VLE>, WIMGE, VF <E.HV>, RPNU, RPNL, U0—U3, and permissions. If the TLB array supports NV, it is written with the NV value.

The effects of updating the TLB entry are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation. Writing a TLB entry that is used by the programming model prior to a context synchronizing operation produces boundedly undefined behavior.

No operands are given for the **tlbwe** instruction.

Specifying invalid values for MAS0[TLBSEL] and MAS0[ESEL] produce boundedly undefined results.

The privilege level of executing **tlbwe** is based on several factors, as shown in this table.

Table 6-6. Privilege Level of Executing tlbwe

E.HV Implemented	E.HV.LRAT Implemented	TLBnCFG[LRAT] = 1	MSR[GS]	MSR[PR]	EPCR[DGTMI]	Result
x[1]	x	x	x	1	n/a	**privilege exception**
no	n/a	n/a	n/a	0	n/a	**write entry**
yes	x	x	0	0	x	**write entry**
yes	x	x	1	0	1	**embedded hypervisor privilege exception**
yes	no	n/a	1	0	0	**embedded hypervisor privilege exception**
yes	yes	no	1	0	0	**embedded hypervisor privilege exception**
yes	yes	yes	1	0	0	**write entry after translating through LRAT, otherwise LRAT error exception**

[1] x denotes a “don’t care” condition.

NOTE: Virtualization

Hypervisor software must emulate guest executions of **tlbwe** and provide a mapping of the logical address in the RPN field in the MAS registers to a real address.

NOTE: Software Considerations

Writing TLB entries should be followed by a context synchronizing instruction (such as an **isync** or an **rfi**) before the new entries are to be used by the programming model.

6.5.3.7 Searching TLB Entries

Software may search the MMU by using the TLB Search instruction, **tlbsx**, which use virtual address identifiers from the MAS registers instead of the normal translation sources. This allows software to search address spaces that differ from the current address space. This is useful for handling TLB faults.

To properly execute a search for a TLB entry, software loads MAS registers with virtual address values to search for as follows:

- PID value - MAS6[SPID]
- AS value - MAS6[SAS]

  If category Embedded.Hypervisor is implemented the following are also used:
- LPID value - MAS5[SLPID]
- GS value - MAS5[SGS]

Software then executes a **tlbsx** instruction. The search performs the same TA to VA comparison described in Section 6.5.4, "Address Translation," except that the virtual address identifier values are taken from the MAS registers. If a matching, valid TLB entry is found, the MAS registers are loaded with the information from that TLB entry as if the TLB entry was read from a **tlbre** instruction. Software can examine the MAS1[V] bit to determine if the search was successful. Successful searches cause the valid bit to be set. Unsuccessful searches cause the valid bit to be clear. Table 6-11 summarizes the update of MAS registers as a result of a **tlbsx** instruction.

The preferred form of the **tlbsx** is **r**A = 0. Some implementations may take an unimplemented instruction exception or an illegal instruction exception if **r**A != 0.

6.5.3.8 Invalidating TLB Entries

This section describes how TLB entries can be invalidated, by any of the following methods:

- As the result of a **tlbwe** instruction that clears MAS1[V] in the entry. **tlbwe** instructions that use hardware entry select (HES) to determine which TLB entry is written do not require invalidations of the written entry, even if the entry is written with MAS1[V] = 0. <E.PT>
- As a result of a flash invalidate operation initiated by writing to bits in MMUCSR0 (see Section 3.12.3, "MMU Control and Status Register 0 (MMUCSR0)").
- As a result of a **tlbivax** instruction or from a received broadcast invalidation resulting from a **tlbivax** on another processor.
- As a result of a **tlbilx** instruction.

Clearing MAS1[V] and writing a specified TLB entry invalidates a single TLB entry. The other methods invalidate multiple TLB entries, possibly including some TLB entries that match part of the invalidation criteria. This is referred to as generous invalidation. The scope of how many TLB entries are cleared beyond the required ones when a generous invalidate occurs is implementation dependent. TLB arrays can provide the invalidate protect bit (IPROT), which can be set to prevent a TLB entry from being invalidated as the result of any invalidation, except for clearing the MAS1[V] bit and writing the entry, as described in Section 6.5.3.8.5, "Generous Invalidates and Invalidation Protection."

NOTE: Software Considerations

> Software must assume that any TLB entry not protected by IPROT may be invalidated whenever an invalidation is performed (except when writing the V bit in the TLB entry to 0).

6.5.3.8.1 Invalidation of a Single TLB Entry by Clearing MAS1[V]

A single TLB entry can be invalidated by executing a **tlbwe** instruction when MAS1[V] = 0, and the entry is specified by MAS0[TLBSEL], MAS0[ESEL] and MAS2[EPN]. This clears the valid bit for that TLB entry. This is the only method that can be used to invalidate a TLB entry whose IPROT bit is set.

6.5.3.8.2 Invalidations Using tlbivax

The **tlbivax** instruction provides a virtual address (EA, MAS6[SPID], MAS6[SAS], MAS5[SLPID] <E.HV>, and MAS5[SGS] <E.HV>) as a target for invalidation. The virtual address to select a TLB entry to invalidate is specified as follows:

- EA[0–51] specifies the effective page number and is compared to the EPN field in the TLB entry. The page size of the TLB entry masks the low order bits in the comparison.
- MAS6[SPID] specifies the effective page number and is compared to the TID field in the TLB entry.
- MAS6[SAS] specifies the effective page number and is compared to the TS field in the TLB entry.
- MAS5[SLPID] specifies the effective page number and is compared to the TLPID field in the TLB entry. <E.HV>
- MAS5[SGS] specifies the effective page number and is compared to the TGS field in the TLB entry. <E.HV>

The comparison is performed only for valid TLB entries in the specified TLB array that do not have the IPROT attribute set (if supported by the TLB array). The EA specified by the **r**A and **r**B operands in the **tlbivax** instruction contains fields in the lower order bits to augment the invalidation to specific TLB arrays and to flash invalidate those arrays. Note that **tlbivax** instructions do not invalidate any entry that has IPROT = 1, unless the specified TLB array does not support the IPROT attribute. The encoding of the EA used by **tlbivax** is shown in Figure 6-8 and Table 6-7.

The tlbivax invalidation is broadcast to each processor in the coherence domain, performing the invalidations there as well. To synchronize the effects of the **tlbivax** globally, the **tlbsync** instruction is used to ensure that the invalidation is performed and that all memory accesses which may have translated prior to the invalidation have been performed.

tlbivax is hypervisor privileged if category Embedded.Hypervisor is implemented, otherwise it is supervisor privileged.

This figure shows the EA format for **tlbivax**.

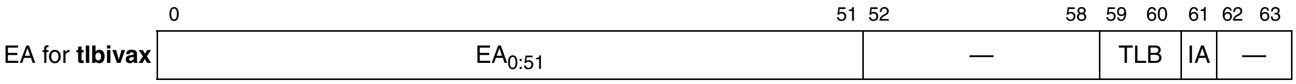


Figure 6-8. EA Format for tlbivax

This table describes EA fields for **tlbivax**.

Table 6-7. Fields for EA Format of tlbivax

Field	Name	Comments or Function when Set
0–51	$EA_{0:51}$	The upper bits of the address to invalidate.
52–58	—	Reserved, should be cleared.
59–60	TLB	Selects TLB array for invalidation. 00 TLB0 01 TLB1 10 TLB2 11 TLB3
61	IA	Invalidate all entries in selected TLB array.
62–63	—	Reserved, should be cleared.

NOTE: Software Considerations

To ensure a TLB entry that is not protected by IPROT is invalidated if software does not know which TLB array the entry is in, software should issue a **tlbivax** instruction targeting each TLB in the implementation with the EA to be invalidated.

NOTE: Software Considerations

The preferred form of **tlbivax** contains the entire EA in **r**B and zero in **r**A. Some implementations may take an unimplemented instruction exception if **r**A is non-zero.

NOTE: Software Considerations

Software should avoid using the IA (invalidate all) function by keeping bit 61 clear because this capability is likely to be removed from future versions of the architecture.

NOTE: Software Considerations

For processors that implement category Embedded.Hypervisor, the preferred method of performing invalidations is **tlbilx**.

6.5.3.8.3 Invalidations Using tlbilx <E.HV>

The **tlbilx** instruction provides a virtual address or a partial virtual address as a target for invalidation. Unlike **tlbivax**, **tlbilx** only guarantees that invalidations are performed on the processor executing **tlbilx** (and are considered local). It is implementation dependent whether any invalidations occur on other processors. Invalidations are performed based on the T field specified in the **tlbilx** instruction as follows:

- If T= 0, all TLB entries for which the TLPID field match MAS5[SLPID] are invalidated. This allows all TLB entries belonging to a particular logical partition to be invalidated.

- If T = 1, all TLB entries for which the TLPID field match MAS5[SLPID] and the TID field match MAS6[SPID] are invalidated. This allows all TLB entries belonging to a particular process ID value to be invalidated.
- If T = 3, all TLB entries for which the TLPID field match MAS5[SLPID], the TID field match MAS6[SPID], the TGS field match MAS5[SGS], the TS field match MAS6[SAS], and the EPN field matches the EA from **r**A and **r**B are invalidated. This allows a particular virtual address to be invalidated.

The comparison is performed only for valid TLB entries in the specified TLB array that do not have the IPROT attribute set (if supported by the TLB array). Note that **tlbilx** instructions do not invalidate any entry that has IPROT = 1, unless the specified TLB array does not support the IPROT attribute.

tlbilx is guest supervisor privileged unless EPCR[DGTMI] is set in which case it is hypervisor privileged.

NOTE: Software Considerations

Note that **tlbilx** only guarantees invalidations on the local processor. Software must arrange to execute the appropriate **tlbilx** on all processors of interest in order to perform the invalidation globally. This will generally be accomplished using an IPC type interrupt such as a doorbell generated by a **msgsnd** instruction.

6.5.3.8.4 Invalidate all Using MMUCSR0

All entries in a TLB array are invalidated by setting the corresponding invalidate all bit (MMUCSR0[TLB*n*_FI]). MMUCSR0 is described in Section 3.12.3, “MMU Control and Status Register 0 (MMUCSR0).” Invalidation is complete when the processor core clears the corresponding invalidate all bit. Writing a 0 to an MMUCSR0 invalidate all bit has no effect. Note that such invalidations do not invalidate any entry that has IPROT = 1.

6.5.3.8.5 Generous Invalidates and Invalidation Protection

Generous invalidates can occur because parts of the translation mechanism may not be fully specified to the hardware at invalidate time. So, in the case of a set-associative TLB array, it may be that all indexed entries (all ways) in the TLB array for a particular effective address are invalidated as a result of a **tlbivax**.

In SMP systems, the **tlbivax** generates an invalidation address that is broadcast to all processors in the system. and the hardware implementation may impose other caveats causing additional invalidations. The architecture assures that the intended TLB is invalidated, but does not guarantee it is the only one.

Protecting against generous invalidation is necessary because certain effective memory regions must always be properly mapped. For example, the instruction memory region that serves as the exception handler for MMU faults must always be mapped, otherwise, an MMU exception cannot be handled because the first address of the handler causes another MMU exception.

To prevent generous invalidation, the invalidate protection (IPROT) bit can be set to protect the corresponding entry from any invalidation other than directly writing the V field to 0. TLB entries with the IPROT bit set can only be invalidated by explicitly writing the TLB entry and specifying a 0 for the valid bit.

Operating systems must use care when using protected TLB entries, particularly in SMP systems. An SMP system that contains protected (IPROT = 1) TLB entries on other processors requires a synchronization mechanism such as a cross-processor interrupt to ensure that each processor performs the required invalidation by clearing the valid bits of its own TLB entries (using **tlbwe**) in a synchronized way with respect to the other processors in the system.

NOTE: Software Considerations

> Note that not all TLB arrays in a given device implement the IPROT attribute. It is likely that implementations that are suitable for demand-page environments implement IPROT for only a single array, while not implementing it for other on-chip TLB arrays. See the user documentation.

6.5.4 Address Translation

The effective address (EA) is the untranslated address for a data or instruction fetch address that is calculated as a result of a load, store, or cache management instruction. The EA is combined with address space identifiers to form a virtual address (VA) . The VA contains these specific elements: EA, AS, PID, LPID <E.HV>, GS <E.HV> as described in Section 6.5.2, “Virtual Address (VA).”

A program references memory by using an effective address (EA) computed by the processor when it executes a load, store, cache management, branch or other control flow-control instruction, and when it fetches the next instruction. Because TID and TLPID <E.HV> values can contain the value 0 and thus match any LPID value or LPID value respectively, additional VAs are formed using the permutation of 0 values for these fields. For example, an implementation that does not implement category Embedded.Hypervisor and only implements one PID register (PID) would form the following virtual addresses for translation:

```
VA0 ← AS ||  0  || EA
VA1 ← AS || PID || EA
```

The value of 0 for a possible match with the TID field allows a TLB entry with a value of 0 in the TID field to match all PID values. Note that VAs may be non-unique. There may be duplicate VAs formed, however such duplicated VAs are ignored. In this particular case if the PID value from the PID register was equal to 0, VA0 and VA1 would be the same VA. In this case, only one of the VAs is considered for translation and the other is ignored.

Virtual addresses are formed differently based on whether the processor implements category Embedded.Hypervisor, category Embedded.External PID, or more than one PID register. When category Embedded.Hypervisor is implemented, non Embedded.External PID data accesses and all instruction fetches form the following virtual addresses:

```
VA0 ← GS || LPIDR || AS || PID || EA
VA1 ← GS ||   0   || AS || PID || EA
VA2 ← GS || LPIDR || AS ||  0  || EA
VA3 ← GS ||   0   || AS ||  0  || EA
```

These VAs are a result of the permutation of LPIDR, a zero value for LPIDR, PID, and a zero value for PID. The concept of multiple virtual addresses is a more formal way of specifying how translation is performed. In practice, the TLB entries for TLPID and TID simply perform a “don’t care” condition when comparing with LPIDR and PID if TLPID or TID contain zero values. Note also in the above example, that if GS and LPIDR are always 0 (that is, the operating system is running bare-metal and has no knowledge of hypervisor capabilities), that the VAs that are formed are the same VAs that would be formed if category Embedded.Hypervisor was not present.

<Embedded.External PID>:
When a category Embedded.External PID instruction makes a storage reference, all the parts of a virtual address except the EA are taken from the EPLC or EPSC register depending on whether the storage access is a load or a store. The virtual addresses that are formed are as follows:

(for External PID loads)

```
VA0 ← EPLC[EGS] || EPLC[ELPID] || EPLC[EAS] || EPLC[EPID] || EA
VA1 ← EPLC[EGS] ||      0      || EPLC[EAS] || EPLC[EPID] || EA
VA2 ← EPLC[EGS] || EPLC[ELPID] || EPLC[EAS] ||     0      || EA
VA3 ← EPLC[EGS] ||      0      || EPLC[EAS] ||     0      || EA
```

(for External PID stores)

```
VA0 ← EPSC[EGS] || EPSC[ELPID] || EPSC[EAS] || EPSC[EPID] || EA
VA1 ← EPSC[EGS] ||      0      || EPSC[EAS] || EPSC[EPID] || EA
VA2 ← EPSC[EGS] || EPSC[ELPID] || EPSC[EAS] ||     0      || EA
VA3 ← EPSC[EGS] ||      0      || EPSC[EAS] ||     0      || EA
```

Earlier versions of the architecture permitted multiple PID registers to be implemented. On implementations that provide multiple PID registers, more virtual addresses are formed using the permutation of PID0, PID1, .. PID*n* and the zero value for the PID value.

Each of the unique VAs are compared to all of the valid TLB entries by comparing specific fields of each TLB entry to each of the VAs. The fields of each valid (TLB[V] = 1) TLB entry are combined to form a set of matching TLB addresses (TAs):

```
TA ← TLB[TS] || TLB[TID] || TLB[EPN] || $^{12}0$
```

or if category Embedded.Hypervisor is implemented:

```
TA ← TLB[TGS] || TLB[TLPID] || TLB[TS] || TLB[TID] || TLB[EPN] || $^{12}0$
```

Each TA is compared to all VAs under a mask based on the page size (TLB[SIZE]) of the TLB entry. The mask of the comparison of the EA and EPN portions of the virtual and translation addresses is computed as follows:

```
    mask ← ~(1024 << (2 * TLB[SIZE])) - 1)
```

where the number of bits in the mask equals the number of bits in a TA (or VA). If a TA matches any VA, the TLB entry is said to match. Multiple matches are considered a serious programming error, and results are undefined. The recommended behavior is that a machine check interrupt be taken.

When a match occurs, the matching TLB is used for access control, storage attributes, and effective-to-physical address translation. Access control, storage attributes, and address translation are defined by the architecture.

6.5.4.1 Match Criteria for TLB Entries

TLB arrays contain TLB entries that are used to match any address presented for translation. All TLB entries for any given implementation are candidates for any given translation. The TLB itself is unordered with respect to the various elements used in address translations, and regardless of implementation, should be considered to perform the translation comparison with all entries in parallel.

There should be only one valid matching translation for a given TA and unique VA. It is considered a programming error for a TLB to contain multiple matching entries. The behavior of any such translation is undefined; the processor is may take a machine check interrupt or produce undefined results.

As shown in Figure 6-9, the following fields are compared in the TLB entries:

- V—The matching entry must have the V bit set.
- TGS <E.HV>—The guest space identifier used for translation. The MSR[GS] bit must match the TGS bit for a matching entry
- TLPID <E.HV>—The contents of the LPIDR register must match the TLPID field of a matching entry, or the TLPID field must be all zeros for a matching entry.
- TS—The address space identifier used for translation. The appropriate bit of MSR[IS] or MSR[DS] must match the TS bit for a matching entry.
- TID—The contents of the PID register must match the TID field of a matching entry, or the TID field must be all zeros for a matching entry.
- EPN—The appropriate number of bits (depending on the page size) of the effective address being translated is compared to the EPN field of the TLB entry.

This figure shows the translation match logic for the EA plus its attributes (collectively called the virtual address) and how it is compared with the corresponding fields in the TLB entries.

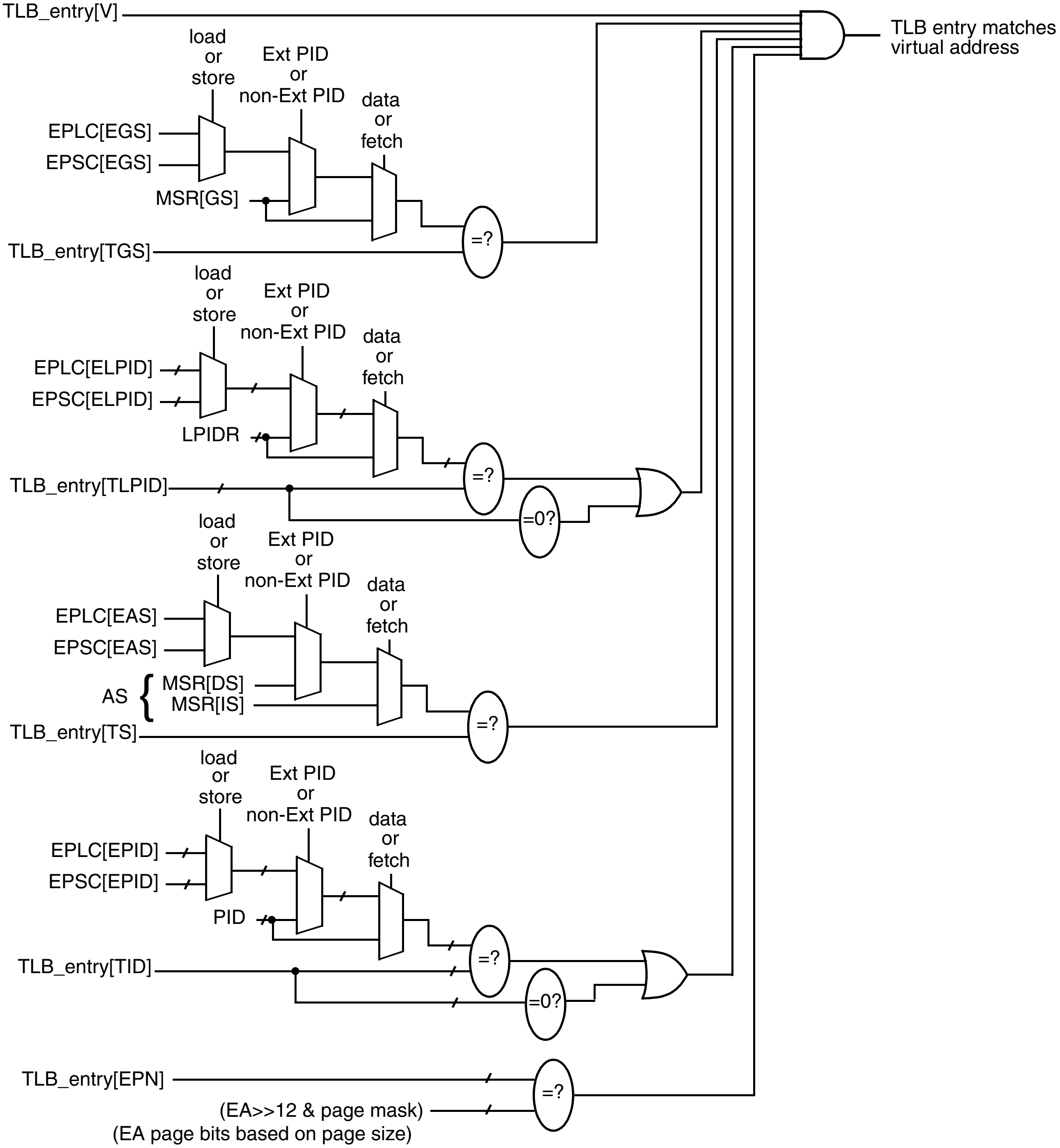


Figure 6-9. Virtual Address and TLB-Entry Comparison

6.5.4.2 Real Address Generation after a TLB Match

If a match occurs on all the fields listed in Section 6.5.4.1, “Match Criteria for TLB Entries,” the real address is formed by combining TLB[RPN] with the lower order offset bits of the EA based on the page size of the TLB entry (where ‘mask’ contains the same number of bits as a real address).

```
mask ←~(1024 << (2 * TLB[TSIZE])) - 1)
real_address ← ((TLB[RPN] << 12) & mask) | (EA & ~mask)
```

The real address is then used to access the memory subsystem using the TLB[ACM,VLE <VLE>,W,I,M,G,E] fields from the TLB entry to determine how the location should be accessed.

The generation of the real address occurs as shown in Figure 6-10. n denotes which bits are part of the page offset or the page number. n = 54 - (2 * TLB[TSIZE]).

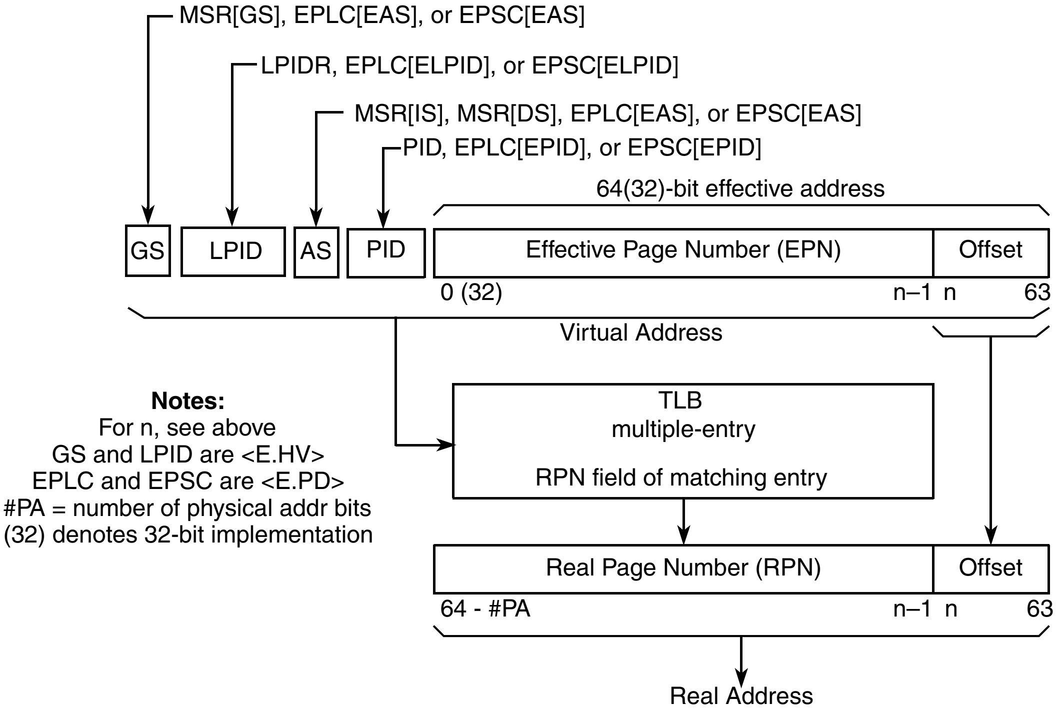

MSR[GS], EPLC[EAS], or EPSC[EAS]
LPIDR, EPLC[ELPID], or EPSC[ELPID]
MSR[IS], MSR[DS], EPLC[EAS], or EPSC[EAS]
PID, EPLC[EPID], or EPSC[EPID]
64(32)-bit effective address
GS
LPID
AS
PID
Effective Page Number (EPN)
Offset
0 (32)
n–1 n
63
Virtual Address
Notes:
For n, see above
GS and LPID are <E.HV>
EPLC and EPSC are <E.PD>
#PA = number of physical addr bits
(32) denotes 32-bit implementation
TLB
multiple-entry
RPN field of matching entry
Real Page Number (RPN)
Offset
64 - #PA
n–1 n
63
Real Address

Figure 6-10. Effective-to-Real Address Translation

6.5.4.3 Page Size and Effective Address Bits Compared

The page size defined for a TLB entry determines how many bits of the effective address are compared with the corresponding EPN field in the TLB entry as shown in Table 6-10.

6.5.4.4 Permission Attribute Comparison

As part of the translation process, the selected TLB entry provides the access permission bits (UX, SX, UW, SW, UR, SR), and memory/cache attributes (ACM, VLE<VLE>, W, I, M, G, and E) for the access. These bits specify whether an access is allowed and how it is to be performed.

If a matching TLB entry is identified, an architecturally defined access permission mechanism selectively grants shared access, grants execute access, grants read access, grants write access, and prohibits access to areas of memory based on a various criteria. TLB entry permission bits are as follows:

- SR—Supervisor read permission
- SW—Supervisor write permission
- SX—Supervisor execute permission
- UR—User read permission
- UW—User write permission
- UX—User execute permission

If the virtual address translation comparison with TLB entries succeeds, the permission bits for the matching entry are checked as shown in Figure 6-11. If the access is not allowed, the processor generates an instruction or data storage interrupt.

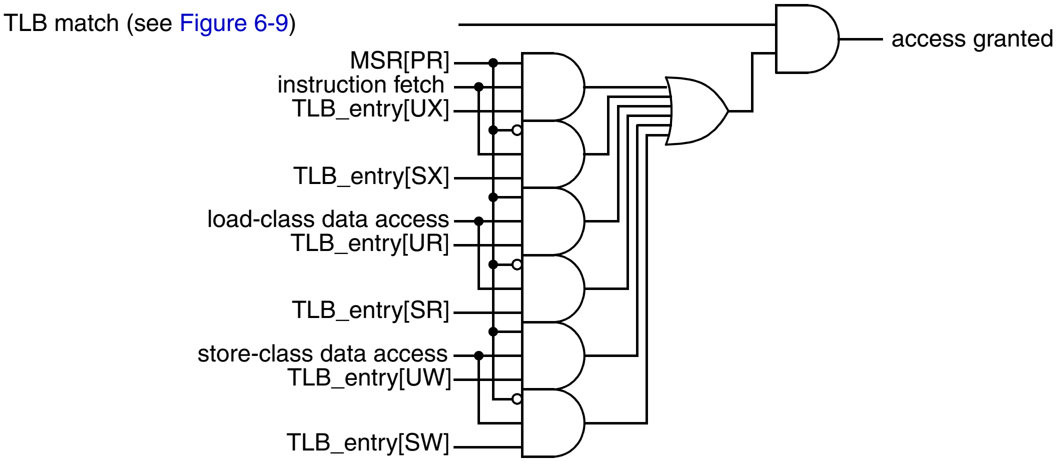


Figure 6-11. Granting of Access Permission

The permission attributes are defined in detail in Section 6.5.6, “Permission Attributes.”

6.5.4.5 Translation Algorithms

The following algorithm describes how translation operates:

```
ea ← effective address
if translation is an instruction address then
   as ← MSR_IS
   gs ← MSR_GS
   pid ← PID
   lpid ← LPIDR
else                    // data address translation
   if external PID instruction
```

```
          if external PID load instruction
              as ← $EPLC_{EAS}$
              gs ← $EPLC_{EGS}$
              pid ← $EPLC_{EPID}$
              lpid ← $EPLC_{ELPID}$
          else            // external PID store instruction
              as ← $EPLC_{EAS}$
              gs ← $EPLC_{EGS}$
              pid ← $EPLC_{EPID}$
              lpid ← $EPLC_{ELPID}$
      else                // normal load/store instruction
          as ← $MSR_{IS}$
          gs ← $MSR_{GS}$
          pid ← PID
          lpid ← LPIDR

   for all TLB entries
      if ! $TLB_V$ | (as != $TLB_{TS}$) | (gs != $TLB_{TGS}$) then
          next            // not valid or no addr space match
      if ($TLB_{TID}$ != 0) & ($TLB_{TID}$ != pid) then
          next            // no 0 for TID or no PID match
      if ($TLB_{TLPID}$ != 0) & ($TLB_{TLPID}$ != lpid) then
          next            // no 0 for TLPID or no LPID match
      mask ← ~((1024 << (2 * $TLB_{TSIZE}$)) - 1)
      if (ea & mask) != ($TLB_{EPN}$ << 12) then
          next            // no address match
      real address ← ($TLB_{RPN}$ << 12) | (ea & ~mask)
      end translation -- success
   endfor
   if translation is an instruction address then
      instruction TLB error interrupt
   else
      data TLB error interrupt
```

If translation is successful, the algorithm for the granting of permission is as follows:

```
   if $MSR_{PR}$ = 0 then
      x ← $TLB_{SX}$
      r ← $TLB_{SR}$
      w ← $TLB_{SW}$
   else
      x ← $TLB_{UX}$
      r ← $TLB_{UR}$
      w ← $TLB_{UW}$

   if instruction_fetch & x = 0 then
      instruction storage interrupt)
   else if load &         r = 0 | $TLB_{VF}$ then
      data storage interrupt
   else if store &        w = 0 | $TLB_{VF}$ then
      data storage interrupt
   else
      access permitted
```

6.5.5 Access Control

If address translation results in a match (hit), the matching TLB entry is used to perform access control (permission checks). These checks are based on the privilege level of the access (MSR[PR]) and the type of access (fetch for execute, read for loads, and write for stores). The TLB entry's permission bits (TLB[US,SX,UW,SW,UR,SR]) determine whether the operation should succeed. If permission is denied, execution of the instruction is suppressed and an instruction or data storage interrupt occurs. Software uses the ESR, SRR0, and DEAR to determine the type of operation attempted, and then must perform a TLB search if updating the TLB is desired.

6.5.5.1 Page Size and Real Address Generation

If no virtual address match occurs, the translation fails and a TLB miss exception occurs, generating the appropriate instruction or data TLB error interrupt. Otherwise, the real page number (RPN) field of the matching TLB entry provides the translation for the effective address. Based on the setting of the TSIZE field of the matching TLB entry, the RPN field replaces the *n* corresponding msbs of the effective address where n = 54 - (2 * TSIZE). 32-bit implementations do not use the upper 32 bits of EA. Note that the untranslated bits must be zero in the RPN field. Real address generation based on page size is shown in Table 6-8.

Table 6-8. Real Address Generation

TSIZE Field	Page Size (4^{TSIZE}Kbytes)	64-Bit Implementations		32-Bit Implementations	
		RPN Bits that Must Equal 0	Real Address	RPN Bits that Must Equal 0	Real Address
0b0001	4 Kbyte	None	RPN[0–51] || EA[52–63]	None	RPN[32–51] || EA[52–63]
0b0010	16 Kbyte	RPN[50–51] = 0	RPN[0–49] || EA[50–63]	RPN[50–51] = 0	RPN[32–49] || EA[50–63]
0b0011	64 Kbyte	RPN[48–51] = 0	RPN[0–47] || EA[48–63]	RPN[48–51] = 0	RPN[32–47] || EA[48–63]
0b0100	256 Kbyte	RPN[46–51] = 0	RPN[0–45] || EA[46–63]	RPN[46–51] = 0	RPN[32–45] || EA[46–63]
0b0101	1 Mbyte	RPN[44–51] = 0	RPN[0–43] || EA[44–63]	RPN[44–51] = 0	RPN[32–43] || EA[44–63]
0b0110	4 Mbyte	RPN[42–51] = 0	RPN[0–41] || EA[42–63]	RPN[42–51] = 0	RPN[32–41] || EA[42–63]
0b0111	16 Mbyte	RPN[40–51] = 0	RPN[0–39] || EA[40–63]	RPN[40–51] = 0	RPN[32–39] || EA[40–63]
0b1000	64 Mbyte	RPN[38–51] = 0	RPN[0–37] || EA[38–63]	RPN[38–51] = 0	RPN[32–37] || EA[38–63]
0b1001	256 Mbyte	RPN[36–51] = 0	RPN[0–35] || EA[36–63]	RPN[36–51] = 0	RPN[32–35] || EA[36–63]
0b1010	1 Gbyte	RPN[34–51] = 0	RPN[0–33] || EA[34–63]	RPN[34–51] = 0	RPN[32–33] || EA[34–63]
0b1011	4 Gbyte	RPN[32–51] = 0	RPN[0–31] || EA[32–63]	—	
0b1100	16 Gbyte	RPN[30–51] = 0	RPN[0–29] || EA[30–63]	—	
0b1101	64 Gbyte	RPN[28–51] = 0	RPN[0–27] || EA[28–63]	—	
0b1110	256 Gbyte	RPN[26–51] = 0	RPN[0–25] || EA[26–63]	—	
0b1111	1 Tbyte	RPN[24–51] = 0	RPN[0–23] || EA[24–63]	—	

6.5.6 Permission Attributes

The permission attributes are shown in this table, and described in the following subsections.

Table 6-9. Permission Control for Instruction, Data Read, and Data Write Accesses

Access Type	MSR[PR]	TLB[UX]		TLB[SX]		TLB[UR]		TLB[SR]		TLB[UW]		TLB[SW]	
		0	1	0	1	0	1	0	1	0	1	0	1
Instruction fetch	0	—	—	ISI	√	—	—	—	—	—	—	—	—
	1	ISI	√	—	—	—	—	—	—	—	—	—	—
Data read (load)	0	—	—	—	—	—	—	DSI	√	—	—	—	—
	1	—	—	—	—	DSI	√	—	—	—	—	—	—
Data write (store)	0	—	—	—	—	—	—	—	—	—	—	DSI	√
	1	—	—	—	—	—	—	—	—	DSI	√	—	—

6.5.6.1 Execute Access Permission

The UX and SX bits of the TLB entry control execute access to the corresponding page.

Instructions may be fetched and executed from a page in memory if MSR[PR] = 1 (user mode) if the UX access control bit for that page is set. If the UX access control bit is cleared, instructions from that page are not fetched and they are not placed into any cache while the processor is in user mode.

Instructions may be fetched and executed from a page in memory if MSR[PR] = 0 (supervisor mode) and the SX access control bit for that page is set. If the SX access control bit is cleared, instructions from that page are not fetched and are not placed into any cache while the processor is in supervisor mode.

If the sequential execution model calls for the execution of an instruction from a page that is not enabled for execution (that is, UX = 0 when MSR[PR] = 1 or SX = 0 when MSR[PR] = 0), an execute access control exception-type instruction storage interrupt is taken.

6.5.6.2 Read Access Permission

The UR and SR bits of the TLB entry control read access to the corresponding page.

Load operations (including load-class cache management instructions) are permitted from a page in memory while the processor is in user mode (MSR[PR] = 1) if the UR access control bit for that page is set. If the UR access control bit is cleared, execution of the load instruction is suppressed and a read access control exception-type data storage interrupt is taken.

Similarly, load operations (including load-class cache management instructions) are permitted from a page in memory if MSR[PR] = 0 (supervisor mode) and the SR access control bit for that page is set. If the SR access control bit is cleared, execution of the load instruction is suppressed and a read access control exception-type data storage interrupt is taken.

6.5.6.3 Write Access Permission

The UW and SW bits of the TLB entry control write access to the corresponding page.

Store operations (including store-class cache management instructions) are permitted to a page in memory if MSR[PR] = 1 (user mode) and the UW access control bit for that page is set. If the UW access control bit is cleared, execution of the store instruction is suppressed and a write access control exception-type data storage interrupt is taken.

Similarly, store operations (including store-class cache management instructions) are permitted to a page in memory if MSR[PR] = 0 (supervisor mode) and the SW access control bit for that page is set. If the SW access control bit is cleared, execution of the store instruction is suppressed and a write access control exception-type data storage interrupt is taken.

6.5.6.4 Permission Control and Cache Management Instructions

The **dcbi**, **dcbz**, **dcbzep** <E.PD>, **dcbzl** <DEO>, and **dcbzlep** <E.PD, DEO> instructions are treated as stores because they can change data (or cause loss of data by invalidating a modified line). As such, they both can cause write access control exception-type DSIs.

Although **dcba** and **dcbal** <DEO> can change data, is treated as a store, and can cause write access control exceptions, these exceptions do not result in a data storage interrupt. If a permission violation occurs, the instruction executes, but the allocate operation is cancelled (essentially, a no-op).

The **icbi** and **icbiep** <E.PD> instructions are treated as a load with respect to permissions checking. As such, it can cause a read access control exception-type data storage interrupt. It is implementation dependent whether execute access (UX or SX depending on MSR[PR]) is sufficient access permission for **icbi** and **icbiep** <E.PD>.

The **dcbt**, **dcbtst**, **dcbtep** <E.PD>, **dcbtstep** <E.PD>, and **icbt** instructions are treated as loads with respect to permissions checking and can cause read access control exceptions. However, such exceptions do not result in data storage interrupts. If a permission violation occurs, the instruction executes, but the operation is cancelled (essentially, a no-op). It is implementation dependent whether execute access (UX or SX depending on MSR[PR]) is sufficient access permission for **icbt**

The **dcbf**, **dcbfep** <E.PD>, **dcbst**, and **dcbstep** <E.PD> instructions are treated as loads with respect to permissions checking. Flushing or storing a line from the cache is not considered a store because the store has already been performed to update the cache and the instruction is only updating the copy in main memory. Like load instructions, these instructions can cause read access control exception-type data storage interrupts.

This table summarizes exceptions caused by cache management instructions due to permissions violations.

Table 6-10. Permission Control and Cache Instructions

Instruction	Can Cause Read Permission Violation Exception?	Can Cause Write Permission Violation Exception?
dcba	No	Yes[1]
dcbal <DEO>	No	Yes[1]
dcbf	Yes	No
dcbfep <E.PD>	Yes	No
dcbi	No	Yes
dcbst	Yes	No

Table 6-10. Permission Control and Cache Instructions (continued)

Instruction	Can Cause Read Permission Violation Exception?	Can Cause Write Permission Violation Exception?
dcbstep <E.PD>	Yes	No
dcbt	Yes[1]	No
dcbtep <E.PD>	Yes[1]	No
dcbtst	Yes[1]	No
dcbtstep <E.PD>	Yes[1]	No
dcbz	No	Yes
dcbzep <E.PD>	No	Yes
dcbzepl <E.PD,DEO>	No	Yes
dcbzl <DEO>	No	Yes
icbi	Yes	No
icbiep <E.PD>	Yes	No
icbt	Yes[1]	No

[1] **dcba**, **dcbal** <DEO>, **dcbt**, **dcbtst**, **dcbtep** <E.PD>, **dcbtstep** <E.PD>, and **icbt** may cause a read access control exception but do not result in a data storage interrupt.

6.5.6.5 Use of Permissions to Maintain Page History

The TLB entry definition does not include bits for maintaining page history information. Software can use TLB entry bits U0–U3 to store history information, but implementations may ignore these bits internally.

System software can implement page changed bit status by disabling write permissions to all pages. The first attempt to write to the page causes a data storage interrupt. At this point, system software can record the page changed bit in memory, update the TLB entry permission to allow writes to that page, and return to the user program allowing further writes to the page to proceed without causing an exception.

6.5.7 Crossing Page Boundaries

Care must be taken with single instruction accesses (load/stores) that cross page boundaries. Examples are **lmw** and **stmw** instructions and misaligned accesses on implementations that support misaligned load/stores. Architecturally, each of the parts of the access that cross the natural boundary of the access size (half word, word, double word) are treated separately with respect to exception conditions. Additionally, these types of instructions may optionally partially complete. For example, a store word instruction that crosses a page boundary because it is misaligned to the last half word of a page might actually store the first 16 bits because the access was permitted, but produce a DSI or data TLB error exception because the second 16 bits in the next page were invalid or protected. An implementation may suppress the first 16-bit store or perform it.

6.5.8 MMU Exception Handling

When translation-related exceptions occur, hardware preloads the MAS registers with information that the interrupt handler likely needs to handle the fault. For a TLB miss exception, some MAS register fields are loaded with default information specified in MAS4. System software should set up the default information in MAS4 before allowing exceptions. In most cases, system software sets this up once depending on its scheme for handling page faults. This simplifies translation-related exception handling. The following subsections detail specific MAS register fields and the contents loaded for each exception type.

6.5.8.1 TLB Miss Exception Types

A TLB miss exception is caused when a virtual address for an access does not match with that of any on-chip TLB entry. This condition causes one of the following:

- An instruction TLB error interrupt
- A data TLB error interrupt

6.5.8.1.1 Instruction TLB Error Interrupt Settings

An instruction TLB error interrupt occurs when the virtual address associated with an instruction address (fetch) causes a TLB miss exception (that is, the address for the instruction cannot be translated). In addition to the values automatically written to the MAS registers (described in Section 6.5.8.1.3, “TLB Miss Exception MAS Register Settings”), the save/restore register 0 (SRR0 or GSRR0 <E.HV>) contains the address of the instruction that caused the instruction TLB error. This register is used to identify the effective address for handling the exception as well as the address to return to when system software has resolved the exception condition by writing a new TLB entry.

6.5.8.1.2 Data TLB Error Interrupt Settings

A data TLB error interrupt occurs when the virtual address associated with a data reference from a load, store, or cache management instruction causes a TLB miss exception (that is, the address of the data item of a load or store instruction cannot be translated). In addition to the values automatically written to the MAS registers (described in Section 6.5.8.1.3, “TLB Miss Exception MAS Register Settings”), the effective address of the data access that caused the exception is automatically loaded in the data exception address register (DEAR or GDEAR <E.HV>). Also, save/restore register 0 (SRR0 or GSRR0 <E.HV>) contains the address of the instruction that caused the data TLB error and its value is used to identify the address to return to when system software has resolved the exception condition (by writing a new TLB entry).

6.5.8.1.3 TLB Miss Exception MAS Register Settings

When either an instruction or data TLB error interrupt occurs, the TLB information and selection fields of the MAS registers are loaded with default values from other MAS registers to assist in processing the exception. The intention is that the common case of a page fault generally requires only system software to load the RPN (corresponding to the physical address that will be used for this page), and the access permissions and the defaults can be used for the remaining MAS fields.

The processor may use the next victim (NV) field from the TLB array to select which TLB entry should be used for the new translation. The method used to select the candidate TLB for replacement (the next victim) is implementation-dependent and may vary on different Freescale implementations. In any case, software is free to choose any TLB entry for the replacement (software can overwrite the value in MAS0[ESEL]).

The EIS defines the fields set in the MAS registers at exception time for an instruction or data TLB error interrupt as shown in Table 6-11.

6.5.8.2 Permissions Violation Exception Types

A permissions violation exception (read access control, write access control, or execute access exception) is caused when a virtual address for an access matches a TLB entry but the permission attributes in the matching TLB entry do not allow the access to proceed, as described in Section 6.5.6, “Permission Attributes.” This condition causes an instruction or data storage interrupt.

6.5.8.2.1 Virtualization Fault Exceptions <E.HV>

A virtualization fault is a special kind of permissions violation. When the VF bit is set in a matching TLB entry during translation, a virtualization fault exception occurs and if it is the highest priority exception condition, the processor takes a data storage interrupt which is always directed to the hypervisor. The hypervisor uses this capability to create TLB entries for accesses that it may wish to emulate.

NOTE: Software Considerations

Using the VF bit in TLB entries allows the hypervisor to create “virtual devices” which it can emulate.

6.5.8.2.2 Instruction Storage Interrupt Settings

An instruction storage interrupt occurs when the virtual address associated with an instruction address (fetch) matches a valid entry in the TLB, but an execute access control exception occurs. Save/restore register 0 (SRR0 or GSRR0 <E.HV>) contains the address of the instruction that caused the instruction storage interrupt and its value is used to identify the address to return to when system software has resolved the exception condition (by writing a new TLB entry).

6.5.8.2.3 Data Storage Interrupt Settings

A data storage interrupt occurs when the virtual address associated with a data reference from a load or store instruction matches a valid entry in the TLB but one of the following exceptions occur:

- read access control exception
- write access control exception
- virtualization fault exception <E.HV>

In this case, the effective address of the data access that caused the exception is contained in the data exception address register (DEAR or GDEAR <E.HV>) is used by system software to identify the address that caused the exception. Also, save/restore register 0 (SRR0 or GSRR0 <E.HV>) contains the address of the instruction that caused the data storage interrupt and its value is used to identify the address to return to when system software has resolved the exception condition by writing a new TLB entry.

6.5.8.3 MAS Register Updates for Exceptions, tlbsx, and tlbre

Table 6-11 summarizes how MAS register are updated by hardware for each stimulus. Note that an implementations may further define how certain MAS fields are set on exceptions.

The table can be interpreted as follows:

- A field name refers to a MAS register field.
- PID, MSR, EPLC <E.HV>, and EPSC <E.HV> refer to their respective registers.
- EA refers to the effective address used for the memory access which caused a TLB error (miss).
- The TLB entry specified by TLBSEL and ESEL is referred to as TLB.

Table 6-11. MMU Assist Register Field Updates

MAS Field	Inst TLB Error Data TLB Error	tlbsx Hit	tlbsx Miss	tlbre
MAS0				
TLBSEL	TLBSELD	which TLB array hit	TLBSELD	—

ESEL	if TLBSELD supports NV:
	hint]
	else
	undefined
		entry that hit	if TLBSELD supports NV:
			hint
			else
			undefined
				—

NV	if TLBSELD supports NV:
	next NV
	else
	undefined
		if TLBSEL supports NV:
		NV
		else
		undefined
			if TLBSELD supports NV:
			next NV
			else
			undefined
				if TLBSEL supports NV:
				NV
				else
				undefined

MAS1				
IPROT	0	If TLB that hits supports IPROT: TLB[IPROT] else 0	0	If TLB that hits supports IPROT: TLB[IPROT] else 0
TID	if ext PID load EPLC[EPID] elseif ext PID store EPSC[EPID] else PID	TLB[TID]	SPID	TLB[TID]
TSIZE	TSIZED	TLB[TSIZE]	TSIZED	TLB[TSIZE]

Table 6-11. MMU Assist Register Field Updates (continued)

MAS Field	Inst TLB Error Data TLB Error	tlbsx Hit	tlbsx Miss	tlbre
TS	if Data TLB Error if ext PID load EPLC[EAS] elseif ext PID store EPSC[EAS] else MSR[DS] else MSR[IS]	TLB[TS]	SAS	TLB[TS]
V	1	1	0	TLB[V]
MAS2				
WIMGE	WIMGED	TLB[WIMGE]	WIMGED	TLB[WIMGE]
VLE <VLE>	VLED	TLB[VLE]	VLED	TLB[VLE]
ACM	ACMD	TLB[ACM]	ACMD	TLB[ACM]
X0, X1	X0D, X1D	TLB[X0, X1]	X0D, X1D	TLB[X0,X1]

EPN[0:31]	EA[0:31] of access
	Note: if MSR[CM] = 0, then EA[0:31] must be 0.
		if MSR[CM] = 0
		undefined
		else
		TLB[EPN[0:31]]
			—	if MSR[CM] = 0
				undefined
				else
				TLB[EPN[0:31]]

EPN[32:51]	EA[32:51] of access	TLB[EPN[32:51]]	—	TLB[EPN[32:51]]
MAS3				
UR,SR,UW, SW,UX,SX	Zeros	TLB[UR,SR,UW,SW,UX,SX]	Zeros	TLB[UR,SR,UW,SW,UX,SX]
U0–U3	Zeros	TLB[U0-U3]	Zeros	TLB[U0-U3]
RPNL	Zeros	TLB[RPN[32:51]]	Zeros	TLB[RPN[32:51]]
MAS4				
WIMGED, X0D,X1D, TIDSELD, TLBSELD, TSIZED	—	—	—	—
VLED <VLE>	—	—	—	—

Table 6-11. MMU Assist Register Field Updates (continued)

MAS Field	Inst TLB Error Data TLB Error	tlbsx Hit	tlbsx Miss	tlbre
ACMD	—	—	—	—
MAS5 <E.HV>				
SGS <E.HV>	—	—	—	—
SLPID <E.HV>	—	—	—	—
MAS6				

SAS	if Data TLB Error
	  if ext PID load
	    EPLC[EAS]
	  elseif ext PID store
	    EPSC[EAS]
	  else
	    MSR[DS]
	else
	  MSR[IS]
		—	—	—

SPID	if ext PID load
	  EPLC[EPID]
	elseif ext PID store
	  EPSC[EPID]
	else
	  PID
		—	—	—

MAS7				
RPNU	Zeros	TLB[RPN[0:31]]	Zeros	TLB[RPN[0:31]]
MAS8 <E.HV>				
TGS <E.HV>	—	TLB[TGS]	—	TLB[TGS]
VF <E.HV>	—	TLB[VF]	—	TLB[VF]
TLPID <E.HV>	—	TLB[TLPID]	—	TLB[TLPID]

6.5.9 MMU Configuration Information

Information about the configuration for a given TLB implementation is available to system software by reading the contents of the MMU configuration SPRs. These SPRs describe the architectural version of the MMU, the number of TLB arrays, and the characteristics of each TLB array.

- MMU configuration register (MMUCFG) contains basic information about the MMU architecture for this device. See Section 3.12.4, “MMU Configuration Register (MMUCFG).”

- TLB configuration registers (TLB*n*CFG) contain configuration information about each TLB array. See Section 3.12.5, "TLB Configuration Registers (TLBnCFG)."

  The TLB*n*CFG number assignment is the same as the value in MAS0[TLBSEL]. For example, TLB0CFG provides configuration information about TLB0, TLB1CFG provides configuration information about TLB1, etc.

The MMU configuration information is provided for software to be able to adapt to differing implementations within the framework of the architecture.

Chapter 7
Interrupts and Exceptions

This chapter provides a description of the exception and interrupt models as they are implemented on Freescale Power ISA™ embedded processors. It identifies and describes the portions of the interrupt model that are defined by the Power ISA and by the Freescale implementation standards (EIS). Other features of the architecture associated with interrupts are also discussed here.

NOTE on Terminology

This document uses the terms 'exception' and 'interrupt' as follows:

- An exception is the event that, if enabled, causes the processor to take an interrupt. Exceptions are generated by signals from internal and external peripherals, instructions, the internal timer facility, debug events, or error conditions.
- An interrupt is the action in which the processor saves its context (typically the machine state register (MSR) and next instruction address) and begins execution at a predetermined interrupt handler address with a modified MSR.

7.1 Overview

As described in Section 7.2, "Interrupt Classes," there are four classes of interrupts, each with a dedicated pair of registers (*x*SRR0 and *x*SRR1) for saving and restoring the machine state when an interrupt is taken, and an instruction that returns control from the interrupt handler to the interrupted process.

Note that these registers are serially reusable, program state may be lost when an unordered interrupt is taken. (See Section 7.10, "Interrupt Ordering and Masking.")

All interrupts are context synchronizing as defined in Section 4.5.4.4, "Context Synchronization."

Interrupt handlers reside at programmable offsets, determined by the contents of the interrupt vector prefix register (IVPR) or guest interrupt vector prefix register (GIVPR)<E.HV> concatenated with the interrupt vector offset register (IVOR*n*) or guest interrupt vector offset register (GIVOR*n*)<E.HV> specific to the interrupt. These are described in Section 7.3, "Interrupt Registers."

7.2 Interrupt Classes

This table describes the interrupt classes defined by the Power ISA. Each has a dedicated pair of save/restore SPRs and an instruction for accessing them. Implementations may define additional classes.

Table 7-1. Interrupt Classes

Category	Description	Programming Resources
Noncritical interrupts	Standard, non-critical interrupts are used to handle most synchronous program exception conditions. Standard interrupts are also used to handle the asynchronous external input exception, asynchronous decrementer exceptions, asynchronous fixed interval timer exceptions, performance monitor exceptions <E.PM>, and asynchronous doorbell exceptions <E.PC,E.HV>.	SRR0/SRR1 SPRs and **rfi** instruction. Asynchronous noncritical interrupts can be masked by the external interrupt enable bit, MSR[EE]. Synchronous program exception interrupts are not maskable. For guest operating systems operating under control of a hypervisor, separate guest save and restore registers are provided (GSRR0/GSRR1) and the **rfgi** instruction. <E.HV> Masking conditions for interrupts also depends on MSR[GS] and whether the interrupt is directed to the hypervisor state or the guest supervisor state.<E.HV>
Critical interrupts	Critical input, watchdog timer, critical asynchronous doorbell <E.PC>, and debug interrupts. These interrupts can be taken during a noncritical interrupt or during regular program flow. If category E.ED is not implemented, debug interrupts are also implemented as critical interrupts.	CSRR0/CSRR1 and **rfci**. Critical interrupts can be masked by the critical enable bit, MSR[CE]. Debug interrupts can be masked by the debug enable bit MSR[DE]. Masking conditions for interrupts also depends on MSR[GS].<E.HV>
Debug interrupt	<E.ED>: Provides a separate set of resources for the debug interrupt. Consult the applicable device reference manual to determine whether the debug interrupt is implemented. See Section 7.8.15, “Debug Interrupt.”	DSRR0/DSRR1 and **rfdi**. Debug interrupts can be masked by the debug enable bit MSR[DE].
Machine check interrupt	Provides a separate set of resources for the asynchronous machine check interrupt, synchronous error report interrupt, and the non-maskable interrupt (NMI). See Section 7.8.2, “Machine Check, NMI, and Error Report Interrupts.”	MCSRR0/MCSRR1 and **rfmci**. Asynchronous machine check interrupts can be masked with the machine check enable bit, MSR[ME]. Synchronous error report interrupts and asynchronous NMI interrupts are not maskable.

Because save/restore register pairs are serially reusable, care must be taken to preserve program state that may be lost when an unordered interrupt is taken. See Section 7.10, “Interrupt Ordering and Masking.”

All interrupts are context synchronizing as defined in Section 4.5.4.4, “Context Synchronization.”

7.2.1 Recoverability from Interrupts

In general, all interrupts except NMI are recoverable. That is, the return address and MSR contents are saved when an interrupt is taken and correctly written software can correctly return. In some cases, software may not be able to correct a machine check condition. Interrupt handlers for all interrupts must be carefully written to avoid taking a synchronous program exception condition prior to the save/restore registers associated with that condition being successfully saved. In general well written software can avoid these conditions. However, it is possible that a synchronous error report exception and interrupt can occur in the machine check/error report interrupt handler prior to saving the save/restore registers. Software may notice this case by manipulating MSR[RI] in the handler to denote whether another interrupt has occurred before the save/restore registers were saved.

NMI interrupts, by nature, are considered non-recoverable because they are asynchronous and non-maskable. NMI interrupts should only be used for situations where the current state of the processor will be discarded such as a soft reset scenario.

7.3 Interrupt Registers

When interrupts occur, information about the state of the processor is saved to associated save/restore registers and the processor begins execution at an address determined by the concatenation of the interrupt vector prefix register (IVPR) and the interrupt vector offset register (IVOR) associated with the interrupt.

If the interrupt is directed to the guest supervisor state (see Section 7.4, “Directed Interrupts <E.HV>), the processor begins execution at an address determined by the concatenation of the guest interrupt vector prefix register (GIVPR) and the guest interrupt vector offset register (GIVOR) associated with the interrupt.<E.HV>

Based on the interrupt, a new MSR value is established, which results in the following:

- Places the processor in a privileged state (MSR[PR], MSR[GS]<E.HV>).
- Sets a predetermined virtual address space (MSR[IS,DS], MSR[GS]<E.HV>).
- Sets the computation mode (MSR[CM]).<64>
- Masks further asynchronous interrupts of the same class and any lower class (MSR[EE,CE,DE,ME]).

Depending on the interrupt, other interrupt registers may be set to indicate more information about the interrupt. Such registers include the exception syndrome register (ESR, GESR<E.HV>), the data exception address register (DEAR, GDEAR<E.HV>), the external proxy register (EPR, GEPR<E.HV>)<EXP>, the machine check syndrome register (MCSR), and the machine check address registers (MCAR, MCARU),

This table shows the interrupt registers and the categories associated with them.

Table 7-2. Interrupt Registers

spr 26	SRR0	Save/restore registers 0/1	Base
spr 27	SRR1		
spr 26	GSRR0	Save/restore registers 0/1	Embedded.Hypervisor
spr 27	GSRR1		

Table 7-2. Interrupt Registers (continued)

spr 58	CSRR0	Critical SRR 0/1	Base
spr 59	CSRR1		
spr 574	DSRR0	Debug SRR 0/1	Embedded.Enhanced Debug
spr 575	DSRR1		
spr 570	MCSRR0	Machine check SRR 0/1	Base
spr 571	MCSRR1		
spr 572	MCSR	Machine check syndrome register	
spr 569	MCARU	Machine check address upper/lower	
spr 573	MCAR		
spr 62	ESR	Exception syndrome register	
spr 61	DEAR	Data exception address register	
spr 63	IVPR	Interrupt vector prefix register	
spr 400	IVOR0	Interrupt vector offset registers 0–15	
spr 401	IVOR1		
	• • •		
spr 415	IVOR15		
spr 528	IVOR32	Interrupt vector offset register 32	SPE or Vector
spr 529	IVOR33	Interrupt vector offset register 33	SP.FD, SP.FS, SP.FV, or Vector
spr 530	IVOR34	Interrupt vector offset register 34	SP.FD, SP.FS, or SP.FV
spr 531	IVOR35	Interrupt vector offset register 35	Embedded.Performance Monitor

Table 7-2. Interrupt Registers (continued)

spr 532	IVOR36	Interrupt vector offset register 36	Embedded.Processor Control
spr 533	IVOR37	Interrupt vector offset register 37	
spr 432	IVOR38	Interrupt vector offset register 38	Embedded.Hypervisor
spr 433	IVOR39	Interrupt vector offset register 39	
spr 434	IVOR40	Interrupt vector offset register 40	
spr 435	IVOR41	Interrupt vector offset register 41	
spr 440	GIVOR2	Guest Interrupt vector offset register 2	
spr 441	GIVOR3	Guest Interrupt vector offset register 3	
spr 442	GIVOR4	Guest Interrupt vector offset register 4	
spr 443	GIVOR8	Guest Interrupt vector offset register 8	
spr 444	GIVOR13	Guest Interrupt vector offset register 13	
spr 445	GIVOR14	Guest Interrupt vector offset register 14	
spr 446	GIVPR	Guest Interrupt vector prefix register	
spr 383	GESR	Guest exception syndrome register	
spr 381	GDEAR	Guest data exception address register	
spr 383	EPR	External proxy register	External Proxy
spr 381	GEPR	Guest external proxy register	External Proxy and Embedded.Hypervisor

7.3.1 Save/Restore Registers

Save/restore registers are automatically updated with machine state information and the return address when an interrupt is taken. These registers are summarized below and described in detail in Section 3.8, “Interrupt Registers.”

- Save/restore register *x*SRR0. *x*SRR0 holds the address of the instruction where an interrupted process should resume. For instruction-caused interrupts, it is typically the address of the instruction that caused the interrupt. In some cases when the interrupt is a post-completion interrupt, the address points to the instruction to be executed after the instruction which caused the interrupt. A debug instruction complete exception and an embedded floating-point round interrupt are examples of such post-completion interrupts. When a return from interrupt instruction (**rfi, rfgi**<E.HV>, **rfci**, **rfdi**<E.ED>, or **rfmci**) executes, instruction execution continues at the address in *x*SRR0.
  - SRR0 is used for non-critical interrupts, and **rfi** is used to return from the interrupt.
  - GSRR0 is used for non-critical interrupts, and **rfgi** is used to return from the interrupt. <E.HV>
  - CSRR0 is used for critical interrupts, and **rfci** is used to return from the interrupt.
  - DSRR0 is used for debug interrupts, and **rfid** is used to return from the interrupt. <E.ED>

  - MCSRR0 is used for machine-check and error report interrupts, and **rfmci** is used to return from the interrupt.
- *x*SRR1 holds machine state information. When an interrupt is taken, MSR contents are placed in *x*SRR1. When a return from interrupt instruction (**rfi, rfgi**<E.HV>, **rfci**, **rfdi**<E.ED>, or **rfmci**) executes, *x*SRR1 contents are placed into MSR.
  - SRR1 is used for non-critical interrupts, and **rfi** is used to return from the interrupt.
  - GSRR1 is used for non-critical interrupts, and **rfgi** is used to return from the interrupt. <E.HV>
  - CSRR1 is used for critical interrupts, and **rfci** is used to return from the interrupt.
  - DSRR1 is used for debug interrupts, and **rfid** is used to return from the interrupt. <E.ED>
  - MCSRR1 is used for machine-check and error report interrupts, and **rfmci** is used to return from the interrupt.

7.3.2 Other Registers that Help Service the Interrupt

Additional registers are provided to further assist the interrupt handler in servicing the interrupt:

- Exception syndrome register (ESR). The ESR provides a way to differentiate between exceptions that can generate an interrupt type.
  The guest exception syndrome register (GESR) is provided for interrupts directed to the guest supervisor state. <E.HV>
- Data exception address register (DEAR). Loaded with the effective address of a data access (caused by a load, store, or cache management instruction) that results in an alignment, data TLB miss, or DSI exception.
  The guest data exception address register (GDEAR) is provided for interrupts directed to the guest supervisor state. <E.HV>
- Interrupt vector prefix register (IVPR). Used with IVORs to determine the vector address. The high-order 48 bits (16 bits in 32-bit mode) of IVPR is concatenated with 12 bits from the IVOR associated with the interrupt to form the address of the interrupt routine.
  The guest interrupt vector prefix register (GIVPR) is provided for interrupts directed to the guest supervisor state. <E.HV>
- Interrupt vector offset registers (IVOR*n*). IVORs provide the index from the base address provided by the IVPR for its respective interrupt type. Figure 4-1 shows IVORs defined by the Power ISA and the Freescale EIS.
  The guest interrupt vector offset registers (GIVOR*n*) are provided for interrupts directed to the guest supervisor state. <E.HV>
- Machine check syndrome register (MCSR). When an asynchronous machine-check condition occurs, MCSR is updated to reflect the condition. A non-zero value for bits associated with asynchronous machine check conditions will cause an asynchronous machine check interrupt if MSR[ME] is enabled. MCSR is also updated when an error report interrupt or NMI interrupt is taken. The MCSR indicates information about the machine-check or error report and software can use such information to fix the condition or decide to take other actions.
- Machine check address register (MCAR/MCARU). When the processor takes a machine-check interrupt that has an address associated with it, MCAR/MCARU indicates the address of the data associated with the machine check. MCAR/MCARU may contain either an effective address or a

real address depending on the error condition. Bits in MCSR indicate whether MCAR/MCARU is valid and if so, the type of address it contains.

- <EXP>:
  External proxy register (EPR). When an external input interrupt occurs, the processor places the vector from the programmable interrupt controller into the EPR register. The act of taking the interrupt causes the programmable interrupt controller to acknowledge the interrupt (as if the IACK register was read). This reduces interrupt latency.
  The guest interrupt proxy register (GEPR) is provided for external input interrupts directed to the guest supervisor state. <E.HV>

7.4 Directed Interrupts <E.HV>

A directed interrupt specifies the state in which the interrupt is taken. Processors running a hypervisor program and a guest operating system require that some interrupts be serviced by the guest operating system, and do not need any intervention from the hypervisor. The state in which the processor takes an interrupt is defined by the term “directed.” Thus, “directed to the hypervisor state,” means the interrupt is taken in the hypervisor state and conversely, “directed to the guest supervisor state,” means the interrupt is taken in the guest supervisor state.

Interrupts are directed to either the guest supervisor state or the hypervisor state. The state to which interrupts are directed determines which SPRs are used to form the vector address, which save/restore register are used to capture the processor state at the time of the interrupt, and which ESR/DEAR is used to post exception status. Interrupts directed to the guest state use the GIVPR to determine the upper 48 bits of the vector address and use GIVORs to provide the lower 16 bits. Interrupts directed to the hypervisor state use the IVPR and the IVORs. Interrupts that are directed to the guest state use GSRR0/GSRR1 registers to save the context at interrupt time. Interrupts directed to the hypervisor state use SRR0/SRR1, CSRR0/CSRR1, DSRR0/DSRR1, and MCSRR0/MCSRR1 for standard, critical, debug, and machine check interrupts respectively, with the exception of guest processor doorbell interrupts which use GSRR0/GSRR1.

In general, all interrupts are directed to the hypervisor state except for the following cases:

- The system call interrupt is directed to the state from which the interrupt was taken. If an **sc 0** instruction is executed in guest state, the interrupt is directed to the guest state. If an **sc 0** instruction is executed in hypervisor state, the interrupt is directed to the hypervisor state. Note that **sc 1** is always directed to the hypervisor state and produces a hypervisor system call interrupt.
- One of the following interrupts occurs while the processor is in the guest state, and the associated control bit in the EPCR is set to configure the interrupt to be directed to the guest state:
  - External input (EPCR[EXTGS] = 1)
  - Data TLB error (EPCR[DTLBGS] = 1)
  - Instruction TLB error (EPCR[ITLBGS] = 1)
  - Data storage (EPCR[DSIGS] = 1 and TLB[VF] = 0 [virtualization fault])
  - Instruction storage (EPCR[ISIGS] = 1)

NOTE

A data storage interrupt caused by a virtualization fault exception will always be directed to the hypervisor state.

In no case will an interrupt be directed to the guest when the processor is executing in the hypervisor state.

The enabling conditions on whether an asynchronous interrupt will occur or not is affected by whether the processor is in the guest supervisor state or the hypervisor state, and which state the interrupt will be directed to. In general, interrupts that are enabled by MSR[EE] are enabled when directed to the hypervisor state when:

```
MSR[GS] | MSR[EE]
```

And interrupts that are enabled by MSR[EE] are enabled when directed to the guest supervisor state when:

```
MSR[GS] & MSR[EE]
```

Guest processor doorbell type interrupts have different enabling conditions because guest processor doorbell type interrupts are always taken when executing in guest supervisor state with the appropriate MSR enable bit set even though these interrupts are directed to the hypervisor state.

NOTE: Software Considerations

What bits in the MSR are set when the first instruction of an interrupt handler executes is affected by whether the interrupt is directed to the guest supervisor state or the hypervisor state. For interrupts directed to the guest supervisor state, MSR[GS] is always set and MSR bits protected by the MSRP register are not changed. See the interrupt definitions (Section 7.8, "Interrupt Definitions") for more details.

7.5 Exceptions

Exceptions are caused directly by instruction execution or by an asynchronous event. In either case, the exception may cause one of several types of interrupts to be invoked.

The following examples are of synchronous exceptions caused directly by instruction execution:

- An attempt to execute a reserved-illegal instruction (illegal instruction exception-program interrupt)
- An attempt by an application program to execute a privileged instruction or to access a privileged SPR regardless of whether that SPR is defined or implemented (privileged instruction exception-program interrupt)
- An attempt to access a nonexistent SPR (illegal instruction exception on all accesses to undefined SPRs, regardless of MSR[GS,PR]). Note the following behavior defined by the EIS:
  - If MSR[PR] = 1 (user mode), SPR bit 5 = 0 (user-accessible SPR), and the SPR number is invalid, an illegal instruction exception is taken.
  - If MSR[PR] = 0 (supervisor mode) and the SPR number is invalid, an illegal instruction exception is taken.

  - If MSR[PR] = 1, SPR bit 5 = 1, and invalid SPR number (supervisor-only SPR), a privileged instruction exception is taken.

- An attempt to access a hypervisor privileged SPR and the processor is in guest supervisor state (embedded hypervisor privilege exception-embedded hypervisor interrupt)
- An attempt to access a location that is either unavailable (TLB miss exception-instruction-instruction or data TLB error interrupt) or not permitted (read, write, or execute access control exception-instruction or data storage interrupt)
- An attempt to access a location with an effective address alignment not supported by the implementation (alignment exception-alignment interrupt)
- Execution of a system call (**sc**) instruction (system call or hypervisor system call exception-system call/hypervisor system call interrupt). Whether a system call interrupt occurs or a hypervisor system call interrupt occurs depends on the value of the LEV operand. <E.HV>
- Execution of a **trap** instruction whose trap condition is met (trap exception-program interrupt)
- Execution of a **ehpriv** instruction (embedded hypervisor privilege exception-embedded hypervisor privilege interrupt) <E.HV>
- Execution of an instruction which encounters an error condition in the processor (error report exception-error report interrupt)
- Execution of an instruction which causes a debug event to occur and the processor is in internal debug mode (debug exception-debug interrupt)
- An attempt to execute a floating-point instruction when floating-point instructions are unavailable (floating-point unavailable exception-floating-point unavailable interrupt) <FP>
- Execution of a floating-point instruction that causes a floating-point enabled exception to exist (floating-point enabled exception-program interrupt) <FP>
- An attempt to execute an SPE, embedded floating-point double precision, or embedded floating-point vector instruction when SPE instructions are unavailable (SPE unavailable exception-SPE unavailable interrupt) <SPE,SP.FD,SP.FV>
- Execution of an embedded floating-point single precision, embedded floating-point double precision, or embedded floating-point vector instruction that causes an embedded floating-point data exception to exist (embedded floating-point data exception-embedded floating-point data interrupt) <SP.FS,SP.FD,SP.FV>
- Execution of an embedded floating-point single precision, embedded floating-point double precision, or embedded floating-point vector instruction that causes an embedded floating-point round exception to exist (embedded floating-point round exception-embedded floating-point roundinterrupt) <SP.FS,SP.FD,SP.FV>
- An attempt to execute an AltiVec instruction when AltiVec instructions are unavailable (AltiVec unavailable exception-AltiVec unavailable interrupt) <V>
- Execution of an AltiVec floating-point instruction for which an input operand is NaN (AltiVec assist exception-AltiVec assist interrupt) <V>
- An attempt to execute a defined instruction that is not implemented (illegal instruction exception-program interrupt). Some older processors may take an unimplemented operation exception-program interrupt.

Invocation of an interrupt from a synchronous exception is precise.

7.6 Synchronous and Asynchronous Interrupts

Interrupts are either synchronous or asynchronous. Synchronous interrupts are associated directly with instruction execution. That is, the exception condition occurs as a result of execution of a specific instruction. Asynchronous interrupts are not associated with execution of a specific instruction, but as a result of an exception condition that occurs independent of the execution of a specific instruction. In some cases, execution of an instruction can cause an asynchronous interrupt to occur (for example, setting MSR[EE] can cause an external input interrupt to occur immediately after the instruction that sets MSR[EE] if an external input interrupt is currently being asserted), but, in general, when the interrupt that occurs is unrelated to the instruction execution.

Interrupts are always precise. Some older processors implement some machine check interrupts as imprecise. See the core reference manual for more details.

When an interrupt occurs, the value placed into *x*SRR0 is generally as shown in this table. For specific information about each interrupt type see Section 7.8, “Interrupt Definitions.”

Table 7-3. Asynchronous and Synchronous Interrupts *x*SRR0 Setting

Class	Description
Asynchronous	The address saved in *x*SRR0 is the address of the instruction that would have executed next, had the asynchronous interrupt not occurred.
Synchronous	The address saved in *x*SRR0 is the address of the instruction which caused the interrupt except for post-completion exceptions, in which case it is the address of the next instruction to be executed.

NOTE

When the execution or attempted execution of an instruction causes a synchronous or asynchronous interrupt (*x*SRR0 setting), the following conditions exist at the interrupt point:

- Whether the *x*SRR0 addresses the instruction causing the exception or the next instruction is determined by the interrupt type and status bits.
- An interrupt is generated such that all instructions before the instruction causing the exception appear to have completed with respect to the executing processor. However, some accesses associated with these preceding instructions may not have been performed with respect to other processors and mechanisms.
- The exception-causing instruction may appear not to have begun execution (except for causing the exception), may be partially executed, or may have completed, depending on the interrupt type. See Section 7.9, “Partially Executed Instructions.”
- Architecturally, no instruction beyond the exception-causing instruction has executed.

7.6.1 Requirements for System Reset Generation

The architecture does not define a system reset interrupt as was defined in the original PowerPC architecture. A system reset is typically initiated in one of the following ways:

- Assertion of a signal through the integrated device that resets the internal state of the core
- By writing a 1 to DBCR0[34], if MSR[DE] = 1
- Most integrated devices provide the capability to perform a system reset through SoC-specific functions including reset actions on the second timeout of the Watchdog Timer. See the Integrated Device Reference Manual.

An NMI interrupt signal can also be used to force the processor to take an NMI interrupt, which can be used by software to provide a soft reset type of capability.

7.7 Interrupt Processing

Associated with each kind of interrupt is an interrupt vector, the address of the initial instruction that is executed when an interrupt occurs.

When an exception exists that causes an interrupt to be generated, and it has been determined that the interrupt can be taken, the following steps are performed:

1. *x*SRR0 is loaded with an instruction address that depends on the type of interrupt. See the specific interrupt description for details.
2. *x*SRR1 is loaded with a copy of the MSR contents.
3. Other interrupt status registers (ESR, DEAR, MCSR, etc.) may be loaded with information specific to the exception type. Note that many interrupt types can only be caused by a single type of exception event, and thus do not need nor use any additional interrupt status registers to provide the cause of the interrupt.
4. A new MSR value takes effect beginning with the first instruction following the interrupt. The new MSR masks asynchronous interrupts at the same level from being enabled. The setting of the new MSR is defined in Section 7.8, “Interrupt Definitions, however, in general it is set as follows:
   - MSR[EE,PR,IS,DS] are set to 0.
   - MSR[SPV] is set to 0. <SP> or <V>
   - MSR[FP,FE0,FE1] is set to 0. <FP>
   - MSR[WE] is set to 0. (Older processors implement WE, newer processors do not. If a processor does not implement WE, the the bit is considered to not be defined).
   - MSR[CE] is set to 0 for critical class interrupts, debug interrupts, and machine check class interrupts.
   - MSR[DE] is set to 0 for debug interrupts and machine check class interrupts. MSR[DE] is also set to 0 for critical class interrupts when category E.ED is not implemented.
   - MSR[ME,RI] is set to 0 for machine check class interrupts.
   - If category E.HV is implemented and the interrupt is directed to the hypervisor state: <E.HV>
     - MSR[PMM] is set to 0. <E.PM>
     - MSR[UCLE] is set to 0. <E.CL>

   - MSR[CM] is set to EPCR[ICM]. <64>
   - MSR[GS] is set to 0.
- If category E.HV is implemented and the interrupt is directed to the guest supervisor state: <E.HV>
   - MSR[PMM] is set to 0 if MSRP[PMMP] = 0. <E.PM>
   - MSR[UCLE] is set to 0 if MSRP[UCLEP] = 0.. <E.CL>
   - MSR[CM] is set to EPCR[GICM]. <64>
- If category E.HV is not implemented:
   - MSR[PMM] is set to 0. <E.PM>
   - MSR[UCLE] is set to 0. <E.CL>
   - MSR[CM] is set to EPCR[ICM]. <64>
- Other defined MSR bits are unchanged by all interrupts. Other undefined MSR bits contain undefined values.
  MSR fields are described in Section 3.6.1, “Machine State Register (MSR).”

5. Instruction fetching and execution resumes, using the new MSR value, at a location specific to the interrupt type (IVPR[0–47] || IVOR*n*[48–59] || 0b0000).
   If the interrupt is directed to the guest supervisor state, instruction fetching and execution resumes, using the new MSR value, at a location specific to the interrupt type (GIVPR[0–47] || GIVOR*n*[48–59] || 0b0000). <E.HV>
   The IVOR*n* for the interrupt type is indicated in Table 7-4. IVPR and IVOR contents are indeterminate upon reset and must be initialized by system software.

Interrupts do not clear reservations obtained with load and reserve instructions. The operating system should do so at appropriate points; for example, at process switch.

At the end of a interrupt handling routine, executing the appropriate return from interrupt instruction (**rfi**, **rfci**, **rfmci**, **rfdi** <E.ED>, **rfgi** <E.HV>) causes the MSR to be restored from the *x*SRR1 contents and instruction execution to resume at the address contained in *x*SRR0.

NOTE: Software Considerations

In general, at process switch, due to possible process interlocks and possible data availability requirements, the operating system needs to consider executing the following:

- **stwcx.**—Clears outstanding reservations to prevent pairing a load and reserve instruction (**lbarx** <ER>, **lharx** <ER>, **lwarx**, **ldarx** <64>) in the old process with a store conditional instruction in the new one
- **sync**—Ensures that memory operations of an interrupted process complete with respect to other processors before that process begins executing on another processor
- Return from interrupt (x**rfi**)—Ensures that instructions in the new process execute in the new context

7.8 Interrupt Definitions

The following table summarizes each interrupt type, exceptions that may cause that interrupt, the interrupt classification, the ESR bits that can be set, the MSR bits that can mask the interrupt type, and which IVOR is used to specify the vector address.

Table 7-4. Interrupt Summary by IVOR

IVOR	Interrupt	Exception	Directing State at Exception <E.HV>	ESR[1]	Enabled by	Type[2]	Save and Restore Registers	Page
IVOR0	Critical input		—	—	MSR[CE] or MSR[GS]	A	CSRR0/1	7-18
IVOR1	Machine check		—	—	MSR[ME] or MSR[GS]	A	MCSRR0/1	7-18
	NMI		—	—	—	A	MCSRR0/1	7-18
	Error report		—	—	—	S	MCSRR0/1	7-18
IVOR2	Data storage (DSI)	Access	MSR[GS] = 0 or EPCR[DSIGS] = 0	[ST], [FP,SPV,VLEMI], [EPID]	—	S	SRR0/1	7-23
		Virtualization fault <E.HV>	TLB[VF] = 1	[ST], [FP,SPV,VLEMI], [EPID], [VLEMI]		S		
		Load reserve or store conditional to write-through required location (W = 1)	MSR[GS] = 0 or EPCR[DSIGS] = 0	[ST]		S		
		Cache locking		[DLK,ILK],[ST]		S		
		Byte ordering		[ST] [FP,SPV,VLEMI],BO, [EPID]		S		
GIVOR2 <E.HV>	Data storage (DSI)	Access	MSR[GS] = 1 and EPCR[DSIGS] = 1	[ST], [FP,SPV,VLEMI], [EPID]	—	S	GSRR0/1	7-23
		Load reserve or store conditional to write- through required location (W = 1)		[ST]	—	S		
		Cache locking		[DLK,ILK],[ST]	—	S		
		Byte ordering		[ST], [FP,SPV,VLEMI], BO, [EPID]	—	S		
IVOR3	Instruction storage (ISI)	Access	MSR[GS] = 0 or EPCR[ISIGS] = 0		—	S	SRR0/1	7-26

Table 7-4. Interrupt Summary by IVOR (continued)

IVOR	Interrupt	Exception	Directing State at Exception <E.HV>	ESR[1]	Enabled by	Type[2]	Save and Restore Registers	Page
GIVOR3 <E.HV>	Instruction storage (ISI)	Access	MSR[GS] = 1 and EPCR[ISIGS] = 1		—	S	GSRR0/1	7-26
IVOR4	External input[3]		EPCR[EXTGS] = 0	—	MSR[EE] or MSR[GS]	A	SRR0/1	7-27
GIVOR4 <E.HV>	External input[3]		EPCR[EXTGS] = 1	—	MSR[EE] and MSR[GS]	A	GSRR0/1	7-27
IVOR5	Alignment		—	[ST], [FP,SPV,VLEMI], [EPID]	—	S	SRR0/1	7-30
IVOR6	Program	Illegal	—	PIL	—	S	SRR0/1	7-31
		Privileged		PPR	—	S		
		Trap		PTR	—	S		
		Floating-point enabled		FP,[PIE]	MSR[FE0]¶ MSR[FE1]	S		
		Unimplemented op		PUO	—	S		
IVOR7 <FP>	Floating-point unavailable		—	—	—	S	SRR0/1	7-33
IVOR8	System call		MSR[GS] = 0	—	—	S*	SRR0/1	7-33
GIVOR8 <E.HV>	System call		MSR[GS] = 1	—	—	S*	GSRR0/1	7-33
IVOR9	AP unavailable (not defined by EIS)		—	—	—	S	SRR0/1	—
IVOR10	Decrementer		—	—	(MSR[EE] or MSR[GS]) and TCR[DIE]	A	SRR0/1	7-34
IVOR11	Fixed interval timer		—	—	(MSR[EE] or MSR[GS]) and TCR[FIE]	A	SRR0/1	7-35
IVOR12	Watchdog		—	—	(MSR[CE] or MSR[GS]) and TCR[WIE]	A	CSRR0/1	7-36

Table 7-4. Interrupt Summary by IVOR (continued)

IVOR	Interrupt	Exception	Directing State at Exception <E.HV>	ESR[1]	Enabled by	Type[2]	Save and Restore Registers	Page
IVOR13	Data TLB error	Data TLB miss	MSR[GS] = 0 or EPCR[DTLBGS] = 0	[ST], [FP,SPV,VLEMI], [EPID]	—	S	SRR0/1	7-39
GIVOR13 <E.HV>	Data TLB error	Data TLB miss	MSR[GS] = 1 and EPCR[DTLBGS] = 1	[ST], [FP,SPV,VLEMI], [EPID]	—	S	GSRR0/1	7-39
IVOR14	Instruction TLB error	Instruction TLB miss	MSR[GS] = 0 or EPCR[ITLBGS] = 0	—	—	S	SRR0/1	7-40
GIVOR14 <E.HV>	Instruction TLB error	Instruction TLB miss	MSR[GS] = 1 and EPCR[ITLBGS] = 1	—	—	S	GSRR0/1	7-40

Table 7-4. Interrupt Summary by IVOR (continued)

IVOR	Interrupt	Exception	Directing State at Exception <E.HV>	ESR[1]	Enabled by	Type[2]	Save and Restore Registers	Page
IVOR15	Debug	Trap (synchronous)	—	—	MSR[DE] and DBCR0[IDM]	S[4]	CSRR0/1 or DSRR0/1 if enhanced debug present	7-41
		Instruction address compare (synchronous)		—	MSR[DE] and DBCR0[IDM]	S[4]		
		Data address compare (synchronous)		—	MSR[DE] and DBCR0[IDM]	S[4]		
		Instruction complete		—	MSR[DE] and DBCR0[IDM]	S[4]*		
		Branch taken		—	MSR[DE] and DBCR0[IDM]	S[4]		
		Return from critical interrupt <E.ED>		—	MSR[DE] and DBCR0[IDM]	S[4]		
		Return from interrupt		—	MSR[DE] and DBCR0[IDM]	S[4]		
		Critical interrupt taken <E.ED>		—	MSR[DE] and DBCR0[IDM]	S[4]		
		Interrupt taken		—	MSR[DE] and DBCR0[IDM]	S[4]		
		Unconditional debug event		—	MSR[DE] and DBCR0[IDM]	A		

Table 7-4. Interrupt Summary by IVOR (continued)

IVOR	Interrupt	Exception	Directing State at Exception <E.HV>	ESR[1]	Enabled by	Type[2]	Save and Restore Registers	Page
IVOR32	SPE/ embedded floating -point/ AltiVec unavailable	SPE unavailable <SP>	—	SPV	—	S	SRR0/1	7-43
		Embedded floating-point unavailable <SP.FD,SP.FV>	—	SPV	—	S	SRR0/1	7-43
		AltiVec unavailable <V>	—	SPV	—	S	SRR0/1	7-43
IVOR33	Embedded floating-point data <SP.FD,SP.FS,SP.FV>		—	SPV	—	S	SRR0/1	7-44
	AltiVec assist <V>		—	SPV	—	S	SRR0/1	7-45
IVOR34 <SP.FD,SP.FS, SP.FV>	Embedded floating-point round		—	SPV	—	S	SRR0/1	7-46
IVOR35 <E.PM>	Performance monitor		MSR[GS] = 0 or EPCR[PMGS] = 0	—	MSR[EE] or MSR[GS]	A	SRR0/1	7-47
IVOR36 <E.PC>	Processor doorbell		—	—	MSR[EE] or MSR[GS]	A	SRR0/1	7-48
IVOR37 <E.PC>	Processor doorbell critical		—	—	MSR[CE] or MSR[GS]	A	CSRR0/1	7-48
IVOR38 <E.HV>	Guest processor doorbell		—	—	MSR[EE] and MSR[GS]	A	GSRR0/1	7-49
IVOR39 <E.HV>	Guest processor doorbell critical		—	—	MSR[CE] and MSR[GS]	A	CSRR0/1	7-50
IVOR39 <E.HV>	Guest processor doorbell machine check		—	—	MSR[ME] and MSR[GS]	A	CSRR0/1	7-51
IVOR40 <E.HV>	Embedded hypervisor system call		—	—	—	S*	SRR0/1	7-51
IVOR41 <E.HV>	Embedded hypervisor privilege		—	—	—	S	SRR0/1	7-52

Note:

[1] In general, when an interrupt affects an ESR as indicated in the table, it also causes all other ESR bits to be cleared. Special rules may apply for implementation-specific ESR bits.

Legend:
xxx (no brackets) means ESR[*xxx*] is set.
[xxx] means ESR[*xxx*] could be set.
[xxx,yyy] means either ESR[*xxx*] or ESR[*yyy*] may be set, but not both.
{xxx,yyy} means either ESR[*xxx*] or ESR[*yyy*] and possibly both may be set.

[2] Interrupt types:
S = synchronous
A = asynchronous
* = post completion interrupt. *x*SRR0 registers point after the instruction causing the exception.

[3] Although it is not specified, it is common for system implementations to provide, as part of the interrupt controller, independent mask and status bits for the various sources of critical input and external input interrupts.

[4] This debug interrupt may be made pending if MSR[DE] = 0 at the time of the exception

7.8.1 Critical Input Interrupt

A critical input interrupt occurs when no higher priority exception exists (see Section 7.11, “Exception Priorities”), a critical input exception is presented to the interrupt mechanism, and MSR[CE] = 1 (or MSR[GS] = 1 <E.HV>). Specific critical input exceptions are implementation-dependent but they are typically caused by assertion of an asynchronous signal that is part of the system. In addition to MSR[CE], implementations may provide other ways to mask this interrupt.

CSRR0, CSRR1, and MSR are updated as shown in this table.

Table 7-5. Critical Input Interrupt Register Settings

Register	Setting
CSRR0	Set to the effective address of the next instruction to be executed
CSRR1	Set to the MSR contents at the time of the interrupt
MSR	• CM is set to EPCR[ICM] <64> • DE is unchanged <E.ED> • ME,RI are unchanged • All other defined MSR bits are cleared

Instruction execution resumes at address IVPR[0–47] || IVOR0[48–59] || 0b0000.

Critical interrupt input signals are level sensitive. To guarantee that the core can take a critical input interrupt, the critical input interrupt signal must be asserted until the interrupt is taken. Otherwise, whether the core takes a critical interrupt depends on whether MSR[CE] (or MSR[GS] <E.HV>) is set when the critical interrupt signal is asserted.

NOTE: Software Considerations

To avoid redundant critical input interrupts, software must take any actions required by the implementation to clear any critical input exception status before reenabling MSR[CE] (or setting MSR[GS] <E.HV>).

7.8.2 Machine Check, NMI, and Error Report Interrupts

A machine check, NMI, or error report interrupt occurs when no higher priority exception exists (see Section 7.11, “Exception Priorities”), a machine check, NMI, or error report exception is presented to the interrupt mechanism, and for machine check interrupts MSR[ME] = 1 (or MSR[GS] = 1 <E.HV>).

The Machine Check Interrupt consists of three different, but related, types of exception conditions that all use the same interrupt vector and same interrupt registers. The three different interrupts are:

- Asynchronous machine check exceptions which are the result of error conditions directly detected by the processor or as a result of the assertion of a machine check signal pin from the integrated device (typically described in the integrated device reference manual as the *mcp* signal) as described by Section 7.8.2.4, “Asynchronous Machine Check Interrupts.”
- Synchronous error report exceptions which are the result of an instruction encountering an error condition, but execution cannot continue without propagating data derived from the error condition as described in Section 7.8.2.3, “Error Report Synchronous Interrupts.”
- Non-maskable (NMI) interrupts which are non-maskable, non-recoverable interrupts that are signaled from the integrated device as described by Section 7.8.2.2, “NMI Interrupts.”

For all of these interrupts, the following occur:

- MCSRR0 and MCSRR1 save the return address and MSR.
- An address related to the machine check may be stored in MCAR (and MCARU).
- The machine check syndrome register, MCSR, is used to log information about the error condition. The MCSR is described in Section 3.8.11, “Machine Check Syndrome Register (MCSR).”
- At the end of the machine check interrupt software handler, a Return from Machine Check Interrupt (**rfmci**) may be used to return to the state saved in MCSRR0 and MCSRR1.

Machine check exceptions are typically caused by a hardware failure or by software performing actions for which the hardware has not been designed to handle, or cannot provide a suitable result. They may be caused indirectly by execution of an instruction, but may not be recognized or reported until long after the processor has executed that instruction.

Machine check and error report conditions are implementation dependent, consult the core reference manual for device specific information.

NOTE: Software Considerations

> Some older processors do not implement error report or NMI interrupts. In these processors, machine check interrupts may be imprecise. If MSR[ME] is zero, the processor enters checkstop state immediately on detecting a machine check condition. Consult the core reference manual.

7.8.2.1 General Machine Check, Error Report, and NMI Mechanism

Asynchronous machine check, error report machine check, and NMI exceptions are independent of each other, even though they share the same interrupt vector.

7.8.2.1.1 Error Detection and Reporting Overview

The general flow of error detection and reporting occurs as follows:

- When the processor detects an error directly (i.e. the error occurs within the processor) or the machine check signal is asserted, the error is posted to the MCSR by setting an error status bit corresponding to the error that was detected. If the error bit set in the MCSR is one of the asynchronous machine check error conditions, an asynchronous machine check will occur when MSR[ME] = 1 (or MSR[GS] = 1 <E.HV>). Note that an asynchronous machine check interrupt will always occur when the asynchronous machine check interrupt is enabled and any of the asynchronous error bits in the MCSR are non-zero.

- If an instruction is a consumer of data associated with the error, the instruction has an error report exception associated with the instruction ensuring that, if the instruction reaches the point of completion, the instruction will take an error report machine check interrupt to prevent the erroneous data from propagating.
- It is possible that a single error within the processor will set both an asynchronous machine check error condition in the MCSR, and will associate an error report with the instruction that consumed data associated with the error. The asynchronous error bit will always be set, and if this triggers an asynchronous machine check interrupt before the instruction that has the error report exception completes, the asynchronous machine check interrupt will flush the instruction with the error report, and the error report will not occur. Likewise, if the instruction with the error report exception attempts to complete before the asynchronous error bit is set in MCSR, the error report machine check interrupt will be taken. In this case, the processor will still set the MCSR asynchronous error bit, probably well before software has read the MCSR. When software reads the MCSR, it will appear that both an asynchronous machine check exception and a synchronous error report occurred, because the error report will have caused the error report bits to be set, and the processor will also have set an asynchronous machine check error bit. This can easily happen if the error occurs when MSR[ME] = 0 (and MSR[GS] = 0 <E.HV>) because the asynchronous machine check interrupt will not be enabled.
- It is also possible that an error report machine check interrupt will occur without an associated asynchronous machine check error bit being set in the MCSR. This can occur when the processor is the consumer of some data for which the error was detected by some agent other than the processor. For example, an error in DRAM may occur and if the processor executed a load instruction which accessed that DRAM where the error occurred, the load instruction would take an error report machine check interrupt if it attempted to complete execution.
- A non-maskable interrupt (NMI) occurs when the integrated device asserts the NMI signal to the processor. The MCSR[NMI] bit is set when the interrupt occurs. The NMI signal is non-maskable and occurs regardless of the state of any interrupt enabling bits in the MSR.

7.8.2.1.2 Machine Check and Non-Maskable Interrupts Considerations

NOTE

An asynchronous machine check interrupt is taken when any of the asynchronous machine check error bits is not zero and the asynchronous machine check interrupt is enabled. The condition persists until software clears the asynchronous machine check error bits in MCSR.

To avoid multiple asynchronous machine check interrupts, software should always read the contents of the MCSR within the asynchronous machine check interrupt handler and clear any set bits in the MCSR prior to re-enabling machine check interrupts by setting MSR[ME].

NOTE

The processor may set asynchronous machine check error bits in MCSR at any time, as errors are detected, including when the processor is in the asynchronous machine check interrupt handler and MSR[ME] = 0.

An asynchronous machine check, error report, or NMI interrupt occurs when no higher priority interrupt exists and an asynchronous machine check, error report, or NMI exception is presented to the interrupt mechanism.

The following general rules apply:

- The instruction whose address is recorded in MCSRR0 has not completed, but may have attempted to execute.
- No instruction after the one whose address is recorded in MCSRR0 has completed execution.
- Instructions in the architected instruction stream prior to this instruction have all completed successfully.

When a machine check interrupt is taken, registers are updated as shown in this table.

Table 7-6. Machine Check Interrupt Settings

Register	Setting
MCSRR0	The core sets this to an EA of an instruction executing or about to execute when the exception occurred.
MCSRR1	Set to the contents of the MSR at the time of the exception.
MSR	• CM is set to EPCR[ICM] <64> • RI is cleared. • All other defined MSR bits are cleared. **Note:** Software should set MSR[RI] in the machine check and error report interrupt handler after MCSRR0/1 are successfully saved to the stack. When returning from an error interrupt, software should examine MCSRR1[RI] to determine whether the error report occurred prior to MCSRR0/1 being successfully saved. If MCSRR1[RI] is not set, the return path has been overwritten and the error is non-recoverable.
MCAR (MCARU)	MCAR is updated with the address of the data associated with the machine check. See Section 3.8.12, “Machine Check Address Register (MCAR/MCARU).”
MCSR	Set according to the machine check condition. See Table 3-32.

Instruction execution resumes at address IVPR[0–47] \|\| IVOR1[48–59] \|\| 0b0000.

NOTE: Software Considerations

For implementations on which a machine check interrupt is caused by referring to an invalid physical address, executing **dcbz**, **dcbzep**<E.PD>, **dcbzlep**<E.PD,DEO>, **dcbzl**<DEO>, **dcba**, or **dcbal**<DEO> can ultimately cause a machine check interrupt long after the instruction executed by establishing a data cache block associated with an invalid physical address. The interrupt can occur later on in an attempt to write that block to main memory, for example, as the result of executing an instruction that causes a cache miss for which the block is the target for replacement or as the result of executing **dcbst** or **dcbf**.

7.8.2.2 NMI Interrupts

Non-maskable interrupt exceptions cause an interrupt on the machine check vector. A non-maskable interrupt occurs when the integrated device asserts the *nmi* signal to the _CORE_. The *nmi* signal is non-maskable and occurs regardless of the state of any MSR interrupt enable. Software should clear the NMI bit in MCSR after the NMI interrupt has been taken before setting MSR[ME] (or MSR[GS] <E.HV>).

NMI interrupts are by definition non-recoverable since the interrupt occurs asynchronously and the interrupt cannot be masked by software. Unrecoverability can occur if the NMI occurs while the processor is in the early part of an asynchronous machine check, error report machine check, or another NMI interrupt handler and the return state in MCSRR0 and MCSRR1 have not yet been saved by software. It is possible for software to use MSR[RI] to determine whether software believes it is safe to return, but the system designer must allow for the case for which MCSRR0 and MCSRR1 have not been saved.

7.8.2.3 Error Report Synchronous Interrupts

Error report machine checks are intended to limit the propagation of bad data. For example, if a cache parity error is detected on a load, the load instruction is not allowed to *complete*, a synchronous error report machine check is generated, and the MCSRR0 holds the address of the load instruction with which the parity error is associated.

Preventing the load instruction from completing prevents the bad data from reaching the GPRs and prevents any subsequent instructions dependent on that data from executing. Error reports do not indicate the source of the problem (such as the cache parity error in the current example); the source is indicated by an asynchronous machine check. When an error report type of machine check occurs, the MCSR indicates the operation that incurred the error as follows:

- Instruction fetch error report (MCSR[IF]). An error occurred while attempting to fetch the instruction corresponding to the address contained in MCSRR0.
- Load instruction error report (MCSR[LD]). An error occurred while attempting to execute the load instruction corresponding to the address contained in MCSRR0.
- Guarded load instruction error report (MCSR[LDG]). If LD is set and the load was a guarded load (i.e. has the *guarded* storage attribute), this bit may be set. Note that some implementations may have specific conditions that govern when this bit is set.
- Store instruction error report (MCSR[ST]). An error occurred while attempting to perform address translation on the instruction corresponding to the address contained in MCSRR0. Since stores may complete with respect to the processor pipeline before their effects are seen in all memory subsystem areas, only translation errors are reported as error reports with stores. Note that some instructions which are considered load instructions with respect to permission checking and debug events are reported as store error reports (MCSR[ST] is set). See the core reference manual for which instructions set MCSR[LD] or MCSR[ST].

Error reports are intended to be a mechanism to stop the propagation of bad data; the asynchronous machine check is intended to allow software to attempt to recover from errors gracefully.

NOTE: Software Considerations

> In a multicore system, the integrated device is likely to steer all error interrupts to one processor. For example, assume that Processor B performs a load that causes an error in the integrated device, and the integrated device steers the error signal to Processor A's machine check input signal. Here, the error report in Processor B prevents the propagation of bad data; Processor A gets the task of attempting a graceful recovery. Some interprocessor communication is likely necessary.

An error report occurs only if the instruction that encountered the error reaches the bottom of the completion buffer (i.e. it becomes the oldest instruction currently in execution) and the instruction would have completed otherwise. If the instruction is flushed (possibly due to a mispredicted branch or asynchronous interrupt, including an asynchronous machine check) before reaching the bottom of the completion buffer, the error report does not occur.

7.8.2.4 Asynchronous Machine Check Interrupts

An asynchronous machine check occurs only when MSR[ME] = 1 (or MSR[GS] = 1 <E.HV>) and an MCSR asynchronous error bit is set. Because MSR[ME] and MSR[GS] are cleared whenever an error report interrupt occurs, a synchronous error report interrupt may clear MSR[ME] and MSR[GS] before the MCSR error bit is posted. If the error report handler clears the MCSR error bit before setting MSR[ME] (or MSR[GS] <E.HV>), no asynchronous machine check interrupt occurs.

7.8.3 Data Storage Interrupt

A data storage interrupt (DSI) occurs when no higher priority exception exists and a data storage exception is presented to the interrupt mechanism.

This table describes DSI exception conditions.

Table 7-7. Data Storage Interrupt Exception Conditions

Exception	Cause
Read access control exception	Occurs when one of the following conditions exists: • In user mode (MSR[PR] = 1), a load or load-class cache management instruction attempts to access a memory location that is not user-mode read enabled (page access control bit UR = 0). • In supervisor mode (MSR[PR] = 0), a load or load-class cache management instruction attempts to access a location that is not supervisor-mode read enabled (page access control bit SR = 0).
Write access control exception	Occurs when either of the following conditions exists: • In user mode (MSR[PR] = 1), a store or store-class cache management instruction attempts to access a location that is not user-mode write enabled (page access control bit UW = 0). • In supervisor mode (MSR[PR] = 0), a store or store-class cache management instruction attempts to access a location that is not supervisor-mode write enabled (page access control bit SW = 0).
Virtualization fault <E.HV>	TLB[VF] is set in the TLB entry that translated the storage access. A Virtualization Fault exception will occur when a Load, Store, or Cache Management Instruction attempts to access a location in storage that has the virtualization fault (VF) bit set in the TLB entry. A data storage interrupt resulting from a virtualization fault exception is always directed to hypervisor state regardless of the setting of EPCR[DSIGS].

Table 7-7. Data Storage Interrupt Exception Conditions (continued)

Exception	Cause
Byte-ordering exception	The implementation cannot access data in the byte order specified by the page's endian attribute. **Note:** • The byte-ordering exception is provided to assist implementations that cannot support dynamically switching byte ordering between consecutive accesses, the byte order for a class of accesses, or misaligned accesses using a specific byte order. • Load/store accesses that cross a page boundary such that endianness changes cause a byte-ordering exception.
Cache locking exception <E.CL>	Setting and clearing cache locks can be restricted to supervisor mode only access. MSR[UCLE] = 0 and MSR[PR] = 1 and execution of a **dcbtls**, **dcbtstls**, **icbtls**, **dcblc**, **dcblq.**, **icblq.**, or **icblc** instruction was attempted.
Storage synchronization exception	Occurs when the following condition exists: • An attempt is made to execute a load and reserve or store conditional instruction from or to memory that is write-through required or caching inhibited. (If no interrupt occurs, the instruction executes correctly.) See Section 4.6.1.16, "Atomic Update Primitives Using **lwarx** and **stwcx.**."

Instructions **icbt**, **dcbt**, **dcbtst**, **dcbtep**<E.PD>, **dcbtstep**<E.PD>, **dcba**, and **dcbal**<DEO> cannot cause a data storage interrupt, regardless of the effective address.

NOTE: Software Considerations

icbi, **icbtls**<E.CL>, **icblc**<E.CL>, **icblq.**<E.CL>, and **icbt** are treated as loads from the addressed byte with respect to address translation and protection. They use MSR[DS], not MSR[IS], to determine translation for their operands. Instruction storage interrupts and instruction TLB error interrupts are associated with instruction fetching and not execution. Data storage interrupts and data TLB error interrupts are associated with the execution of instruction cache management instructions.

When a DSI occurs, the processor suppresses execution of the instruction causing the exception.

<E.HV>:
The interrupt is directed to the hypervisor unless the following conditions exist determined as follows:

- The exception is not a virtualization fault (TLB[VF] = 0).
- The state in which the exception occurred is the guest state (MSR[GS] = 1).
- The interrupt is programmed to be directed to the guest state (EPCR[DSIGS] = 1).

If all the above conditions are met, the DSI is directed to the guest supervisor state. If the interrupt is directed to the guest supervisor state, GSRR0, GSRR1, GESR, MSR, and GDEAR, are updated as shown in this table.

Table 7-8. Data Storage Interrupt Register Settings (directed to guest)

Register	Setting
GSRR0	Set to the effective address of the interrupt-causing instruction
GSRR1	Set to the MSR contents at the time of the interrupt

Table 7-8. Data Storage Interrupt Register Settings (continued)(directed to guest)

Register	Setting
GESR	FP Set if the interrupt-causing instruction is a floating-point load or store; otherwise cleared <FP> ST Set if the interrupt-causing instruction is a store or store-class cache management instruction; otherwise cleared DLK Set if the interrupt-causing instruction is a **dcbtls**, **dcbtstls**, **dcblq.**, or **dcblc** is executed in user mode and MSR[UCLE] = 0; otherwise cleared <E.CL> ILK Set if the interrupt-causing instruction is a **icbtls**, **icblq.**, or **icblc** is executed in user mode and MSR[UCLE] = 0; otherwise cleared <E.CL> SPV Set if the interrupt-causing instruction is an SPE or AltiVec load or store; otherwise cleared <SP or V> BO Set if the interrupt-causing instruction encountered a byte-ordering exception; otherwise cleared VLEMI Set if the interrupt-causing instruction resides in storage with the VLE attribute; otherwise cleared <VLE> EPID Set if the interrupt-causing instruction is an external PID instruction; otherwise cleared <E.PD> All other defined ESR bits are cleared.
MSR	• CM is set to EPCR[GICM] <64> • GS, RI, ME, DE, CE are unchanged • UCLE is cleared if MSRP[UCLEP] = 0, otherwise it is unchanged • PMM is cleared if MSRP[PMMP] = 0, otherwise it is unchanged • All other defined MSR bits are cleared.
GDEAR	Updated with the EA of the access that caused the exception. This is generally the EA of the instruction, except for some instructions that are misaligned or reference more than a single storage element.

If category E.HV is not implemented or the interrupt is directed to the hypervisor state, SRR0, SRR1, ESR, MSR, and DEAR, are updated as shown in this table.

Table 7-9. Data Storage Interrupt Register Settings (directed to hypervisor)

Register	Setting
SRR0	Set to the effective address of the interrupt-causing instruction
SRR1	Set to the MSR contents at the time of the interrupt
ESR	FP Set if the interrupt-causing instruction is a floating-point load or store; otherwise cleared <FP> ST Set if the interrupt-causing instruction is a store or store-class cache management instruction; otherwise cleared DLK Set if the interrupt-causing instruction is a **dcbtls**, **dcbtstls**, **dcblq.**, or **dcblc** is executed in user mode and MSR[UCLE] = 0; otherwise cleared <E.CL> ILK Set if the interrupt-causing instruction is a **icbtls**, **icblq.**, or **icblc** is executed in user mode and MSR[UCLE] = 0; otherwise cleared <E.CL> SPV Set if the interrupt-causing instruction is an SPE or AltiVec load or store; otherwise cleared <SP or V> BO Set if the interrupt-causing instruction encountered a byte-ordering exception; otherwise cleared VLEMI Set if the interrupt-causing instruction resides in storage with the VLE attribute; otherwise cleared <VLE> EPID Set if the interrupt-causing instruction is an external PID instruction; otherwise cleared <E.PD> All other defined ESR bits are cleared.

Table 7-9. Data Storage Interrupt Register Settings (continued)(directed to hypervisor)

Register	Setting
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared.
DEAR	Updated with the EA of the access that caused the exception. This is generally the EA of the instruction, except for some instructions that are misaligned or reference more than a single storage element.

If category E.HV is implemented and the interrupt is directed to the guest supervisor state, instruction execution resumes at address GIVPR[0–47] || GIVOR2[48–59] || 0b0000. Otherwise, instruction execution resumes at address IVPR[0–47] || IVOR2[48–59] || 0b0000.

7.8.4 Instruction Storage Interrupt

An instruction storage interrupt (ISI) occurs when no higher priority exception exists and an instruction storage exception is presented to the interrupt mechanism.

Exception conditions are described in this table.

Table 7-10. Instruction Storage Interrupt Exception Conditions

Exception	Cause
Execute access control exception	Occurs when one of the following conditions exists: • In user mode (MSR[PR] = 1), an instruction fetch attempts to access a memory location that is not user-mode execute enabled (page access control bit UX = 0). • In supervisor mode (MSR[PR] = 0), an instruction fetch attempts to access a memory location that is not supervisor-mode execute enabled (page access control bit SX = 0). •
Byte-ordering exception	The implementation cannot fetch the instruction in the byte order specified by the page's endian attribute.
Misaligned instruction storage exception <VLE>	An attempt is made to execute an instruction that is not 32-bit aligned (bit 62 of the instruction addresses is set), and the VLE attribute is not set for that page of storage.

When an ISI occurs, the processor suppresses execution of the instruction causing the exception.

If category E.HV is implemented and the interrupt is directed to the guest supervisor state, GSRR0, GSRR1, MSR, and GESR, are updated as shown in this table.

Table 7-11. Instruction Storage Interrupt Register Settings (directed to guest)

Register	Setting
GSRR0	Set to the effective address of the interrupt-causing instruction
GSRR1	Set to the MSR contents at the time of the interrupt

Table 7-11. Instruction Storage Interrupt Register Settings (continued)(directed to guest)

Register	Setting
GESR	BO Set if the interrupt-causing instruction encountered a byte-ordering exception or a misaligned instruction storage exception <VLE>; otherwise cleared VLEMI Set if the interrupt-causing instruction resides in storage with the VLE attribute; otherwise cleared <VLE> All other defined ESR bits are cleared.
MSR	• CM is set to EPCR[GICM] <64> • GS, RI, ME, DE, CE are unchanged • UCLE is cleared if MSRP[UCLEP] = 0, otherwise it is unchanged • PMM is cleared if MSRP[PMMP] = 0, otherwise it is unchanged • All other defined MSR bits are cleared.

If category E.HV is not implemented or the interrupt is directed to the hypervisor state, SRR0, SRR1, MSR, and ESR, are updated as follows:

Table 7-12. Instruction Storage Interrupt Register Settings (directed to hypervisor)

Register	Setting
SRR0	Set to the effective address of the interrupt-causing instruction
SRR1	Set to the MSR contents at the time of the interrupt
ESR	BO Set if the interrupt-causing instruction encountered a byte-ordering exception or a misaligned instruction storage exception <VLE>; otherwise cleared VLEMI Set if the interrupt-causing instruction resides in storage with the VLE attribute; otherwise cleared <VLE> All other defined ESR bits are cleared.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared.

NOTE: Software Considerations

> Permissions violation and byte-ordering exceptions are not mutually exclusive. Even if ESR[BO] is set, system software must examine the TLB entry accessed by the fetch to determine whether a permissions violation also occurred.

If category E.HV is implemented and the interrupt is directed to the guest supervisor state, instruction execution resumes at address GIVPR[0–47] || GIVOR3[48–59] || 0b0000. Otherwise, instruction execution resumes at address IVPR[0–47] || IVOR3[48–59] || 0b0000.

7.8.5 External Input Interrupt

An external input interrupt occurs when no higher priority exception exists (see Section 7.11, “Exception Priorities”), an external input exception is presented to the interrupt mechanism, and MSR[EE] = 1 (or MSR[GS] = 1 and EPCR[EXTGS] = 0 <E.HV>). Specific external input exceptions are implementation-dependent but they are typically caused by assertion of an asynchronous signal that is part of the system. In addition to MSR[EE], implementations may provide other ways to mask this interrupt.

If the category EXP is implemented and enabled in the integrated device, the source of the interrupt is placed in the EPR (or GEPR <E.HV>) register and the interrupt is acknowledged and placed *in-service*. See Section 7.8.5.1, "External Proxy <EXP>.

If category E.HV is implemented and the interrupt is directed to the guest supervisor state, GSRR0, GSRR1, GEPR <EXP>, and MSR, are updated as shown in this table.

Table 7-13. External Input Interrupt Register Settings (directed to guest)

Register	Setting
GSRR0	Set to the effective address of the next instruction to be executed
GSRR1	Set to the MSR contents at the time of the interrupt
MSR	• CM is set to EPCR[GICM] <64> • GS, RI, ME, DE, CE are unchanged • UCLE is cleared if MSRP[UCLEP] = 0, otherwise it is unchanged • PMM is cleared if MSRP[PMMP] = 0, otherwise it is unchanged • All other defined MSR bits are cleared
GEPR <EXP>	If external proxy interrupt delivery is configured in the external device, GEPR is set to the vector offset that identifies the source that generated the interrupt triggered from the integrated device interrupt controller.Otherwise, GEPR is set to all zeros.

If category E.HV is not implemented or the interrupt is directed to the hypervisor state, SRR0, SRR1, EPR <EXP>, and MSR are updated as shown in this table.

Table 7-14. External Input Interrupt Register Settings (directed to hypervisor)

Register	Setting
SRR0	Set to the effective address of the next instruction to be executed
SRR1	Set to the MSR contents at the time of the interrupt
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared
EPR <EXP>	If external proxy interrupt delivery is configured in the external device, EPR is set to the vector offset that identifies the source that generated the interrupt triggered from the integrated device interrupt controller.Otherwise, EPR is set to all zeros.

If category E.HV is implemented and the interrupt is directed to the guest supervisor state, instruction execution resumes at address GIVPR[0–47] || GIVOR4[48–59] || 0b0000. Otherwise, instruction execution resumes at address IVPR[0–47] || IVOR4[48–59] || 0b0000.

External interrupt input signals are level sensitive; to guarantee that the core can take an external input interrupt, the external input interrupt signal must be asserted until the interrupt is taken. Otherwise, whether the core takes an external interrupt depends on whether MSR[EE] is set (or MSR[GS] is set and EPCR[EXTGS] is not set <E.HV>) when the external interrupt signal is asserted.

NOTE: Software Considerations

To avoid redundant external input interrupts, software must take any actions required by the implementation to clear any critical input exception status before reenabling MSR[EE] (or setting MSR[GS] <E.HV>).

7.8.5.1 External Proxy <EXP>

The external proxy facility defines an interface for using a processor-to-interrupt controller hardware interface for acknowledging external interrupts from a programmable interrupt controller (PIC) implemented as part of the integrated device. This functionality is enabled through a register field defined by the PIC and documented in the reference manual for the integrated device.

Using this interface reduces the latency required to read and acknowledge the interrupt that normally requires a cache-inhibited guarded load to the memory controller.

In integrated devices without the external proxy facility, when the processor receives a signal from the PIC indicating that the external interrupt was necessary to handle a condition typically presented by an integrated peripheral device, the interrupt handler responds by reading a memory-mapped register (interrupt acknowledge, or IACK) defined by the Open PIC standard. The contents returned from the read of the IACK indicate the source of the interrupt as a vector offset. In addition to providing a vector offset specific to the peripheral device, this read negates the internal signal and changes the status of the interrupt request from *pending* to *in-service* in which state it would remain until the completion of the interrupt handling.

The external proxy eliminates the need to read the IACK register by presenting the vector to the external proxy register (EPR), or guest external proxy register (GEPR), described in Section 3.8.13, “External Proxy Register (EPR) <EXP>,” and Section 3.8.14, “Guest External Proxy Register (GEPR) <EXP,E.HV>.”

Instead of just signaling an external input interrupt, the PIC also provides the specific vector for the interrupt. When the interrupt is enabled and the PIC is asserting a vector, the interrupt occurs and the processor communicates to the PIC that the interrupt has been taken. The processor provides the vector from the PIC in the (G)EPR register, which software can then read. As part of the communication with the PIC, the PIC puts the specific interrupt *in-service* as if software had read the IACK register in the legacy method. The PIC always asserts the highest priority pending interrupt to the processor and the interrupt that is put in-service is determined by when the processor takes the interrupt based on the appropriate enabling conditions.From a system software perspective, the processor does not acknowledge the interrupt until the external input interrupt is taken.

Software in the external input interrupt handler would then read (G)EPR to determine the vector for the interrupt. The value of the vector in (G)EPR does not change until the next external input interrupt occurs and therefore software must read (G)EPR before re-enabling the interrupt.

NOTE: Software Considerations

When using external proxy (and even with the legacy method), software must ensure that end-of-interrupt (EOI) processing is synchronized with taking of external input interrupts such that the EOI indicator is received so that the interrupt controller can properly pair it with the source. For example, writing the EOI register for the PIC would require that the following sequence occur:

```
block interrupts;     // turn EE off for external interrupts
write EOI register;   // signal end of interrupt
read EOI register;    // ensure write has completed
unblock interrupts;   // allow interrupts
```

7.8.6 Alignment Interrupt

An alignment interrupt occurs when no higher priority exception exists and an alignment exception is presented to the interrupt mechanism. An alignment exception may occur when an implementation cannot perform a data access for one of the following reasons:

- The operand of a load or store is not aligned.
- The instruction is a load multiple or store multiple and an operand addresses memory that is little-endian.
- A **dcbz** type instruction (**dcbz**, **dcbzl** <DEO> , **dcbzep** <E.PD> , or **dcbzlep** <DEO,E.PD>) operand is in write-through-required or caching-inhibited memory, or **dcbz** type instruction is executed in an implementation with no data cache or a write-through data cache.
- The operand of a store conditional instruction, is in write-through required or caching-inhibited memory.
- A load multiple word or store multiple word instruction reads an address that is not a multiple of four.
- A load and reserve or store conditional instruction (**lharx**<ER>, **lwarx**, **ldarx**<64>, **sthcx.**<ER>, **stwcx.**, or **stdcx.**<64>) instruction references an address that is not aligned based on the number of bytes accessed. That is, for halfword accesses the address must be a multiple of two, word accesses a multiple of four, and doubleword accesses a multiple of eight.

NOTE: Software Considerations

The architecture does not support the use of an unaligned effective address by load and reserve and store conditional instructions. If an alignment interrupt occurs because one of these instructions specifies an unaligned effective address, the interrupt handler must treat the instruction as a programming error rather than attempt to emulate the instruction.

When an alignment interrupt occurs, the processor suppresses execution of the instruction causing the exception.

SRR0, SRR1, MSR, DEAR, and ESR are updated as shown in this table.

Table 7-15. Alignment Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the alignment interrupt
SRR1	Set to the MSR contents at the time of the interrupt
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared
DEAR	Updated with the EA of the access that caused the exception. This is generally the EA produced by the instruction, except for some instructions that are misaligned or that reference multiple storage elements.
ESR	FP Set if the interrupt-causing instruction is a floating-point load or store; otherwise cleared <FP> ST Set if the interrupt-causing instruction is a store or store-class cache management instruction; otherwise cleared SPV Set if the interrupt-causing instruction is an SPE or AltiVec load or store; otherwise cleared <SP or V> VLEMI Set if the interrupt-causing instruction resides in storage with the VLE attribute; otherwise cleared <VLE> EPID Set if the interrupt-causing instruction is an external PID instruction; otherwise cleared <E.PD> All other defined ESR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || IVOR5[48–59] || 0b0000.

7.8.7 Program Interrupt

A program interrupt occurs when no higher priority exception exists (see Section 7.11, “Exception Priorities”), a program exception is presented to the interrupt mechanism, and, for floating-point enabled exceptions, MSR[FE0,FE1] <FP> are non-zero.

A program exception is caused when any of the exceptions listed in this table arises during execution of an instruction.

Table 7-16. Program Interrupt Exception Conditions

Exception	Cause
Floating-point enabled <FP>	FPSCR[FEX] is set by the execution of a floating-point instruction that causes an enabled exception, including the case of a move to FPSCR instruction that causes both an exception bit and the corresponding enable bit to be 1. In this context, the term 'enabled exception' refers to the enabling provided by FPSCR control bits. See Section 3.4.2, "Floating-Point Status and Control Register (FPSCR)." MSR[FE0,FE1] determines whether a floating-point enabled interrupt is precise or imprecise, as follows: 01,10,11 Precise. 00 The interrupt is masked and the interrupt subsequently occurs if and when floating-point enabled exception-type program interrupts are enabled by setting either or both FE0,FE1 and also causes ESR[PIE] to be set.
Illegal instruction	Execution is attempted of any of the following kinds of instructions. • A reserved-illegal instruction • When MSR[PR]=1 (user mode), an **mtspr** or **mfspr** that specifies an SPRN value with SPRN[5]=0 (user-mode accessible) that represents an unimplemented SPR • A defined instruction that is not implemented (unimplemented operation exception) May occur when execution is attempted of any of the following kinds of instructions. If the exception does not occur, the alternative is shown in parentheses. • An instruction with an invalid form (boundedly undefined results) • A reserved-no-op instruction (no-operation performed is preferred) • When MSR[PR]=0 (supervisor mode), an **mtspr** or **mfspr** that specifies an SPRN value that represents an unimplemented SPR (boundedly undefined) • For 64-bit implementations, when the processor is in 32-bit mode and execution of a 64-bit instruction is attempted. (instruction executes normally)
Privileged instruction	MSR[PR]=1 and execution is attempted of any of the following kinds of instructions. • A privileged instruction • An **mtspr** or **mfspr** instruction that specifies an SPRN value with SPRN[5] = 1
Trap	Any of the conditions specified in **trap** are met and the exception is not also enabled as a debug interrupt, in which case a debug interrupt (that is DBCR0[TRAP]=1, DBCR0[IDM]=1, and MSR[DE]=1), a debug interrupt is taken instead of a program interrupt.
Unimplemented operation	May occur when execution is attempted of a defined, unimplemented instruction. Otherwise an illegal instruction exception occurs.

SRR0, SRR1, MSR, and ESR are updated as shown in this table.

Table 7-17. Program Interrupt Register Settings

Register	Description
SRR0	For all program interrupts except when a disabled floating-point exception is subsequently enabled, set to the EA of the instruction that caused the program interrupt. <FP>: When a disabled floating-point exception is subsequently enabled, SRR0 is set to the EA of the instruction that would have executed next, if the interrupt did not occur and ESR[PIE] is also set. The interrupt may occur at any time after the instruction that enables the floating-point exception but before and up to the next context synchronizing instruction (or event).
SRR1	Set to the contents of the MSR at the time of the interrupt.

Table 7-17. Program Interrupt Register Settings (continued)

Register	Description
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared
ESR	PIL Set if an illegal instruction exception type program interrupt; otherwise cleared PPR Set if a privileged instruction exception type program interrupt; otherwise cleared PTR Set if a trap exception type program interrupt; otherwise cleared PUO Set if an unimplemented operation exception type program interrupt; otherwise cleared PIE Set if a floating-point enabled exception type program interrupt, and the address saved in SRR0 is not the address of the instruction causing the exception (that is the instruction that caused FPSCR[FEX] to be set); otherwise cleared.<FP> FP Set if the interrupt-causing instruction is a floating-point load or store; otherwise cleared <FP> VLEMI Set if the interrupt-causing instruction resides in storage with the VLE attribute; otherwise cleared <VLE> All other defined ESR bits are cleared.

Instruction execution resumes at address IVPR[0–47] || IVOR6[48–59] || 0b0000.

7.8.8 Floating-Point Unavailable Interrupt <FP>

A floating-point unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point loads, stores, and moves), and MSR[FP] = 0.

When a floating-point unavailable interrupt occurs, the processor suppresses execution of the instruction causing the floating-point unavailable interrupt.

SRR0, SRR1, and MSR are updated as shown in this table.

Table 7-18. Floating-Point Unavailable Interrupt Register Settings

Register	Description
SRR0	Set to the effective address of the instruction that caused the interrupt
SRR1	Set to the MSR contents at the time of the interrupt
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared

Instruction execution resumes at address IVPR[0–47] || IVOR7[48–59] || 0b0000.

7.8.9 System Call Interrupt

A system call interrupt occurs when no higher priority exception exists and a System Call (**sc**) instruction with LEV = 0 is executed.

If category E.HV is implemented and the interrupt is directed to the guest supervisor state, GSRR0, GSRR1, and MSR are updated as shown in this table.

Table 7-19. System Call Interrupt Register Settings (directed to guest)

Register	Setting
GSRR0	Set to the effective address of the instruction after the **sc** instruction.
GSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[GICM] <64> • GS, RI, ME, DE, CE are unchanged • UCLE is cleared if MSRP[UCLEP] = 0, otherwise it is unchanged • PMM is cleared if MSRP[PMMP] = 0, otherwise it is unchanged • All other defined MSR bits are cleared

If category E.HV is not implemented or the interrupt is directed to the hypervisor state, SRR0, SRR1, and MSR are updated as shown in this table.

Table 7-20. System Call Interrupt Register Settings (directed to hypervisor)

Register	Setting
SRR0	Set to the effective address of the instruction after the **sc** instruction.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared

If category E.HV is implemented and the interrupt is directed to the guest supervisor state, instruction execution resumes at address GIVPR[0–47] || GIVOR8[48–59] || 0b0000. Otherwise, instruction execution resumes at address IVPR[0–47] || IVOR8[48–59] || 0b0000.

The system call interrupt is a post-completion synchronous interrupt.

7.8.10 Decrementer Interrupt

A decrementer interrupt occurs when no higher priority exception exists, a decrementer exception exists (TSR[DIS] = 1), and the interrupt is enabled (TCR[DIE] = 1 and (MSR[EE] = 1 or (MSR[GS] = 1 <E.HV>))). See Section 3.7.4, “Decrementer Register (DEC).”

NOTE: Software Considerations

MSR[EE] also enables other asynchronous interrupts.

TSR[DIS] is set when a decrementer exception exists.

SRR0, SRR1, and MSR, are updated as shown in this table.

Table 7-21. Decrementer Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared
TSR	DIS is set when a decrementer exception exists, not as a result of the interrupt. See Section 3.7.2, “Timer Status Register (TSR).”

Instruction execution resumes at address IVPR[0–47] || IVOR10[48–59] || 0b0000.

NOTE: Software Considerations

> To avoid redundant decrementer interrupts, before reenabling MSR[EE], the interrupt handler must clear TSR[DIS] by writing a word to TSR using **mtspr** with a 1 in any bit position to be cleared and 0 in all others. Data written to the TSR is not direct data, but a mask. Writing a 1 to this bit causes it to be cleared; writing a 0 has no effect.

7.8.11 Fixed-Interval Timer Interrupt

A fixed-interval timer interrupt occurs when no higher priority exception exists, a fixed-interval timer exception exists (TSR[FIS] = 1), and the interrupt is enabled (TCR[FIE] = 1 and (MSR[EE] = 1 or (MSR[GS] = 1 <E.HV>))). See Section 8.5, “Fixed-Interval Timer.”

The fixed-interval timer period is determined by TCR[FPEXT] || TCR[FP], which specifies one of 64 bit locations of the time base used to signal a fixed-interval timer exception on a transition from 0 to 1.

TCR[FPEXT] || TCR[FP] = 0b0000_00 selects TBU[0]. TCR[FPEXT] || TCR[FP] = 0b1111_11 selects TBL[63].

NOTE: Software Considerations

> MSR[EE] also enables other asynchronous interrupts.

TSR[FIS] is set when a fixed-interval timer exception exists.

SRR0, SRR1, and MSR, are updated as shown in this table.

Table 7-22. Fixed-Interval Timer Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.

Table 7-22. Fixed-Interval Timer Interrupt Register Settings (continued)

Register	Setting
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared
TSR	FIS is set when a fixed-interval timer exception exists, not as a result of the interrupt. See Section 3.7.2, “Timer Status Register (TSR).”

Instruction execution resumes at address IVPR[0–47] || IVOR11[48–59] || 0b0000.

NOTE: Software Considerations

> To avoid redundant fixed-interval timer interrupts, before reenabling MSR[EE], the interrupt handler must clear TSR[FIS] by writing a word to TSR using **mtspr** with a 1 in any bit position to be cleared and 0 in all others. Data written to the TSR is not direct data, but a mask. Writing a 1 to this bit causes it to be cleared; writing a 0 has no effect.

7.8.12 Watchdog Timer Interrupt

A watchdog timer interrupt occurs when no higher priority exception exists, a watchdog timer exception exists (TSR[WIS] = 1), and the interrupt is enabled (TCR[WIE] = 1 and (MSR[CE] = 1 or (MSR[GS] = 1 <E.HV>))). See Section 8.6, “Watchdog Timer.”

The watchdog timer period is determined by TCR[WPEXT] || TCR[WP], which specifies one of 64 bit locations of the time base used to signal a watchdog timer exception on a transition from 0 to 1.

TCR[WPEXT] || TCR[WP] = 0b0000_00 selects TBU[0]. TCR[WPEXT] || TCR[WP] = 0b1111_11 selects TBL[63].

NOTE: Software Considerations

> MSR[CE] also enables other asynchronous critical interrupts.

TSR[WIS] is set when a watchdog timer exception exists. CSRR0, CSRR1, and MSR, are updated as shown in this table.

Table 7-23. Watchdog Timer Interrupt Register Settings

Register	Setting
CSRR0	Set to the effective address of the next instruction to be executed.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME are unchanged • DE is unchanged <E.ED> • All other defined MSR bits are cleared
TSR	WIS is set when a watchdog timer exception exists, not as a result of the interrupt. See Section 3.7.2, “Timer Status Register (TSR).”

Instruction execution resumes at address IVPR[0–47] || IVOR12[48–59] || 0b0000.

NOTE: Software Considerations

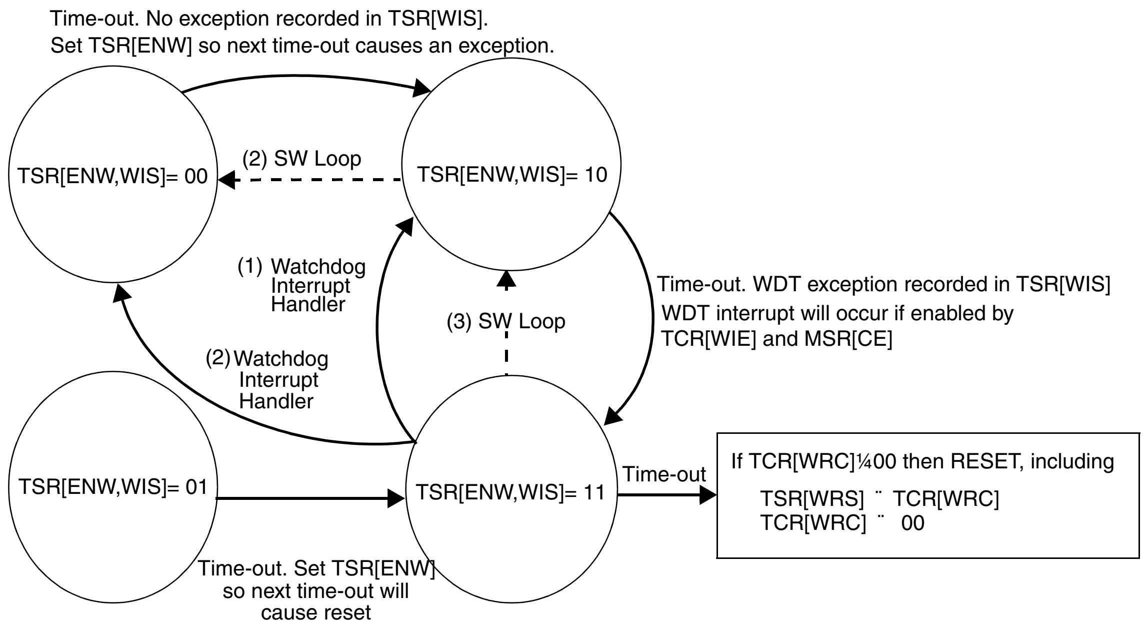
> To avoid redundant watchdog timer interrupts, before reenabling MSR[CE], the interrupt handling routine must clear TSR[WIS] by writing a word to TSR using **mtspr** with a 1 in any bit position to be cleared and 0 in all others. The data written to the TSR is not direct data, but a mask. Writing a 1 to this bit causes it to be cleared; writing a 0 has no effect.

The watchdog timer facility aids system recovery from faulty software or hardware. Watchdog time-outs occur on 0 to 1 transitions of selected bits from the time base (see Section 3.7.1, “Timer Control Register (TCR)”).

The following figure describes the watchdog timer state machine and watchdog timer controls. The numbers in parentheses in the figure refer to the discussion of modes of operation.

When a watchdog timer time-out occurs while watchdog timer interrupt status is clear (TSR[WIS] = 0) and the next watchdog time-out is enabled (TSR[ENW] = 1), a watchdog timer exception is generated and logged by setting TSR[WIS]. This is referred to as a watchdog timer first time out. A watchdog timer interrupt occurs if it is enabled by TCR[WIE] and MSR[CE] (or MSR[GS]<E.HV>). The purpose of the first time-out is to indicate a potential problem so the system can perform corrective action or capture a failure before a reset occurs from the watchdog timer second time-out. (Note that a watchdog timer exception also occur if an **mtspr** writes a 1 to the selected bit when its previous value was 0.

A second time-out occurs if TSR[WIS] = 1 and TSR[ENW] = 1 and a processor reset occurs if it is enabled by a nonzero value of the watchdog reset control field (TCR[WRC]). The assumption is that TSR[WIS] was not cleared because the processor could not execute the watchdog timer interrupt handler, leaving reset as the only way to restart the system. Note that once TCR[WRC] is set to a nonzero value, it cannot be reset by software; this feature prevents errant software from disabling the watchdog timer reset capability.


Time-out. No exception recorded in TSR[WIS].
Set TSR[ENW] so next time-out causes an exception.
TSR[ENW,WIS]= 00
(2) SW Loop
TSR[ENW,WIS]= 10
(1) Watchdog Interrupt Handler
Time-out. WDT exception recorded in TSR[WIS]
WDT interrupt will occur if enabled by TCR[WIE] and MSR[CE]
(3) SW Loop
(2) Watchdog Interrupt Handler
TSR[ENW,WIS]= 01
TSR[ENW,WIS]= 11
Time-out
If TCR[WRC]¼00 then RESET, including
TSR[WRS] ¨ TCR[WRC]
TCR[WRC] ¨ 00
Time-out. Set TSR[ENW] so next time-out will cause reset

Enable Next WDT (TSR[ENW])	WDT Status (TSR[WIS])	Action when timer interval expires
0	0	Set enable next watchdog timer (TSR[ENW]=1).
0	1	Set enable next watchdog timer (TSR[ENW]=1).
1	0	Set watchdog timer interrupt status bit (TSR[WIS]=1). If watchdog timer interrupt is enabled (TCR[WIE]=1 and MSR[CE]=1), then interrupt.
1	1	Cause watchdog timer reset action specified by TCR[WRC]. Reset will copy pre-reset TCR[WRC] into TSR[WRS], then clear TCR[WRC].

Figure 7-1. Watchdog State Machine

The controls described in the table imply three programmable modes, each of which assumes that TCR[WRC] has been set to allow processor reset by the watchdog facility:

1. Always take the watchdog timer interrupt when pending and never attempt to prevent its occurrence—the watchdog timer interrupt caused by a first time-out is used to clear TSR[WIS] so a second time-out never occurs. TSR[ENW] is not cleared, allowing the next time-out to cause another interrupt.
2. Always take the watchdog timer interrupt when pending, but avoid when possible—To avoid a first time-out exception, a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the fixed-interval timer interrupt handler) repeatedly clears TSR[ENW] so no watchdog timer interrupt occurs. If TSR[ENW] is cleared, software has between one and two full watchdog periods before a watchdog exception will be posted in TSR[WIS]. If this occurs before

software can clear TSR[ENW] again, a watchdog timer interrupt occurs. In this case, the watchdog timer interrupt handler then clears both TSR[ENW,WIS], hopefully avoiding the next watchdog timer interrupt.

3. Never take the watchdog timer interrupt—To avoid a reset due to a second time-out, watchdog timer interrupts are disabled (TCR[WIE]=0) and the system depends on a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the fixed-interval timer interrupt handler) to repeatedly clear TSR[WIS]. TSR[ENW] is not cleared, allowing the next time-out to set TSR[WIS] again. The loop must have a period shorter than one watchdog timer period.

7.8.13 Data TLB Error Interrupt

A data TLB error interrupt occurs when no higher priority exception exists and a data TLB miss exception (a virtual address associated with a data storage access do not match any valid TLB entry as specified in Section 6.5.4.1, “Match Criteria for TLB Entries”) is presented to the interrupt mechanism.

If a store conditional instruction would not perform its store in the absence of a data storage interrupt and an unconditional store to the specified effective address would cause a data storage interrupt, a data storage interrupt occurs.

When a data TLB error interrupt occurs, the processor suppresses execution of the instruction causing the data TLB error exception.

If category E.HV is implemented and the interrupt is directed to the guest supervisor state, GSRR0, GSRR1, GDEAR, GESR, MAS*n*, and MSR are updated as shown in this table.

Table 7-24. Data TLB Error Interrupt Register Settings (directed to guest)

Register	Setting
GSRR0	Set to the effective address of the instruction causing the data TLB error interrupt.
GSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[GICM] <64> • GS, RI, ME, DE, CE are unchanged • UCLE is cleared if MSRP[UCLEP] = 0, otherwise it is unchanged • PMM is cleared if MSRP[PMMP] = 0, otherwise it is unchanged • All other defined MSR bits are cleared
GDEAR	Set to the EA of a byte that is both within the range of the bytes being accessed by the storage access or cache management instruction, and within the page whose access caused the data TLB error exception.
GESR	ST Set if the interrupt-causing instruction is a store, **dcbi, dcbz**, **dcbzep** <E.PD>, **dcbzl** <DEO>, or **dcbzlep** <DEO,E.PD> instruction; otherwise cleared. FP Set if the interrupt-causing instruction is a floating-point load or store; otherwise cleared.<FP> SPV Set if the interrupt-causing instruction is an SPE or AltiVec load or store; otherwise cleared.<SP or V> VLEMI Set if the interrupt-causing instruction resides in VLE storage.<VLE> EPID Set if the interrupt-causing instruction is an external process ID instruction; otherwise cleared.<E.PD> All other defined ESR bits are cleared.
MAS*n*	See Table 6-11.

If category E.HV is not implemented or the interrupt is directed to the hypervisor state, SRR0, SRR1, DEAR, ESR, MAS*n*, and MSR are updated as shown in this table.

Table 7-25. Data TLB Error Interrupt Register Settings (directed to hypervisor)

Register	Setting
SRR0	Set to the effective address of the instruction causing the data TLB error interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared
DEAR	Set to the EA of a byte that is both within the range of the bytes being accessed by the storage access or cache management instruction, and within the page whose access caused the data TLB error exception.
ESR	ST Set if the interrupt-causing instruction is a store, **dcbi, dcbz**, **dcbzep** <E.PD>, **dcbzl** <DEO>, or **dcbzlep** <DEO,E.PD> instruction; otherwise cleared. FP Set if the interrupt-causing instruction is a floating-point load or store; otherwise cleared.<FP> SPV Set if the interrupt-causing instruction is an SPE or AltiVec load or store; otherwise cleared.<SP or V> VLEMI Set if the interrupt-causing instruction resides in VLE storage.<VLE> EPID Set if the interrupt-causing instruction is an external process ID instruction; otherwise cleared.<E.PD> All other defined ESR bits are cleared.
MAS*n*	See Table 6-11.

Table 6-11 shows MAS register settings for data and instruction TLB error interrupts. For more information, see Section 6.5.8.3, “MAS Register Updates for Exceptions, **tlbsx**, and **tlbre**.”

If category E.HV is implemented and the interrupt is directed to the guest supervisor state, instruction execution resumes at address GIVPR[0–47] || GIVOR13[48–59] || 0b0000. Otherwise, instruction execution resumes at address IVPR[0–47] || IVOR13[48–59] || 0b0000.

7.8.14 Instruction TLB Error Interrupt

An instruction TLB error interrupt occurs when no higher priority exception exists and an instruction TLB miss exception (the virtual address associated with a fetch does not match any valid TLB entry as specified in Section 6.5.4.1, “Match Criteria for TLB Entries”) is presented to the interrupt mechanism.

When an instruction TLB error interrupt occurs, the processor suppresses execution of the instruction causing the instruction TLB error exception.

If category E.HV is implemented and the interrupt is directed to the guest supervisor state, GSRR0, GSRR1, MAS*n*, and MSR are updated as shown in this table.

Table 7-26. Instruction TLB Error Interrupt Register Settings (directed to guest)

Register	Setting
GSRR0	Set to the effective address of the instruction causing the instruction TLB error interrupt
GSRR1	Set to the MSR contents at the time of the interrupt

Table 7-26. Instruction TLB Error Interrupt Register Settings (continued)(directed to guest)

Register	Setting
MSR	• CM is set to EPCR[GICM] <64> • GS, RI, ME, DE, CE are unchanged • UCLE is cleared if MSRP[UCLEP] = 0, otherwise it is unchanged • PMM is cleared if MSRP[PMMP] = 0, otherwise it is unchanged • All other defined MSR bits are cleared
MAS*n*	See Table 6-11.

If category E.HV is not implemented or the interrupt is directed to the hypervisor state, SRR0, SRR1, MAS*n*, and MSR are updated as shown in this table.

Table 7-27. Instruction TLB Error Interrupt Register Settings (directed to hypervisor)

Register	Setting
SRR0	Set to the effective address of the instruction causing the instruction TLB error interrupt
SRR1	Set to the MSR contents at the time of the interrupt
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared
MAS*n*	See Table 6-11.

Table 6-11 shows MAS register settings for data and instruction TLB error interrupts. Section 6.5.8.3, “MAS Register Updates for Exceptions, **tlbsx**, and **tlbre**.”

If category E.HV is implemented and the interrupt is directed to the guest supervisor state, instruction execution resumes at address GIVPR[0–47] || GIVOR14[48–59] || 0b0000. Otherwise, instruction execution resumes at address IVPR[0–47] || IVOR14[48–59] || 0b0000.

7.8.15 Debug Interrupt

A debug exception occurs when a debug event causes a corresponding DBSR bit to be set. A debug interrupt occurs when no higher priority exception exists (see Section 7.11, “Exception Priorities”), a debug exception is indicated in the DBSR, and debug interrupts are enabled (DBCR0[IDM] = 1 and MSR[DE] = 1). See Section 3.13, “Debug Registers.”

If category E.ED is implemented, DSRR0, DSRR1 and MSR are updated as shown in this table.

Table 7-28. Debug Interrupt Register Settings <E.ED>

Register	Setting
DSRR0	DSRR0 is set as follows: • For instruction complete (ICMP) debug exceptions, set to the address of the instruction that would have executed after the one that caused the debug interrupt. • For unconditional debug event (UDE) debug exceptions, set to the address of the instruction that would have executed next if the debug interrupt had not occurred. • For interrupt taken (IRPT) debug exceptions, set to the effective address of the first instruction of the interrupt that caused the interrupt taken debug event. • For critical interrupt taken (CRPT) debug exceptions, set to the effective address of the first instruction of the interrupt that caused the critical interrupt taken debug event. • For return from interrupt (RET) debug exceptions, set to the address of the **rfi** instruction that caused the debug interrupt. • For return from critical interrupt (CRET) debug exceptions, set to the address of the **rfci** instruction that caused the debug interrupt. • For unconditional debug event (UDE) debug exceptions, set to the address of the instruction that would have executed next if the debug interrupt had not occurred. • For all other debug events, set to the address of the instruction causing the debug interrupt. Some processors support imprecise debug events where DBSR bits are set when debug interrupts are disabled (MSR[DE] = 0 or DBCR0[IDM] = 0). If at a later time, both MSR[DE] and DBCR0[IDM] are modified such that they are both 1 and if the debug event is still set in DBSR, a delayed debug interrupt will occur prior to the next context synchronizing event. Likewise, if **mtspr** is used to set bits in DBSRWR<E.HV> and debug interrupts are already enabled, a delayed debug interrupt will occur. When a delayed debug interrupt occurs, DSRR0 is set to the address of the instruction that would have executed next, not with the address of the instruction that modified DBCR0, DBSRWR<E.HV>, or MSR and thus caused the interrupt.
DSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME are unchanged • All other defined MSR bits are cleared
DBSR	DBSR is set when a debug event occurs, not as a result of the interrupt. See Section 3.13.4, “Debug Status Register (DBSR/DBSRWR).”

If category E.ED is not implemented, CSRR0, CSRR1, and MSR are updated as shown in this table.

Table 7-29. Debug Interrupt Register Settings <E.ED>

Register	Setting
CSRR0	CSRR0 is set as follows: • For instruction complete (ICMP) debug exceptions, set to the address of the instruction that would have executed after the one that caused the debug interrupt. • For unconditional debug event (UDE) debug exceptions, set to the address of the instruction that would have executed next if the debug interrupt had not occurred. • For interrupt taken (IRPT) debug exceptions, set to the effective address of the first instruction of the interrupt that caused the interrupt taken debug event. • For return from interrupt (RET) debug exceptions, set to the address of the **rfi** instruction that caused the debug interrupt. • For unconditional debug event (UDE) debug exceptions, set to the address of the instruction that would have executed next if the debug interrupt had not occurred. • For all other debug events, set to the address of the instruction causing the debug interrupt. Some processors support imprecise debug events where DBSR bits are set when debug interrupts are disabled (MSR[DE] = 0 or DBCR0[IDM] = 0). If at a later time, both MSR[DE] and DBCR0[IDM] are modified such that they are both 1 and if the debug event is still set in DBSR, a delayed debug interrupt will occur prior to the next context synchronizing event. Likewise, if **mtspr** is used to set bits in DBSRWR<E.HV> and debug interrupts are already enabled, a delayed debug interrupt will occur. When a delayed debug interrupt occurs, CSRR0 is set to the address of the instruction that would have executed next, not with the address of the instruction that modified DBCR0, DBSRWR<E.HV>, or MSR and thus caused the interrupt.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME are unchanged • All other defined MSR bits are cleared
DBSR	DBSR is set when a debug event occurs, not as a result of the interrupt. See Section 3.13.4, “Debug Status Register (DBSR/DBSRWR).”

Instruction execution resumes at address IVPR[0–47] \|\| IVOR15[48–59] \|\| 0b0000.

Debug interrupts resulting from ICMP debug events are post-completion interrupts.

NOTE: Software Considerations

Delayed debug interrupts will be removed from a future version of the architecture and should not be relied on by software.

7.8.16 SPE/Embedded Floating-Point/AltiVec Unavailable Interrupt <SP or V>

The SPE, embedded floating-point, or AltiVec unavailable interrupt occurs when no higher priority exception exists, and an attempt is made to execute an SPE, embedded floating-point double-precision or embedded floating-point vector single-precision, or AltiVec instruction and MSR[SPV] = 0.

When an SPE, embedded floating-point, or AltiVec unavailable interrupt occurs, the processor suppresses execution of the interrupt-causing instruction. Table 7-30 describes register settings. If the SPE/embedded floating-point/AltiVec unavailable interrupt occurs, the processor suppresses execution of the instruction causing the exception.

The SRR0, SRR1, MSR, and ESR registers are modified as shown in this table.

Table 7-30. SPE/Embedded Floating-Point/Vector Unavailable Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the interrupt-causing instruction.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared
ESR	SPV Always set VLEMI Set if the interrupt-causing instruction resides in VLE storage; otherwise cleared<VLE> EPID Set if the interrupt-causing instruction is an external process ID instruction; otherwise cleared.<E.PD> All other defined ESR bits are cleared.

NOTE: Software Considerations

> For 32-bit devices that implement SPE, software should use this interrupt to determine whether an application uses the upper 32 bits of the GPRs and thus should thus be required to save and restore them on a context switch.

Instruction execution resumes at address IVPR[0–47] || IVOR32[48–59] || 0b0000.

7.8.17 Embedded Floating-Point Data Interrupt <SP.FD, SP.FS, SP.FV>

The embedded floating-point data interrupt occurs when no higher priority exception exists and an embedded floating-point data exception is presented to the interrupt mechanism. The embedded floating-point data exception causing the interrupt is indicated in the SPEFSCR. An embedded floating-point data interrupt is generated in the following cases:

- SPEFSCR[FINVE] = 1 and either SPEFSCR[FINVH,FINV] = 1
- SPEFSCR[FDBZE] = 1 and either SPEFSCR[FDBZH,FDBZ] = 1
- SPEFSCR[FUNFE] = 1 and either SPEFSCR[FUNFH,FUNF] = 1
- SPEFSCR[FOVFE] = 1 and either SPEFSCR[FOVFH,FOVF] = 1

Note that although SPEFSCR status bits can be updated by using **mtspr**, interrupts occur only if they are set as the result of an arithmetic operation.

When an embedded floating-point data interrupt occurs, the processor suppresses execution of the interrupt-causing instruction. SRR0, SRR1, ESR, SPEFSCR, and MSR are updated as shown in this table.

Table 7-31. Embedded Floating-Point Data Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the interrupt-causing instruction.
SRR1	Set to the MSR contents at the time of the interrupt.

Table 7-31. Embedded Floating-Point Data Interrupt Register Settings (continued)

Register	Setting
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared
ESR	SPV Always set VLEMI Set if the interrupt-causing instruction resides in VLE storage; otherwise cleared<VLE> EPID Set if the interrupt-causing instruction is an external process ID instruction; otherwise cleared.<E.PD> All other defined ESR bits are cleared.
SPEFSCR	One or more of the FINVH, FINV, FDBZH, FDBZ, FUNFH, FUNF, FOVFH, or FOVF bits are set to indicate the floating-point data exception type.

Instruction execution resumes at address IVPR[0–47] \|\| IVOR33[48–59] \|\| 0b0000.

NOTE: Software Considerations

AltiVec assist interrupt and embedded floating-point data interrupt share the same IVOR because they are mutually exclusive in implementations.

7.8.18 AltiVec Assist Interrupt <V>

The AltiVec assist interrupt occurs when no higher priority exception exists and a denormalized floating-point number is an operand to an AltiVec floating-point instruction requiring software assist, an AltiVec assist exception is presented to the interrupt mechanism. The instruction handler is required to emulate the interrupt causing instruction to provide correct results with the denormalized input.

When an AltiVec assist interrupt occurs, the processor suppresses execution of the interrupt-causing instruction. SRR0, SRR1, ESR, and MSR, are updated as shown in this table.

Table 7-32. AltiVec Assist Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the interrupt-causing instruction.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared
ESR	SPV Always set VLEMI Set if the interrupt-causing instruction resides in VLE storage; otherwise cleared<VLE> EPID Set if the interrupt-causing instruction is an external process ID instruction; otherwise cleared.<E.PD> All other defined ESR bits are cleared.

Instruction execution resumes at address IVPR[0–47] \|\| IVOR33[48–59] \|\| 0b0000.

NOTE: Software Considerations

AltiVec assist interrupt and embedded floating-point data interrupt share the same IVOR because they are mutually exclusive in implementations.

7.8.19 Embedded Floating-Point Round Interrupt <SP.FD, SP.FS, SP.FV>

The embedded floating-point round interrupt occurs when no higher priority exception exists, SPEFSCR[FINXE] is set, and any of the following occurs:

- The unrounded result of an embedded floating-point operation is not exact
- An overflow occurs and overflow exceptions are disabled (FOVF or FOVFH is set and FOVFE is cleared)
- An underflow occurs and underflow exceptions are disabled (FUNF is set and FUNFE is cleared).

The value of SPEFSCR[FINXS] is 1, indicating that one of the above exceptions has occurred, and additional information about the exception is found in SPEFSCR[FGH FG FXH FX].

Note that although SPEFSCR status bits can be updated by using an **mtspr**[SPEFSCR], interrupts occur only if they are set as the result of an arithmetic operation.

When an embedded floating-point round interrupt occurs, the processor completes the execution of the instruction causing the exception and writes the result to the destination register prior to taking the interrupt.

When an embedded floating-point round interrupt occurs, the unrounded (truncated) result is placed in the target register and SRR0, SRR1, ESR, SPEFSCR, and ESR are set as described in this table.

Table 7-33. Embedded Floating-Point Round Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction following the interrupt-causing instruction.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared
ESR	SPV Always set VLEMI Set if the interrupt-causing instruction resides in VLE storage; otherwise cleared<VLE> EPID Set if the interrupt-causing instruction is an external process ID instruction; otherwise cleared.<E.PD> All other defined ESR bits are cleared.
SPEFSCR	FGH, FXH, FG, and FX are set appropriately to indicate the results of the computation for the interrupt handler to decide how to round. Note: Some e500 cores always require rounding when SPEFSCR[FMRC] is set to round toward +infinity or round to -infinity.

Instruction execution resumes at address IVPR[0–47] || IVOR34[48–59] || 0b0000.

This is a post-completion interrupt.

NOTE: Software Considerations

If an implementation does not support ±Infinity rounding modes, the rounding mode is set to be +Infinity or -Infinity, and no higher priority exception exists, an embedded floating-point round interrupt occurs after every embedded floating-point instruction for which rounding might occur, regardless of the value of FINXE.

When the interrupt occurs, the unrounded (truncated) result of an inexact high or low element is placed in the target register. If only one element is inexact, the other, exact element is updated with the correctly rounded result and the FG and FX bits corresponding to the other element are both cleared.

The FG (FGH) and FX (FXH) bits allow an interrupt handler to round the result as it desires. FG (FGH) is the value of the bit immediately to the right of the lsb of the destination format mantissa from the infinitely precise intermediate calculation before rounding. FX (FXH) is the value of the OR of all the bits to the right of the FG (FGH) of the destination format mantissa from the infinitely precise intermediate calculation before rounding.

7.8.20 Performance Monitor Interrupt <E.PM>

A performance monitor exception occurs when counter overflow detection is enabled and a counter overflows. More specifically, for each performance monitor counter register *n*, if PMGC0[PMIE]=1 and PMLCa*n*[CE]=1 and PMC*n*[OV] =1 and MSR[EE] = 1 (or MSR[GS] = 1 <E.HV>), a performance monitor exception is said to exist. If it is the highest priority exception, the performance monitor exception condition causes a performance monitor interrupt.

The performance monitor exception is level sensitive and the condition may cease to exist if any of the required conditions fail to be met. Thus, if MSR[EE] remains 0 during the overflow condition, a counter can overflow and continue counting events until PMC*n*[OV] becomes 0 without taking a performance monitor interrupt. To avoid this, software should program the counters to freeze if an overflow condition is detected.

For a performance monitor interrupt to be signaled on an enabled condition or event, PMGC0[PMIE] must be set.

The performance monitor can freeze the performance monitor counters triggered by an enabled condition or event. For the performance monitor counters to freeze on an enabled condition or event, PMGC0[FCECE] must be set.

Although the interrupt condition could occur with MSR[EE] = 0, the interrupt cannot be taken until MSR[EE] = 1 (or MSR[GS] = 1 <E.HV>). If a counter overflows while the interrupt is not enabled, it is possible for the counter to wrap around to all zeros again without the performance monitor interrupt being taken.

The performance monitor interrupt is precise and asynchronous.

When a performance monitor interrupt occurs, SRR0, SRR1, and MSR, are updated as follows:

Table 7-34. Performance Monitor Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared

Instruction execution resumes at address IVPR[0–47] || IVOR35[48–59] || 0b0000.

NOTE: Software Considerations

When taking a performance monitor interrupt, software should clear the overflow condition by reading the counter register and setting it to a non-overflow value since the normal return from the interrupt will set MSR[EE] back to 1.

7.8.21 Processor Doorbell Interrupt <E.PC>

A processor doorbell interrupt occurs when no higher priority exception exists, a processor doorbell exception is present, and MSR[EE] = 1 (or MSR[GS] = 1 <E.HV>). Processor doorbell exceptions are generated when DBELL messages (see Section 7.12, “Processor Signaling (msgsnd and msgclr) <E.PC>”) are received and accepted by the processor.

When a processor doorbell interrupt occurs, SRR0, SRR1, and MSR are updated as shown in this table.

Table 7-35. Processor Doorbell Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the next instruction to be executed.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared

Instruction execution resumes at address IVPR[0–47] || IVOR36[48–59] || 0b0000.

7.8.22 Processor Doorbell Critical Interrupt <E.PC>

A processor doorbell critical interrupt occurs when no higher priority exception exists, a processor doorbell critical exception is present, and MSR[CE] = 1 (or MSR[GS] = 1 <E.HV>). Processor doorbell critical exceptions are generated when DBELL_CRIT messages (see Section 7.12, “Processor Signaling (msgsnd and msgclr) <E.PC>”) are received and accepted by the processor.

When a processor doorbell critical interrupt occurs, CSRR0, CSRR1, and MSR are updated as shown in this table.

Table 7-36. Processor Doorbell Critical Interrupt Register Settings

Register	Setting
CSRR0	Set to the effective address of the next instruction to be executed.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME are unchanged • DE is unchanged <E.ED> • All other defined MSR bits are cleared

Instruction execution resumes at address IVPR[0–47] \|\| IVOR37[48–59] \|\| 0b0000.

7.8.23 Guest Processor Doorbell Interrupt <E.HV>

A guest processor doorbell interrupt occurs when no higher priority exception exists, a guest processor doorbell exception is present, and MSR[EE] = 1 and MSR[GS] = 1. Guest processor doorbell exceptions are generated when G_DBELL messages (see Section 7.12, “Processor Signaling (msgsnd and msgclr) <E.PC>”) are received and accepted by the processor.

When a guest processor doorbell interrupt occurs, GSRR0, GSRR1, and MSR, are updated as shown in this table.

Table 7-37. Guest Processor Doorbell Interrupt Register Settings

Register	Setting
GSRR0	Set to the effective address of the next instruction to be executed.
GSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared

Instruction execution resumes at address IVPR[0–47] \|\| IVOR38[48–59] \|\| 0b0000.

NOTE: Virtualization

Guest processor doorbell interrupts can be used by the hypervisor to determine when MSR[EE] is set in the guest. The enabling condition for this interrupt requires that MSR[EE] be set and that the processor is in the guest state (MSR[GS] is set).

NOTE: Software Considerations

The guest processor doorbell uses GSRR0/1 to save state even though it is taken only in the hypervisor state. The primary function of this interrupt is to appropriately interrupt the guest state when MSR[EE] = 1 so that the hypervisor can safely reflect EE based interrupts to the guest. Using GSRR0/1 allows the hypervisor to construct the reflected interrupt vector into SRR0 and target MSR into SRR1 and then rfi to guest state since the contents of GSRR0/1 will already be correctly set for guest execution of its interrupt handler.

7.8.24 Guest Processor Doorbell Critical Interrupt <E.HV>

A guest processor doorbell critical interrupt occurs when no higher priority exception exists, a guest processor doorbell critical exception is present, and MSR[CE] = 1 and MSR[GS] = 1. Guest processor doorbell critical exceptions are generated when G_DBELL_CRIT messages (see Section 7.12, “Processor Signaling (msgsnd and msgclr) <E.PC>”) are received and accepted by the processor.

When a guest processor doorbell critical interrupt occurs, CSRR0, CSRR1, and MSR are updated as shown in this table.

Table 7-38. Guest Processor Doorbell Critical Interrupt Register Settings

Register	Setting
CSRR0	Set to the effective address of the next instruction to be executed.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME are unchanged • DE is unchanged <E.ED> • All other defined MSR bits are cleared

Instruction execution resumes at address IVPR[0–47] || IVOR39[48–59] || 0b0000.

NOTE: Virtualization

Guest processor doorbell critical interrupts can be used by the hypervisor to determine when MSR[CE] is set in the guest. The enabling condition for this interrupt requires that MSR[CE] be set and that the processor is in the guest state (MSR[GS] is set).

NOTE: Software Considerations

Guest processor doorbell critical interrupts use the same IVOR as Guest processor doorbell machine check interrupts. Software can discriminate these interrupts based on which was expected and what CSRR1[ME,CE] are.

7.8.25 Guest Processor Doorbell Machine Check Interrupt <E.HV>

A guest processor doorbell machine check interrupt occurs when no higher priority exception exists, a guest processor doorbell machine check exception is present, and MSR[ME] = 1 and MSR[GS] = 1. Guest processor doorbell machine check exceptions are generated when G_DBELL_MC messages (see Section 7.12, “Processor Signaling (msgsnd and msgclr) <E.PC>”) are received and accepted by the processor.

When a guest processor doorbell machine check interrupt occurs, CSRR0, CSRR1, and MSR are updated as shown in this table.

Table 7-39. Guest Processor Doorbell Machine Check Interrupt Register Settings

Register	Setting
CSRR0	Set to the effective address of the next instruction to be executed.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME are unchanged • DE is unchanged <E.ED> • All other defined MSR bits are cleared

Instruction execution resumes at address IVPR[0–47] || IVOR39[48–59] || 0b0000.

NOTE: Virtualization

Guest processor doorbell machine check interrupts can be used by the hypervisor to determine when MSR[ME] is set in the guest. The enabling condition for this interrupt requires that MSR[ME] be set and that the processor is in the guest state (MSR[GS] is set).

NOTE: Software Considerations

Guest processor doorbell critical interrupts use the same IVOR as Guest processor doorbell machine check interrupts. Software can discriminate these interrupts based on which was expected and what CSRR1[ME,CE] are.

7.8.26 Embedded Hypervisor System Call Interrupt <E.HV>

An embedded hypervisor system call interrupt occurs when no higher priority exception exists and a System Call (**sc**) instruction with LEV = 1 is executed.

SRR0, SRR1, and MSR are updated as shown in this table.

Table 7-40. Embedded Hypervisor System Call Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction after the **sc** instruction.

Table 7-40. Embedded Hypervisor System Call Interrupt Register Settings (continued)

Register	Setting
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared

Instruction execution resumes at address IVPR[0–47] || IVOR40[48–59] || 0b0000.

The system call interrupt is a post-completion synchronous interrupt.

NOTE: Virtualization

The embedded hypervisor system call is intended for use by the hypervisor as a mechanism to receive “HCALLs”.

NOTE: Software Considerations

The Embedded Hypervisor should check SRR1[PR] and SRR1[GS] to determine the privilege level of the software making the system call to determine what action, if any, should be taken as a result of the system call.

7.8.27 Embedded Hypervisor Privilege Interrupt <E.HV>

An embedded hypervisor privilege interrupt occurs when no higher priority exception exists and an embedded hypervisor privilege exception is present. An embedded hypervisor privilege exception occurs when the processor executes an instruction in the guest supervisor state (MSR[GS] = 1 & MSR[PR] = 0) and the operation is allowed only in the hypervisor state. An embedded hypervisor privilege exception also occurs when one of the following occurs:

- **ehpriv** instruction is executed, regardless of the state of the processor
- **tlbilx** or **tlbwe** is executed in the guest supervisor state and EPCR[DGTMI] = 1

SRR0, SRR1, and MSR are updated as shown in this table.

Table 7-41. Embedded Hypervisor Privilege Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared

Instruction execution resumes at address IVPR[0–47] || IVOR41[48–59] || 0b0000.

Embedded hypervisor privilege interrupts are provided as a means for restricting the guest supervisor state from performing operations allowed only in the embedded hypervisor state.

NOTE: Virtualization

The embedded hypervisor privilege interrupt is the entry point for the hypervisor to provide virtualization for resources which are not directly accessible from the guest supervisor state. In general, the hypervisor will need to retrieve the instruction pointed at by SRR0, then decode and emulate it appropriately.

7.8.28 LRAT Error Interrupt <E.HV.LRAT>

An LRAT error interrupt occurs when no higher priority exception exists and an LRAT error exception is present. An LRAT error exception occurs when any one of the following conditions exist:

- The processor executes a **tlbwe** instruction in the guest supervisor state (MSR[GS] = 1 & MSR[PR] = 0) to a TLB array which is eligible for LRAT translation and no LRAT entry matches the logical address specified in the RPN fields of the MAS registers.

SRR0, SRR1, and MSR are updated as shown in this table.

Table 7-42. Embedded Hypervisor Privilege Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	• CM is set to EPCR[ICM] <64> • RI, ME, DE, CE are unchanged • All other defined MSR bits are cleared

Instruction execution resumes at address IVPR[0–47] || IVOR42[48–59] || 0b0000.

LRAT error interrupts are provided as a means for allowing the hypervisor to manage logical to real translations providing the capability for guest operating systems to write TLB entries.

7.9 Partially Executed Instructions

In general, the architecture permits load and store instructions to be partially executed, interrupted, and then restarted from the beginning upon return from the interrupt. To guarantee that a particular load or store instruction completes without being interrupted and restarted, software must mark the memory as guarded and use an elementary (non-multiple) load or store aligned on an operand-sized boundary.

To guarantee that load and store instructions can, in general, be restarted and completed correctly without software intervention, the following rules apply when an interrupt occurs after partial execution:

- For an elementary load, no part of a target register (for example, **r**D or **fr**D) has been altered.
- For update forms of load or store, the update register, **r**A, will not have been altered (note these are invalid forms).

The following effects are permissible when certain instructions are partially executed and then restarted:

- For any store, bytes at the target location may have been altered (if write access to that page in which bytes were altered is permitted by the access control mechanism). In addition, for store conditional instructions, CR0 has been set to an undefined value, and it is undefined whether the reservation has been cleared or not.

- For any load, bytes at the addressed location may have been accessed (if read access to that page in which bytes were accessed is permitted by the access control mechanism).
- For load multiple, some registers in the range to be loaded may have been altered. Including the addressing registers **r**A and possibly **r**B in the range to be loaded is a programming error, and thus the rules for partial execution do not protect these registers against overwriting.

In no case is access control violated.

As previously stated, elementary, aligned, guarded loads and stores are the only access instructions guaranteed not to be interrupted after being partially executed. The following list identifies the instruction types that can be interrupted after partial execution, as well as the specific interrupt types that could cause the interruption:

1. Any load or store (except elementary, aligned, guarded):
   - Any asynchronous interrupt
   - Error report interrupt
2. Unaligned elementary load or store, or any multiple:

   All of the above listed under item 1, plus the following:
   - Data storage (if the access crosses a protection boundary)
   - Data TLB Error (if the access crosses a protection boundary)
   - Debug (data address compare, data value compare)
3. **mtcrf** may also be partially executed due to the occurrence of any of the interrupts listed under item 1 while **mtcrf** was executing.
   - All instructions prior to the **mtcrf** have completed execution. (Some storage accesses generated by these preceding instructions may not have completed.)
   - No subsequent instruction has begun execution.
   - The **mtcrf** instruction (the address of which is saved in SRR0/CSRR0/MCSRR0/(DSRR0<E.HV>) at the occurrence of the interrupt), may appear not to have begun or may have partially executed.

7.10 Interrupt Ordering and Masking

Multiple exceptions can exist simultaneously, each of which could cause the generation of an interrupt. The architecture does not provide for reporting more than one interrupt of the same class (machine check class, debug class <E.ED>, critical class, and base class). Therefore, the architecture defines that interrupts are ordered with respect to each other and provides a masking mechanism for certain persistent interrupt types.

When an interrupt is masked (disabled), and an exception occurs that would normally generate the interrupt, the exception persists as a status bit in a register (which register depends upon the exception type) or within the processor state. However, no interrupt is generated. Later, if that interrupt is enabled (unmasked), and the exception status has not been cleared by software, the interrupt due to the original exception event is finally generated.

All asynchronous interrupts, except for NMI can be masked. Additionally, certain synchronous interrupts can be masked. An example of such an interrupt is the floating-point enabled exception type program interrupt. The execution of a floating-point instruction that causes FPSCR[FEX] to be set is considered an exception event, regardless of the setting of MSR[FE0,FE1]. If MSR[FE0,FE1] are both 0, then the floating-point enabled exception type of program interrupt is masked, but the exception persists in FPSCR[FEX]. Later, if MSR[FE0,FE1] are enabled, the interrupt will finally be generated. EIS considers such masked synchronous interrupt as if they are asynchronous.

The architecture enables implementations to avoid situations in which an interrupt would cause the state information (saved in save/restore registers) from a previous interrupt to be overwritten and lost. In order to do this, the architecture defines interrupt classes in a hierarchical manner. At each interrupt class, hardware automatically disables any further interrupts associated with the interrupt class by masking the interrupt enable in the MSR when the interrupt is taken. Additionally, each interrupt class masks the interrupt enable in the MSR for each lower class in the hierarchy.

The hierarchy of interrupt classes from highest to lowest is shown in this table.

Table 7-43. Interrupt Hierarchy

Interrupt Class	MSR Enables Cleared	Save/Restore Registers	Notes
Machine check	ME,DE, CE, EE	MSRR0/1	—
Debug	DE,CE,EE	DSRR0/1	<E.ED>. When category E.ED is not present, debug interrupts are treated as critical class interrupts
Critical	DE[1],CE,EE	CSRR0/1	When category E.ED is not present, debug interrupts are treated as critical class interrupts
Base[2]	EE	SRR0/1 (GSRR0/1<E.HV>)	—

Note:

[1] When category E.ED is present, critical class interrupts do not clear DE.

[2] If category E.HV is present, guest interrupts are considered part of the base class.

<Embedded.Hypervisor>:
Interrupts that are normally masked by EE, CE, DE, and ME are not masked for interrupts that are directed to the hypervisor when executing in the guest state (MSR[GS]=1) except for guest processor doorbell, guest processor doorbell critical, and guest processor doorbell machine check. This allows the hypervisor to service interrupts regardless of the state of the guest. No state is lost because for base class interrupts the guest uses GSRR0/1 and the save/restore registers for other classes of interrupts are emulated by the hypervisor.

Base interrupts and error report interrupts that occur as a result of precise exceptions are not masked by MSR[EE] or MSR[ME], and any such exception that occurs before software saves the save/restore registers in an exception handler results in a situation that could result in the loss of state information.

This first step of the hardware clearing the lower-hierarchy MSR enable bits shown in Figure 7-43 prevents any subsequent asynchronous interrupts from overwriting the save/restore registers (SRR0/SRR1, GSRR0/GSRR1<E.HV>, CSRR0/CSRR1, DSRR0/DSRR1<E.ED>, or

MCSRR0/MCSRR1), before software being able to save their contents. Hardware also automatically clears, on any interrupt, MSR[WE,PR,FP,FE0,FE1,IS,DS,SPV<SP or V>] and MSR[GS] when directed to the hypervisor state <E.HV>, which helps avoid subsequent interrupts of certain other types. However, guaranteeing that interrupt classes lower in the hierarchy do not occur and overwrite the save/restore registers also requires the cooperation of system software. Specifically, system software must avoid the execution of instructions that could cause (or enable) a subsequent interrupt, if the contents of the save/restore registers have not yet been saved.

7.10.1 Guidelines for System Software

Before saving the save/restore registers' contents, system software must avoid the following actions:

- Re-enabling an interrupt class that is at the same or a lower level in the interrupt hierarchy. This includes the following actions:
  - Re-enabling MSR[EE]
  - Setting MSR[GS] when MSR[GS] = 0 <E.HV>
  - Re-enabling MSR[CE,EE] in critical class interrupt handlers, and if the category E.ED is not supported, re-enabling of MSR[DE].
  - Re-enabling of MSR[CE,EE,DE] in debug class interrupt handlers <E.ED>
  - Re-enabling MSR[EE,CE,DE,ME] in machine check class interrupt handlers.
- Branching (or sequential execution) to addresses not mapped by the TLB, or mapped without UX=1 or SX=1 permission. This prevents instruction storage and instruction TLB error interrupts.
- Load, store or cache management instructions to addresses not mapped by the TLB or not having required access permissions. This prevents data storage and data TLB error interrupts.
- Execution of System Call (**sc**) or trap (**tw**, **twi**, **td <64>**, **tdi <64>**) instructions. This prevents system call and trap exception type program interrupts.
- Execution of any floating-point <FP>, SPE <SP>, embedded floating-point <SP.FD, SP.FV>, or AltiVec <V> instruction. This prevents unavailable interrupts. Note that this interrupt would occur upon the execution of any floating-point instruction, due to the automatic clearing of MSR[FP]. However, even if software were to re-enable MSR[FP], floating-point instructions must still be avoided to prevent program interrupts due to possible program interrupt exceptions (floating-point enabled, unimplemented operation).
- Re-enabling of MSR[PR]. This prevents privileged instruction exception type program interrupts. Alternatively, software could re-enable MSR[PR], but avoid the execution of any privileged instructions.
- Execution of any illegal instructions. This prevents illegal instruction exception type program interrupts.
- Execution of any instruction that could cause an alignment interrupt. This prevents alignment interrupts. Included in this category are any multiple instructions, and any unaligned elementary load or store instructions. See Section 7.8.6, "Alignment Interrupt," for a complete list of instructions that may cause alignment interrupts.

It is not necessary for hardware or software to avoid interrupts higher in the interrupt hierarchy (see Figure 7-43) from within interrupt handlers (and hence, for example, hardware does not automatically clear MSR[CE,ME,DE] upon a base class interrupt), since interrupts at each level of the hierarchy use different pairs of save/restore registers to save the instruction address and MSR. The converse, however, is not true. That is, hardware and software must cooperate in the avoidance of interrupts lower in the hierarchy from occurring within interrupt handlers, even though the these interrupts use different save/restore register pairs. This is because the interrupt higher in the hierarchy may have occurred from within a interrupt handler for an interrupt lower in the hierarchy prior to the interrupt handler having saved the save/restore registers. Therefore, within an interrupt handler, save/restore registers for all interrupts lower in the hierarchy may contain data that is necessary to the system software.

7.11 Exception Priorities

All synchronous interrupts, except for synchronous interrupts that can be delayed (for example, floating-point enabled exceptions <FP> and delayed debug interrupts), are reported in program order, as required by the sequential execution model. Delayed synchronous exceptions are considered asynchronous interrupts.

For any single instruction attempting to cause multiple exceptions for which the corresponding synchronous interrupt types are enabled, this section defines the priority order by which the instruction can cause a single enabled exception, thus generating a particular synchronous interrupt. Note that it is this exception priority mechanism, along with the requirement that synchronous interrupts be generated in program order, that guarantees that at any given time, there exists for consideration only one of the synchronous interrupt types. The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions.

The following instructions may be broken into multiple accesses, which may be performed in any order:

- Unaligned load and store
- Load multiple
- Store multiple
- Load string
- Store string

The exception priority mechanism applies to each of the multiple storage accesses, in the order in which they are performed.

This section does not define the permitted setting of multiple exceptions for which corresponding interrupt types are disabled. The generation of such exceptions has no effect on the generation of other exceptions for whose interrupt types are enabled. Conversely, if a particular exception for which the corresponding interrupt type is enabled is shown in the following sections to be of a higher priority than another exception, it prevents the setting of that lower priority exception, regardless of whether the lower priority exception’s interrupt type is enabled.

Except as specifically noted, only one exception type listed for a given instruction type can be generated at any given time. The priority of the exception types are listed in the following sections ranging from highest to lowest, within each instruction type.

NOTE: Software Considerations

> Some exception types may even be mutually exclusive of each other and could otherwise be considered the same priority. In such cases, exceptions are listed in the order suggested by the sequential execution model.

Exception types are defined to be either synchronous, in which case the exception occurs as a direct result of an instruction in execution, or asynchronous, which occurs based on an event external to the execution of a particular instruction or an instruction removes a gating condition to a pending exception. Exceptions are either synchronous or asynchronous exclusively although some synchronous exceptions which can be delayed are considered asynchronous when they are delayed.

Processors order asynchronous exceptions among only asynchronous exceptions and order synchronous exceptions among only synchronous exceptions. This is because asynchronous exceptions may temporally be sampled either before or after an instruction is completed. The distinction is important because certain synchronous exceptions require post completion actions. These exceptions cannot be separated from the completion of the instruction (e.g. system call or debug instruction complete). Therefore, asynchronous exceptions cannot be sampled during the completion and post completion synchronous exceptions for a given instruction.

The relative priority of each exception type is included in two tables, one for asynchronous exceptions and one for synchronous exceptions. In many cases, it is impossible to have certain exceptions occur at the same time (e.g. program - trap and program - illegal cannot occur on the same instruction). In general those exceptions are grouped together at the same relative priority.

Table 7-44 and Table 7-45 define the exception priority ordering. Exceptions are ordered from the highest priority to the lowest priority.

Table 7-44. Asynchronous Exception Priorities

Relative Priority	Exception	Interrupt Class	Comments
0	Machine check	Machine Check	Asynchronous exceptions may come from the processor or from an external source.
1	Guest processor doorbell machine check <E.HV>	Critical	—
2	Debug- UDE	Critical or Debug<E.ED>	Debug-UDE is generally used for an externally generated high priority attention signal.
	Delayed debug- IDE	Critical or Debug<E.ED>	Imprecise debug event usually taken after MSR[DE] goes from 0 to 1 via **rfdi** <E.ED>, **rfci** or **mtmsr**.
	Debug - interrupt taken	Critical or Debug<E.ED>	Debug interrupt taken after original interrupt has changed NIA and MSR.
	Debug - critical interrupt taken <E.ED>	Debug<E.ED>	Debug interrupt taken after original critical interrupt has changed NIA and MSR.
3	Critical input	Critical	—
4	Watchdog	Critical	—
5	Processor doorbell critical <E.PC>	Critical	—

Table 7-44. Asynchronous Exception Priorities (continued)

Relative Priority	Exception	Interrupt Class	Comments
6	Guest processor doorbell critical <E.HV>	Critical	—
7	External input	Base	—
14	Program - delayed floating-point enabled <FP>	Base	Delayed floating-point enabled exceptions occur when FPCSR[FEX] = 1 and MSR[FE0,FE1] change from 0b00 to a non-zero value.
30	Fixed interval timer	Base	—
31	Decrementer	Base	—
32	Processor doorbell <E.PC>	Base	—
33	Guest processor doorbell <E.HV>	Base	—
34	Performance monitor <E.PM>	Base	—

Table 7-45. Synchronous Exception Priorities

Relative Priority	Exception	Interrupt Class	Pre or Post Completion[1]	Comments
0	Error report	Machine check	pre	—
8	Debug - instruction address compare	Critical or Debug<E.ED>	pre	—
9	Instruction TLB error	Base	pre	—
	Instruction storage	Base	pre	—
11	Program - privileged instruction	Base	pre	—
	Embedded hypervisor privilege <E.HV>	Base	pre	—
12	Floating-point unavailable <FP>	Base	pre	—
	AltiVec unavailable <V>	Base	pre	—
	SPE unavailable <SP>	Base	pre	—
	Embedded floating-point unavailable <SP.FD or SP.FV>	Base	pre	—
13	Debug - trap	Critical or Debug<E.ED>	pre	—

Table 7-45. Synchronous Exception Priorities (continued)

Relative Priority	Exception	Interrupt Class	Pre or Post Completion[1]	Comments
14	Program - illegal instruction	Base	pre	—
	Program - unimplemented operation	Base	pre	Note: unimplemented operation is being phased out of the architecture.
	Program - trap	Base	pre	—
	Program - floating-point enabled <FP>	Base	pre	If a floating-point enabled exception is masked, it becomes a delayed floating-point enabled exception which is asynchronous.
15	(Alignment)	Base	pre	Alignment may be handled at either priority.
16	Data TLB error	Base	pre	—
17	Data storage - virtualization fault <E.HV>	Base	pre	A virtualization fault always takes priority over all other causes of data storage.
18	Data storage	Base	pre	All other data storage exceptions
19	Alignment	Base	pre	Alignment may be handled at either priority.
21	System call	Base	post	System Call Interrupt has SRR0 pointing to instruction after sc (i.e. post completion).
	Embedded hypervisor system call <E.HV>	Base	post	Embedded Hypervisor System Call Interrupt has SRR0 pointing to instruction after **sc** (i.e. post completion).
	Embedded floating-point data <SP.FD,SP.FS,SP.FV>	Base	pre	—
	Embedded floating-point round <SP.FD,SP.FS,SP.FV>	Base	post	—
	AltiVec assist <V>	Base	pre	—
22	Debug - return from interrupt	Critical or Debug<E.ED>	pre	—
	Debug - return from critical interrupt	Debug <E.ED>	pre	—
	Debug - branch taken	Critical or Debug<E.ED>	pre	—
23	Debug - data address compare	Critical or Debug<E.ED>	pre or post	.
24	Debug - data value compare	Critical or Debug<E.ED>	pre or post	
25	Debug - instruction complete	Critical or Debug<E.ED>	post	—

[1] Pre or Post Completion refers to whether the exception occurs before an instruction completes (pre) and the corresponding interrupt xSRR0 points to the instruction causing the exception, or if the instruction completes (post) and the corresponding xSRR0 points to the instruction after the instruction causing the exception.

7.12 Processor Signaling (msgsnd and msgclr) <E.PC>

Processor signaling provides facilities for processors within a coherence domain to send messages to other devices in the coherence domain. The facility provides a mechanism for sending interrupts to other processors that are not dependent on the interrupt controller and allow message filtering by the processors that receive the message.

Processor signaling may also be useful for sending messages to a device to provide specialized services.

This section describes only how processors send messages and what actions processors take on the receipt of a message. The actions taken by devices other than processors is defined in the architectural descriptions of those devices and is beyond the scope of EREF.

Messages are sent by processors when the **msgsnd** instruction is executed. When a message is received by a processor, it is "filtered" (how the processor determines if the message is intended for it) based on the message type and payload. If the processor determines that the message is intended for itself, the message is "accepted" and the processor performs predefined actions based on the message type.

Table 7-47 describes the message types that can be sent and their resulting actions on processors that accept them.

7.12.1 Sending Messages

Processors initiate a message by executing the **msgsnd** instruction and specifying a message type and message payload in **r**B. Sending a message causes the message to be sent to all the devices in the coherence domain in a reliable manner. The message is broadcast on the interconnect mechanism that connects all devices in the coherence domain and has a unique transaction type that cannot be generated using any other processor operations. The uniqueness of the transaction type insures that processors can only generate a message transaction using the defined instructions that send such messages.

Each device receives all messages that are sent. The actions that a device takes are dependent on the message type and payload. There are no restrictions on what messages a processor can send. However, **msgsnd** is hypervisor privileged.

Messages are also partitioned in that the payload contains an LPID field. <E.HV>

To provide interprocessor interrupt (IPI) capability, two message types are defined that produce interrupts on processors that receive and accept these message types. These messages are called doorbell messages because the message payload is only used to determine which processor accepts these messages. These message types are:

DBELL	Processor doorbell. Processor doorbell generates a processor doorbell exception on the targeted processor(s).
DBELL_CRIT	Processor doorbell critical. Processor doorbell critical generates a processor doorbell critical exception on the targeted processor(s).

<E.HV>
Other message types are defined for specific use by the hypervisor to interrupt a guest operating system when the guest operating system is in a specific state with respect to whether asynchronous interrupts are

enabled or not. There are three message types defined for this purpose. These message types are called guest doorbell messages because they require that the guest be in execution (for example, MSR[GS] = 1) and that the associated asynchronous interrupt enable bit is also set (MSR[EE]. MSR[CE], or MSR[ME]). These messages are used by the hypervisor to deliver asynchronous interrupts to the guest operating systems because the message will cause an interrupt when the appropriate enabling conditions are met while executing in the guest. These message types are:

G_DBELL — Guest processor doorbell. Guest processor doorbell generates a guest processor doorbell exception on processors which accept the message. <E.HV>

G_DBELL_CRIT — Guest processor doorbell critical. Guest processor doorbell critical generates a guest processor doorbell critical exception on processors which accept the message. <E.HV>

G_DBELL_MC — Guest processor doorbell machine check. Guest processor doorbell machine check generates a guest processor doorbell machine check exception on processors which accept the message. <E.HV>

In general, sending a message is not ordered with memory accesses. However, stores can be ordered with message sending by using **sync 0** as a barrier. The ordering capability only allows previous stores to be ordered with a subsequent message send. For example the following code sequence ensures that the store is performed before the message is sent:

```
stw    r4,0,r3
sync   0            // ensures the ordering of the store prior to the message send
msgsnd r10
```

However, the converse cannot be ensured. The following code sequence will *not* guarantee that the message is sent before the store is performed:

```
msgsnd r10
sync   0            // will *not* guarantee that the message is sent prior to the store
stw    r4,0,r3
```

See the **sync** instruction for more details about the barrier.

7.12.2 Receiving, Filtering, and Accepting Messages

When a processor receives a message, the processor will filter the message by examining the payload and decide whether to accept the message. Processors will accept doorbell (and guest doorbell <E.HV>) type messages that are intended for the processor. When a message is accepted by a processor, the asynchronous exception associated with the particular message type occurs. The exception condition remains until the processor takes the interrupt associated with the exception or the exception condition is cleared by the processor executing a **msgclr** instruction with the associated message type.

Once a message is accepted (and the exception is pending), changing state associated with the acceptance criteria will not clear a pending exception. For example, if a message is accepted on a processor based on a match of PIR and PIRTAG, and the associated exception is pending, a change to PIR will not clear the pending exception.

While message delivery from a message send is reliable and guaranteed, exception conditions resulting from accepting a message are not cumulative (i.e. they do not queue). That is for a given processor, if an exception condition exists for a particular message type, and another message of the same type is accepted on that processor, the acceptance of the second message does not cause a queued exception condition. If a one to one send/receive paradigm is desired, software must account for the possibility that each message sent will not result in a distinct interrupt on the accepting processor.

7.12.3 Message Types and Definitions

This table show the message types and criteria for acceptance by processors based on the value of **r**B from a **msgsnd** instruction.

Table 7-46. msgsnd rB Field Descriptions

Bits	Name	Description
32–36	TYPE	Message type. The type of message. Valid message types accepted by processor are: 0 DBELL. A processor doorbell exception is generated if the message is accepted. 1 DBELL_CRIT. A processor doorbell critical exception is generated if the message is accepted. 2 G_DBELL. A guest processor doorbell exception is generated if the message is accepted. <E.HV> 3 G_DBELL_CRIT. A guest processor doorbell critical exception is generated if the message is accepted. <E.HV> 4 G_DBELL_MC. A guest processor doorbell machine check exception is generated if the message is accepted. <E.HV>
37	BRDCAST	Broadcast. The message is accepted by all processors regardless of the value of PIRTAG.. 0 Not a broadcast. PIRTAG (and LPIDTAG <E.HV>) determine whether the message is accepted or not. 1 The message is accepted regardless of the value of PIRTAG (if LPIDTAG = LPIDR <E.HV>).
38–41	—	Reserved, should be cleared.
42–49	LPIDTAG <E.HV>	LPID tag. The message is only accepted if LPIDTAG = LPIDR on the receiving processor, regardless of the values of PIRTAG or BRDCAST.
50–63	PIRTAG	PIR tag. This field is used to identify a particular processor. The contents of this field are compared with bits 50:63 of the PIR register for DBELL and DBELL_CRIT type messages. The contents of this field are compared with bits 50:63 of the GPIR register for G_DBELL, G_DBELL_CRIT, and G_DBELL_MC type messages. <E.HV>. If the PIRTAG matches (and LPIDTAG = LPIDR <E.HV>), then the message is accepted.

This table shows processor actions when a message is accepted by message type.

Table 7-47. Processor Actions When a Message is Accepted

Message Type	PIRTAG compared to	Exception	Interrupt Enabling Condition	Interrupt	Save and Restore Registers	State Interrupt Directed to
DBELL	PIR[50:63]	Processor doorbell	MSR[EE] | MSR[GS]	Section 7.8.21, “Processor Doorbell Interrupt <E.PC>”	SRR0/1	Hypervisor
DBELL_CRIT	PIR[50:63]	Processor doorbell critical	MSR[CE] | MSR[GS]	Section 7.8.22, “Processor Doorbell Critical Interrupt <E.PC>”	CSRR0/1	Hypervisor
G_DBELL	PIR[50:63]	Guest processor doorbell	MSR[EE] & MSR[GS]	Section 7.8.23, “Guest Processor Doorbell Interrupt <E.HV>”	GSRR0/1[1]	Hypervisor
G_DBELL_CRIT	PIR[50:63]	Guest processor doorbell critical	MSR[CE] & MSR[GS]	Section 7.8.24, “Guest Processor Doorbell Critical Interrupt <E.HV>”	CSRR0/1	Hypervisor
G_DBELL_MC	PIR[50:63]	Guest processor doorbell machine check	MSR[ME] & MSR[GS]	Section 7.8.25, “Guest Processor Doorbell Machine Check Interrupt <E.HV>”	CSRR0/1[2]	Hypervisor

[1] Even though the interrupt is directed to the hypervisor, the state is saved in GSRR0/1. Since this interrupt is used by the hypervisor to emulate another MSR[EE] enabled interrupt to the guest, this simplifies the emulation since GSRR0/1 will be appropriately set.

[2] Critical level and machine check level guest processor doorbell interrupts are both directed to the same vector since the hypervisor uses this to emulate a critical or a machine check interrupt to the guest. The hypervisor knows which interrupts are being emulated and CSRR1[ME] and CSRR1[CE] can be used to determine which interrupt to emulate.

Chapter 8
Timer Facilities

This chapter describes the architecture-defined timer resources, which are as follows:

- Section 8.3, "Time Base Operation"
- Section 8.4, "Decrementer Operation"
- Section 8.5, "Fixed-Interval Timer"
- Section 8.6, "Watchdog Timer"
- Section 8.7, "Alternate Time Base"

8.1 Timer Facilities

The time base, decrementer, fixed-interval timer, watchdog timer, and alternate time base provide timing functions for the system. All of these must be initialized during start-up.

- The time base provides a long-period counter driven at an implementation-dependent frequency.
- The decrementer provides a counter that is updated at the same rate as the time base and provides a means of signaling an exception after a specified time has elapsed unless one of the following occurs:
  - — The decrementer is altered by software in the interim.
  - — The time base update frequency changes.

  The decrementer is typically used as a general-purpose software timer.
- The fixed-interval timer is essentially a selected bit of the time base, which provides a means of signaling an exception whenever the selected bit transitions from 0 to 1, in a repetitive fashion. The fixed-interval timer is typically used to trigger periodic system maintenance functions. Software may select any bit in the time base to serve as the fixed-interval timer. See Section 8.5, "Fixed-Interval Timer."
- The watchdog timer is also a selected bit of the time base, which provides a means of signalling a critical class exception whenever the selected bit transitions from 0 to 1. In addition, if software does not respond in time to the initial exception, then a watchdog timer-generated processor reset may result, if so enabled. The watchdog timer is typically used to provide a system error recovery function. See Section 8.6, "Watchdog Timer."
- Alternate time base register <ATB>—contains a 64-bit unsigned integer that is incremented periodically at an implementation-defined frequency. See Section 3.7.6, "Alternate Time Base Registers (ATB)."

8.2 Timer Registers

This section summarizes registers used by the timer facilities, shown in Figure 8-1. Full descriptions are given in Section 3.7, “Timer Registers.”

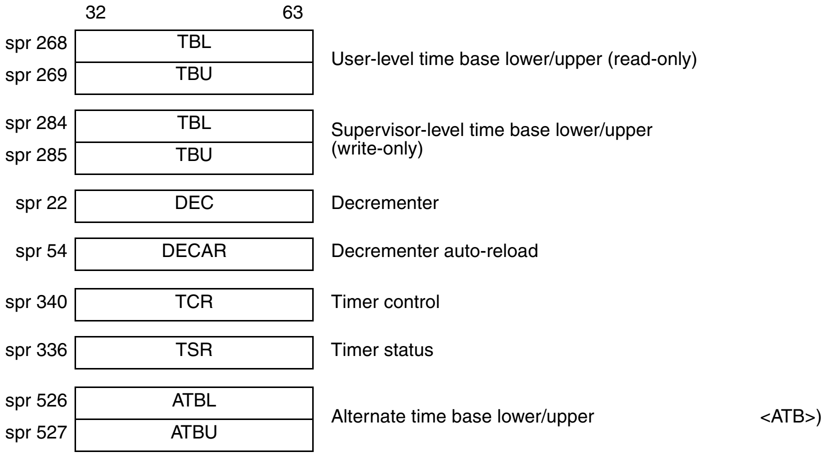


Figure 8-1. Timer/Decrementer Registers Comparison

- Time base (TB)—The time base is a 64-bit counter composed of two 32-bit registers, the time base upper (TBU) concatenated on the right with the time base lower (TBL). Section 3.7.3, “Time Base (TB),” describes the TB in detail.
- Decrementer register (DEC)—The decrementer timer is a 32-bit register which decrements at the time base frequency. The DEC can be programmed to interrupt when a decrement occurs on the value of 1 in DEC. See Section 3.7.4, “Decrementer Register (DEC),” for more information.
- Decrementer auto-reload register (DECAR)—Supports the auto-reload feature of the decrementer to reload the DEC when the decrementer counts down to 0. See Section 3.7.5, “Decrementer Auto-Reload Register (DECAR),” for more information.
- On some processors, the time base clock is selected via the hardware implementation-dependent register 0 (HID0[SEL_TBCLK]) and the time base is enabled via HID0[TBEN]. See Section 3.6.4.1, “Hardware Implementation-Dependent Register 0 (HID0).” On some processors the frequency of the time base clock and whether time base ticks are enabled is controlled through the integrated device. See the core reference manual and the integrated device reference manual for more information.
- Timer control register (TCR)—Provides control information for the decrementer, fixed interval timer, and watchdog timer. Section 3.7.1, “Timer Control Register (TCR),” describes TCR fields.
- Timer status register (TSR)—Contains status on timer events and the most recent watchdog timer-initiated processor reset. Section 3.7.2, “Timer Status Register (TSR),” describes TSR fields.
- Alternate time base register (ATB) <ATB>—Provides a read-only alternate 64-bit time base that increments at an implementation defined frequency (usually the processor clock). Like the time base, the ATB is divided into ATBL and ATBU registers. ATB is an alias for ATBL. See Section 3.7.6, “Alternate Time Base Registers (ATB).

8.3 Time Base Operation

This section describes how time base registers are accessed and programmed. The relationship of these timer facilities to each other is shown in Figure 8-2.

On some processors the clock source for the TB is specified by two fields in HID0: time base enable (TBEN), and select time base clock (SEL_TBCLK). If the TB is enabled, (HID0[TBEN] = 1), the clock source is determined as follows:

- If HID0[TBEN] = 1 and HID0[SEL_TBCLK] = 0, the time base updates at a core-defined rate.
- If HID0[TBEN] = 1 and HID0[SEL_TBCLK] = 1, the time base is updated based on an external signal to the core.

On some processors the clock source for the TB is controlled by the integrated device. The implementation documentation identifies the supported frequencies and clock input sources.

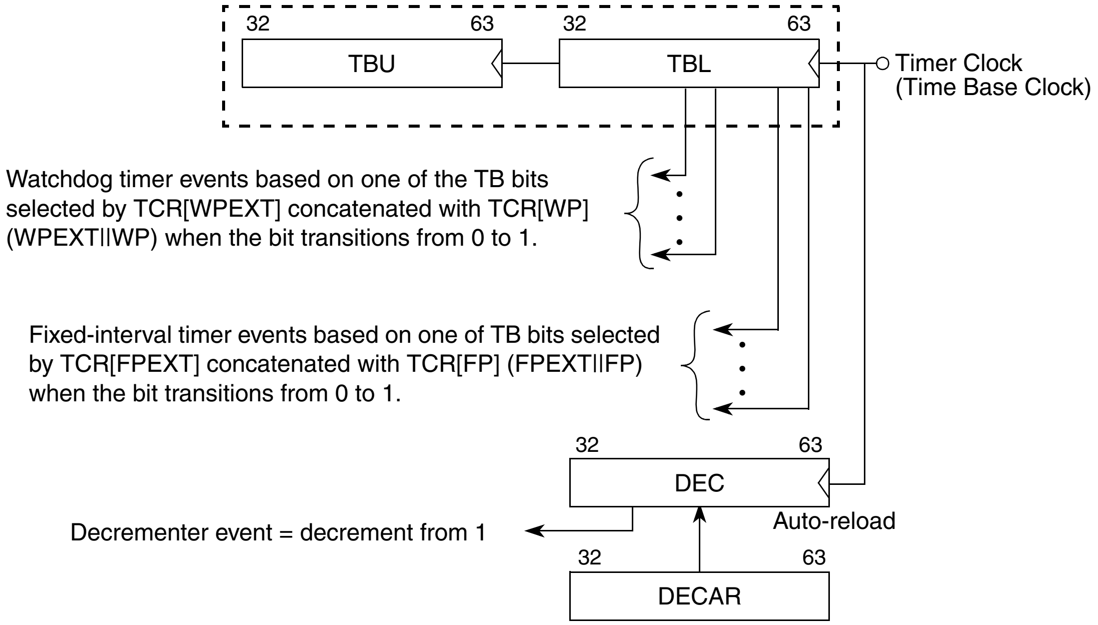

32
63
TBU
32
63
TBL
Timer Clock
(Time Base Clock)
Watchdog timer events based on one of the TB bits selected by TCR[WPEXT] concatenated with TCR[WP] (WPEXT||WP) when the bit transitions from 0 to 1.
Fixed-interval timer events based on one of TB bits selected by TCR[FPEXT] concatenated with TCR[FP] (FPEXT||FP) when the bit transitions from 0 to 1.
32
63
DEC
Auto-reload
Decrementer event = decrement from 1
32
63
DECAR

Figure 8-2. Relationship of Timer Facilities to Time Base

8.3.1 Writing the TB

It is not possible to write the entire 64-bit TB using a single instruction. The TB can be written by a sequence such as the following:

```
lwzRx,upper  # load 64-bit value for
lwz Ry,lower # TB into Rx and Ry
addi Rz,R0,0
mtspr TBL,Rz # force TBL to 0
mtspr TBU,Rx # set TBU
mtspr TBL,Ry # set TBL
```

Provided that no interrupts occur while the last three instructions are being executed, loading 0 into time base lower prevents the possibility of a carry from time base lower to TBU while the TB is being initialized.

8.3.2 Reading the TB

The 64-bit time base can be read in a single instruction on 64-bit implementations by reading TBL. In 32-bit implementations the 64-bit time base cannot be read in a single instruction. For 32-bit implementations, **mfspr r**D**,TBL** reads the lower half into to a GPR, and **mfspr r**D**,TBU** reads the upper half. Because a carry from TBL to TBU can occur between reads, a sequence such as the following is necessary.

```
loop:
mfspr    Rx,TBU                                                #load from TBU
mfspr    Ry,TBL                                                #load from TBL
mfspr    Rz,TBU                                                #load from TBU
cmp      cr0,0,Rz,Rx                                    #see if 'old' = 'new'
bc       4,2,loop                                     #loop if carry occurred
```

The comparison and loop ensure that a consistent pair of values is obtained.

8.3.3 Computing Time of Day from the Time Base

Because the TB update frequency is implementation-dependent, the algorithm for converting the current value in the TB to time of day is also implementation-dependent.

For example, assume TB is constantly incremented once for every 32 cycles of a 100-MHz CPU instruction clock. What is wanted is the pair of 32-bit values comprising a POSIX standard clock: the number of whole seconds that have passed since midnight January 1, 1970, and the remaining fraction of a second expressed as a number of nanoseconds.

Assume the following:

- The value 0 in the TB represents the start time of the POSIX clock (if this is not true, a simple 64-bit subtraction makes it so).
- The integer constant ticks_per_sec contains the following value, which is the number of times the TB is updated each second:
  - — 100 MHz ÷ 32 = 3,125,000
- The integer constant ns_adj contains the following value, which is the number of nanoseconds per tick of the TB:
  - — 1,000,000,000 ÷ 3,125,000 = 320

The POSIX clock can be computed with an instruction sequence such as the following:

```
# Read Time Base
loop:
mfspr    Rx,TBU                                       #load from TBU into Rx
mfspr    Ry,TBL                                       #load from TBL into Ry
mfspr    Rz,TBU                                       #load from TBU into Rz
cmp      CR0,0,Rz,Rx                           #see if 'old TBU' = 'new TBU'
bc       4,2,loop                                     #loop if carry occurred
rldimi   Ry,Rx,32,0                                 #splice TBU & TBL into Ry
#
# Compute POSIX clock
#
lwz      Rx,ticks_per_sec
divd     Rz,Ry,Rx                                        #Rz = whole seconds
```

```
stw     Rz,posix_sec
mulld   Rz,Rz,Rx                                        #Rz = quotient * divisor
sub     Rz,Ry,Rz                                              #Rz = excess ticks
lwz     Rx,ns_adj
mulld   Rz,Rz,Rx                                        #Rz = excess nanoseconds
stw     Rz,posix_ns
```

In the absence of a **divd** <64>, **mulld** <64>, and **rldimi** <64> instructions in a 32-bit implementation, direct implementation of the algorithm given above is awkward.

Such division can be avoided entirely if a time of day clock in POSIX format is updated at least once each second.

Assume the following:

- The operating system maintains the following variables:
  - *posix_tb* (64 bits)
  - *posix_sec* (32 bits)
  - *posix_ns* (32 bits)

  These variables hold the value of the TB and the computed POSIX second and nanosecond values from the last time the POSIX clock was computed.
- The operating system arranges for an interrupt to occur at least once per second, at which time it recomputes the POSIX clock values.
- The integer constant billion contains the value 1,000,000,000.

The POSIX clock can be computed with an instruction sequence such as this:

```
loop:
mfspr   Rx,TBU                                                        #Rx = TBU
mfspr   Ry,TBL                                                        #Ry = TBL
mfspr   Rz,TBU                                            #Rz = 'new' TBU value
cmp     CR0,0,Rz,Rx                                       #see if 'old' = 'new'
bc      4,2,loop                                        #loop if carry occurred
# now have 64-bit TB in Rx and Ry
lwz     Rz,posix_tb+4
sub     Rz,Ry,Rz                                            #Rz = delta in ticks
lwz     Rw,ns_adj
mullw   Rz,Rz,Rw                                               #Rz = delta in ns
lwz     Rw,posix_ns
add     Rz,Rz,Rw                                              #Rz = new ns value
lwz     Rw,billion
cmp     CR0,0,Rz,Rw                                       #see if past 1 second
bc      12,0,nochange                                            #branch if not
sub     Rz,Rz,Rw                                            #adjust nanoseconds
lwz     Rw,posix_sec
addi    Rw,Rw,1                                                 #adjust seconds
stw     Rw,posix_sec                                        #store new seconds
nochange:
stw     Rz,posix_ns                                              #store new ns
stw     Rx,posix_tb                                        #store new time base
stw     Ry,posix_tb+4
```

Note that the upper half of the TB does not participate in the calculation to determine the new POSIX time of day. This is correct as long as the time change does not exceed one second.

8.3.4 Nonconstant Update Frequency

In a system in which the update frequency of the TB may change over time, it is not possible to convert an isolated TB value into time of day. Instead, a TB value has meaning only with respect to the current update frequency and the time of day that the update frequency was last changed. Each time the update frequency changes, either the system software is notified of the change by way of an interrupt (Chapter 7, “Interrupts and Exceptions,” describes register behavior caused by the decrementer interrupt), or the change was instigated by the system software itself. At each such change, the system software must compute the current time of day using the old update frequency, compute a new value of *ticks_per_sec* for the new frequency, and save the time of day, TB value, and tick rate. Subsequent calls to compute time of day use the current TB value and the saved data.

8.4 Decrementer Operation

As Figure 8-2 shows, the DEC decrements at the same rate that the TB increments. A decrementer event occurs when a decrement occurs on a decrementer value of 0x0000_0001. At this point, the decrementer has the following modes of operation, depending on whether the auto-reload function is enabled:

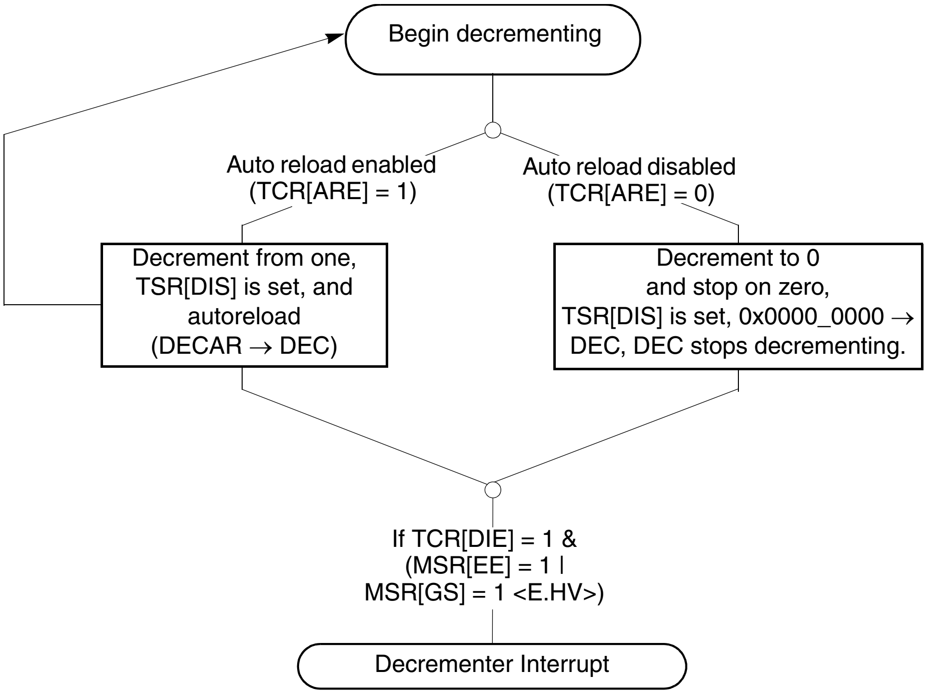


Figure 8-3. Decrementer Operation

- If TCR[ARE] is clear (auto-reload function disabled); decrement and stop on 0.
  When the decrement event occurs, the decrementer interrupt status bit (TSR[DIS]) is set and the value 0x0000_0000 is placed into the DEC, which then stops decrementing. It remains at 0 until software reloads it using an **mtspr** instruction.
  If the decrementer interrupt is enabled (TCR[DIE] and (MSR[EE] or MSR[GS] <E.HV>), the interrupt is taken. Section 7.8.10, “Decrementer Interrupt,” describes register the decrementer interrupt.

- If TCR[ARE] is set (auto-reload function enabled): Decrement and auto-reload.
  When the decrementer event occurs, TSR[DIS] is set and the DECAR contents are placed into DEC, which continues decrementing from the reloaded value.
  If the decrementer interrupt is enabled (TCR[DIE] and (MSR[EE] or MSR[GS] <E.HV>), the interrupt is taken. Section 7.8.10, “Decrementer Interrupt,” describes the decrementer interrupt.

The decrementer interrupt handler must reset TSR[DIS] to avoid taking another, redundant decrementer interrupt once MSR[EE] is re-enabled (assuming TCR[DIE] is not cleared instead). This is done by writing a word to TSR using **mtspr** with a 1 in the bit corresponding to TSR[DIS] (and any other bits that are to be cleared) and 0 in all other bits. The write-data to the TSR is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Forcing the decrementer to 0 using **mtspr** does not cause a decrementer exception; however, decrementing which was in progress at the instant of the **mtspr** may cause the exception. To eliminate the decrementer as a source of exceptions, set TCR[DIE] to 0 (clear the decrementer interrupt enable bit).

To eliminate all decrementer activity, the procedure is as follows:

1. Write 0 to TCR[DIE]. This prevents decrementer activity from causing exceptions.
2. Write 0 to TCR[ARE] to disable the decrementer auto-reload.
3. Write 0 to DEC. This halts decrementing. Although this action does not cause a decrementer exception to be set in TSR[DIS], a near simultaneous decrement may have done so.
4. Write 1 to TSR[DIS]. This action clears TSR[DIS] and removes any pending decrementer exception. Because the decrementer is frozen at zero, no further decrementer events are possible.

If the auto-reload feature is disabled (TCR[ARE] is cleared), when the DEC reaches zero, it stays there until software reloads it using the **mtspr** instruction.

On reset, TCR[ARE] is cleared, which disables the auto-reload feature.

8.5 Fixed-Interval Timer

The fixed-interval timer (FIT) is a mechanism for providing timer interrupts with a repeatable period, to facilitate system maintenance. It is similar in function to an auto-reload decrementer, except that there are fewer selections of interrupt period available. The fixed-interval timer exception occurs on 0 to 1 transitions of a selected bit from the time base.

The fixed-interval timer exception is logged by TSR[FIS]. A fixed-interval timer interrupt occurs if TCR[FIE] and (MSR[EE] or MSR[GS] <E.HV>) are enabled. See Section 7.8.11, “Fixed-Interval Timer Interrupt,” for details of register behavior caused by the fixed-interval timer interrupt.

The interrupt handler must reset TSR[FIS] via software, in order to avoid another fixed-interval timer interrupt once MSR[EE] is re-enabled (assuming TCR[FIE] is not cleared instead). This is done by writing a word to the TSR using **mtspr** with a 1 in the bit corresponding to TSR[FIS] (and any other bits that are to be cleared) and 0 in all other bits. The write-data to the TSR is not direct data, but is a mask. Writing a 1 causes the bit to be cleared; writing a 0 has no effect.

A fixed-interval timer interrupt also occurs if the selected time base bit changes from 0 to 1 due to an **mtspr** that writes a 1 to a bit that was previously a 0. The fixed-interval timer interrupt is described in Section 7.8.11, “Fixed-Interval Timer Interrupt.”

8.6 Watchdog Timer

The watchdog timer is a facility intended to aid system recovery from faulty software or hardware. Watchdog time-outs occur on 0 to 1 transitions of selected bits from the time base.

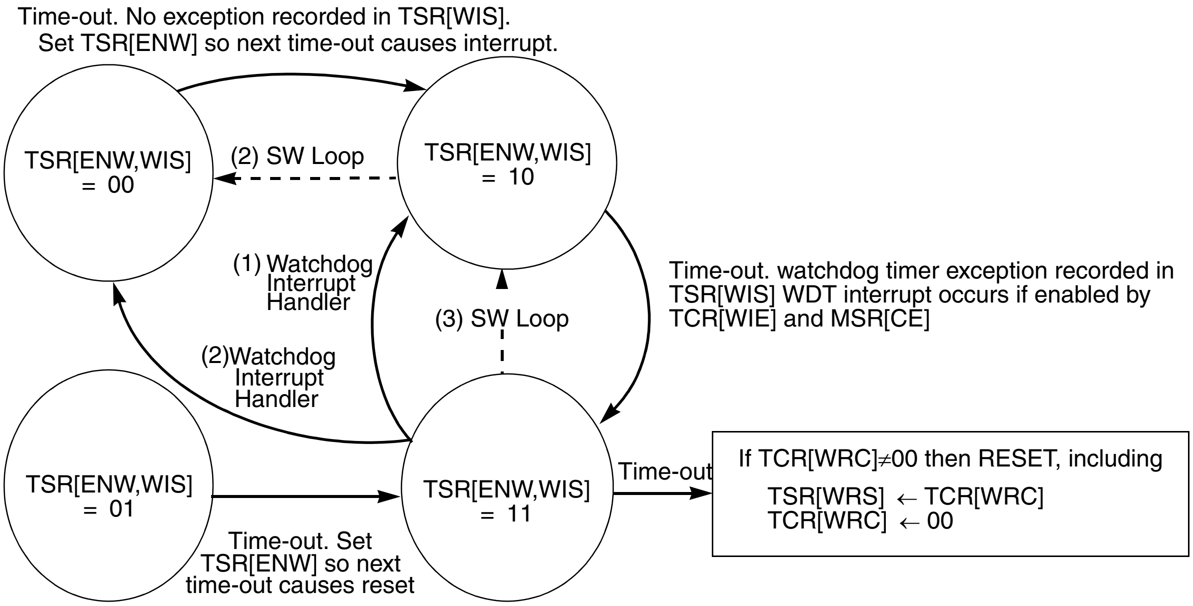
When a watchdog timer time-out occurs while watchdog timer interrupt status is clear (TSR[WIS] = 0) and the next watchdog time-out is enabled (TSR[ENW] = 1), a watchdog timer exception is generated and logged by setting TSR[WIS]. This is referred to as a watchdog timer first time out. A watchdog timer interrupt occurs if enabled by TCR[WIE] and (MSR[CE] or MSR[GS] <E.HV>). See Section 7.8.12, “Watchdog Timer Interrupt,” for details of the watchdog timer interrupt.

To avoid another watchdog timer interrupt once MSR[CE] is re-enabled (assuming TCR[WIE] is not cleared instead), the interrupt handler must reset TSR[WIS] through software. This is done by writing a word to TSR using **mtspr** with a 1 in the bit corresponding to TSR[WIS] (and any other bits that are to be cleared) and a 0 in all other bits. The write-data to the TSR is not direct data, but is a mask. A 1 causes the bit to be cleared; a 0 has no effect.

Note that a watchdog timer exception also occurs if the selected time base bit transitions from 0 to 1 due to an **mtspr** that writes a 1 to the bit when its previous value was 0.

When a watchdog timer time-out occurs while TSR[WIS] = 1 and TSR[ENW] = 1, a processor reset occurs if it is enabled by a nonzero value of the watchdog reset control field (TCR[WRC]). This is referred to as a watchdog timer second time out. The assumption is that TSR[WIS] was not cleared because the processor was unable to execute the watchdog timer interrupt handler, leaving reset as the only available means to restart the system. Note that once TCR[WRC] has been set to a nonzero value, it cannot be reset by software; this feature prevents errant software from disabling the watchdog timer reset capability.

This figure shows the watchdog timer state diagram.


Time-out. No exception recorded in TSR[WIS].
Set TSR[ENW] so next time-out causes interrupt.
TSR[ENW,WIS] = 00
(2) SW Loop
TSR[ENW,WIS] = 10
(1) Watchdog Interrupt Handler
(3) SW Loop
Time-out. watchdog timer exception recorded in TSR[WIS] WDT interrupt occurs if enabled by TCR[WIE] and MSR[CE]
(2)Watchdog Interrupt Handler
TSR[ENW,WIS] = 01
TSR[ENW,WIS] = 11
Time-out
If TCR[WRC]≠00 then RESET, including
TSR[WRS] ← TCR[WRC]
TCR[WRC] ← 00
Time-out. Set TSR[ENW] so next time-out causes reset

1 Always take the watchdog timer interrupt when pending, and never attempt to prevent its occurrence. In this mode, the watchdog timer interrupt caused by a first time-out is used to clear TSR[WIS] so a second time-out never occurs. TSR[ENW] is not cleared, thereby allowing the next time-out to cause another interrupt.

2 Always take the watchdog timer interrupt when pending, but avoid when possible. In this mode, a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the fixed-interval timer interrupt handler) is used to repeatedly clear TSR[ENW] such that a first time-out exception is avoided, and thus no watchdog timer interrupt occurs. Once TSR[ENW] has been cleared, software has between one and two full watchdog periods before a watchdog exception is posted in TSR[WIS]. If this occurs before the software can clear TSR[ENW] again, a watchdog timer interrupt occurs. In this case, the watchdog timer interrupt handler then clears both TSR[ENW] and TSR[WIS], in an attempt to avoid the next watchdog timer interrupt.

3 Never take the watchdog timer interrupt. In this mode, watchdog timer interrupts are disabled (TCR[WIE] is cleared), and the system depends upon a recurring code loop of reliable duration (or perhaps a periodic interrupt handler such as the fixed-interval timer interrupt handler) to repeatedly clear TSR[WIS] such that a second time-out is avoided, and thus no reset occurs. TSR[ENW] is not cleared, thereby allowing the next time-out to set TSR[WIS] again. The recurring code loop must have a period that is less than one watchdog timer period to guarantee that a watchdog timer reset does not occur.

Figure 8-4. Watchdog State Machine

This table describes the watchdog timer state machine and watchdog timer controls.

Table 8-1. Watchdog Timer Controls

Enable Next Watchdog Timer (TSR[ENW])	Timer Status (TSR[WIS])	Action When Timer Interval Expires
0	0	Set enable next watchdog timer (TSR[ENW] is set).
0	1	Set enable next watchdog timer (TSR[ENW] is set).

Table 8-1. Watchdog Timer Controls (continued)

Enable Next Watchdog Timer (TSR[ENW])	Timer Status (TSR[WIS])	Action When Timer Interval Expires
1	0	Set watchdog timer interrupt status bit (TSR[WIS] is set). If the watchdog timer interrupt is enabled (TCR[WIE] = 1 and MSR[CE] = 1), then interrupt.
1	1	Cause watchdog timer reset action specified by TCR[WRC]. Reset copies prereset TCR[WRC] into TSR[WRS], then clears TCR[WRC].

The controls described in Table 8-1 imply three different modes of operation that a programmer might select for the watchdog timer. Each of these modes assumes that TCR[WRC] has been set to allow processor reset by the watchdog facility.

8.7 Alternate Time Base

The alternate time base counter (ATB) is formed by concatenating the upper and lower alternate time base registers (ATBU and ATBL), described in Section 3.7.6, “Alternate Time Base Registers (ATB).” It contains a 64-bit unsigned integer that is incremented periodically at an implementation-defined frequency. Like the TB implementation, ATBL is an aliased name for ATB and provides read-only access to the 64-bit alternate time base counter.

The alternate time base increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1). At the next increment, its value becomes 0x0000_0000_0000_0000. There is no explicit indication (such as an interrupt) that this has occurred.

The effect of entering a power-savings mode or of processor frequency changes on counting in the alternate time base is implementation-dependent.

8.8 Freezing the Timer Facilities

The debug mechanism provides a means of temporarily freezing the timers when a debug event occurs. Specifically, the time base and decrementer can be frozen and prevented from incrementing/decrementing, respectively, whenever a debug event is set in the DBSR. This allows a debugger to simulate the appearance of real time, even though the application has been temporarily halted to service the debug event. See the description of DBCR0[FT] (freeze timers on debug event) in Section 3.13.2.1, “Debug Control Register 0 (DBCR0).”

Chapter 9 Debug Facilities

This chapter describes the internal debug features defined by the EIS. It does not describe external hardware debug features, although processors generally include such a facility.

9.1 Debug Facilities Overview

Processors provide debug facilities to enable hardware and software debug functions, such as instruction and data breakpoints and program single stepping. The debug facilities consist of the following:

- A set of debug control registers (DBCR0–DBCR6)
  See Section 3.13.2, "Debug Control Registers (DBCR0–DBCR6)."
- A set of address and data value compare registers (IAC1–IAC8, DAC1, DAC2, DVC1, and DVC2)
  See Section 9.7.2, "Instruction Address Compare Debug Event," and Section 9.7.3, "Data Address Compare Debug Event."
- A debug status register (DBSR/DBSRWR) for enabling and recording various debug events
  See Section 3.13.4, "Debug Status Register (DBSR/DBSRWR)."
- A special debug interrupt type built into the interrupt mechanism
  See Section 7.8.15, "Debug Interrupt."

The debug facilities also provide a mechanism for software-controlled processor reset and for controlling the timers in a debug environment.

The **mfspr** and **mtspr** instructions (see Section 4.6.1.13.2, "Move to/from Special-Purpose Register Instructions") provide access to the debug SPRs.

Along with the facilities described here, implementations typically include implementation-specific debug facilities, modes, and access mechanisms. For example, implementations typically provide external access to the debug facilities through a dedicated interface such as the Test Access Port (JTAG) conforming to IEEE Std 1149.1™.

NOTE

> Not all implementations will provide all the debug facilities described here. For more information, see the core reference manual.

9.2 Internal and External Debug

The internal debug facilities are used by software when the processor is in "internal debug mode" (DBCR0[IDM]). Internal debug mode only functions when the processor is not in the halted state. Freescale defines another set of debug facilities that are accessible through an external interface to the processor, that can access the resources that software normally does as well as extended resources to control execution and read/write architected and implementation dependent resources. These facilities are defined as the "external debug" facilities and are not defined here, however some instructions and registers share internal and external debug roles and are briefly described as necessary. When the processor is in external debug mode, DBCR0[EDM] is set. DBCR0[EDM] is only visible to software running on the processor and cannot be changed by software. The setting of DBCR0[EDM] takes precedence over

DBCR0[IDM]. If both are set, the processor is still considered to be in external debug mode, and it is implementation dependent how internal debug facilities operate.

The “halted” state occurs when the processor is controlled through the external debug facility and the external debug facility commands the processor to halt or a **dnh** instruction is executed in the non halted state and the external debug facility has enabled **dnh** instructions to enter the halted state on execution. When a processor is in the halted state, normal software instruction execution is frozen, instructions are not fetched, interrupts are not taken, and the core does not execute instructions from the architectural instruction stream and control of the processor is managed by the external debug facility.

Unless otherwise mentioned, the rest of this chapter assumes the processor is in internal debug mode.

9.3 Programming Model

9.3.1 Debug Registers

The internal debug facility registers are fully described in Section 3.13, “Debug Registers.” Several of the EIS defined debug facility registers contain implementation-specific fields and registers. Consult the core reference manual for processor specific information.

9.3.2 Debug Instructions

The Debugger Notify Halt instruction **dnh)** provides a bridge between the execution of instructions on the core in a non-halted mode, and the external debug facility. The **dnh** instruction allows software to transition the core from a running state to a halted state if enabled by an external debugger, and to notify the external debugger with both a message passed to the external debugger and bits reserved in the instruction itself to pass additional information.

Execution of **dnh** causes the processor to stop fetching instructions and taking interrupts if the instruction has been enabled by the external debug facility to halt on **dnh**. The contents of the DUI field are sent to the external debug facility to identify the reason for the halt. The **dnh** instruction is described in Chapter 5, “Instruction Set.”

9.3.3 Debug Interrupt

A debug interrupt occurs when no higher priority exception exists, a debug exception is presented to the interrupt mechanism, and MSR[DE] = 1. Debug exceptions and interrupts are caused by debug events that occur which set specific bits in DBSR.

The debug interrupt is fully described in Section 7.8.15, “Debug Interrupt.”

The debug interrupt uses DSRR0 and DSRR1 to save interrupted state when category E.ED is implemented. If category E.ED is not implemented, the debug interrupt uses CSRR0 and CSRR1. Category E.ED allows the debug features to be used to debug interrupts that use CSRR0 and CSRR1 to save interrupted state.

9.4 Enabling Internal Debug Mode

Debug events can occur as the result of a specific “debug condition” in the processor which are appropriately enabled in the DBCR*n* registers. This includes such things as instruction and data breakpoints that, when they occur, set DBSR status bits. A DBSR bit being set is considered a debug exception, which, if enabled, causes a debug interrupt. MSR[DE] and the internal debug mode bit (DBCR0[IDM]) must both be set to enable debug interrupts:

- DBCR0[IDM] = 1 (internal debug mode) and MSR[DE] = 1. Debug events cause debug interrupts. Software at the debug interrupt vector is given control upon the occurrence of a debug event and can access (using standard instructions) all architected resources and controls the processor, gathers status, and interacts with debug hardware connected to the processor.
- DBCR0[IDM] = 1 (internal debug mode) and MSR[DE] = 0. Debug conditions do not cause debug events or debug interrupts. Previous versions of the architecture allowed debug events to be recorded in DBSR in this case and when MSR[DE] is later set to 1, cause a delayed imprecise debug interrupt if the DBSR bit(s) are still set. Therefore some processors may support such imprecise debug events, however, newer processors will generally not support such a feature and software which is not processor specific should not depend on such a feature.
- DBCR0[IDM] = 0 (internal debug mode disabled). Some processors allow debug events to occur and be recorded in the DBSR from which they can be monitored by software. This capability and software which uses this feature on any given processor is implementation-dependent. A debug interrupt may result later when DBCR0[IDM] is set (if MSR[DE] = 1). Processor behavior when debug events occur while DBCR0[IDM] = 0 is implementation-dependent.

NOTE: Architecture

The “imprecise” or “delayed” debug interrupts other than UDE, will likely be removed from a future version of Power ISA.

NOTE: Implementation

New implementations should only set DBSR bits when a debug condition occurs, the event is enabled, and MSR[DE] = 1 or through a **mtspr** to DBSRWR <E.HV>.

9.5 External Debug Mode

In external debug mode, facilities external to the processor can access processor resources and control execution. These external debug facilities are not defined here, however some instructions and registers share internal and external debug roles and are briefly described as necessary.

The **dnh** instruction is provided to stop instruction fetching and execution and to allow the processor to be managed by an external debug facility.

9.6 Dealing with Debug Resources

Software should be aware of these issues when dealing with debug resources.

9.6.1 Synchronizing Changes to Debug Facility Registers

Special care must be taken when manipulating debug registers which control the debug facility. Modifying debug facility registers requires special synchronization. This is because many of the debug conditions deal with address space, fetch addresses, and data addresses which must be synchronized with respect to processor execution. To perform any modification to a debug facility register with **mtspr**, software must perform the following actions to ensure that the change is guaranteed to be synchronized with respect to processor execution:

- MSR[DE] must transition from 0 to 1;
- A context synchronizing instruction or event must occur.
  See Section 4.5.4.4, "Context Synchronization."

The results of the modification are not guaranteed to be seen until both events occur. However, the modification may be seen at any time after the debug register was written up until the synchronization is completed.

NOTE: Software Considerations

Software should only make changes to the debug facility registers while in the debug interrupt handler or when MSR[DE] = 0. Doing so in the debug interrupt handler works naturally since MSR[DE] is set to 0 on entry to the handler, and on return the **rfci** (**rfdi** <E.ED>) naturally will set MSR[DE] to 1 and perform a context synchronization.

9.6.2 Pre-Completion and Post-Completion Interrupts

Debug interrupts can occur either before or after execution of an instruction which sets a debug event, as described in the following sections.

9.6.2.1 Synchronous Pre-Completion Debug Interrupts

All debug interrupts, except for those generated by instruction complete exceptions (and any implementation-dependent debug events which are defined as post-completion) and UDE which is asynchronous are pre-completion exceptions. These exceptions all take the interrupt upon encountering an excepting instruction without updating any architectural state other than DBSR and any register updated by the interrupt mechanism. That is, the instruction does not complete normally and the results of the instruction execution are suppressed for that instruction.

The CSRR0 (DSRR0 <E.ED>) for this type of exception points to the instruction that encountered the exception. This includes IAC, DAC, branch taken, and so on.

The following steps show how forward progress is made for any pre-completion debug exception:

1. Software sets up pre-completion exceptions (for example branch taken debug exceptions) and then returns to normal program operation.
2. Hardware takes debug interrupt upon the first branch taken debug exception, pointing to the branch with CSRR0 (DSRR0 <E.ED>)

3. Software, in the debug handler, sees the branch taken exception type, does whatever logging/analysis it wants to, then clears all debug event enables in the DBCR except for the instruction complete debug event enable.
4. Software does an **rfci** (**rfdi** <E.HV>)
5. Hardware would execute and complete one instruction (the branch taken in this case), and then take a debug interrupt with CSRR0 (DSRR0 <E.ED>) pointing to the target of the branch.
6. Software would see the instruction complete interrupt type. It clears the instruction complete event enable, then enables the branch taken interrupt event again.
7. Software does an **rfci** (**rfdi** <E.HV>)
8. Hardware resumes on the target of the taken branch and continues until another taken branch, in which case we end up at step 2 again.

That two debug interrupts occur for every instance of a pre-completion exception may seem like duplication, but this approach ensures forward progress on pre-completion debug exceptions.

9.6.2.2 Synchronous Post-Completion Debug Interrupts

A post-completion exception is taken after an instruction completes and writes it results to architected state. The instruction complete debug exception causes a post-completion debug interrupt. The CSRR0 (DSRR0 <E.ED>) for this type of exception points to the instruction after the instruction that caused the exception (it points to the instruction that would have executed next).

9.6.2.3 Asynchronous Debug Interrupts

An asynchronous exception is after one instruction completes, but before the next instruction is completed. The unconditional debug exception (UDE) causes an asynchronous debug interrupt. The CSRR0 (DSRR0 <E.ED>) for this type of exception points to the instruction that would have executed next had the asynchronous debug interrupt not occurred. The asynchronous interrupt is not associated with an instruction since it is considered asynchronous.

9.7 Debug Conditions and Debug Events

A “debug condition” indicates that a set of specific criteria have been met such that a corresponding debug event occurs in the absence of any gating or masking. The criteria for debug conditions are obtained from debug control registers. For a specific debug condition to occur, the corresponding debug event enable bit must be set in the DBCR*n* register (see Section 3.13.2, “Debug Control Registers (DBCR0–DBCR6)”) and the criteria specific to the debug condition (for example, DBCR0[BRT] must be set and the processor must take a branch for the branch taken condition to occur) must be present.

A “debug event” means the setting of a bit in the debug status register (DBSR) upon the occurrence of the associated debug condition. However, a debug condition does not always result in a debug event. Conditions are prioritized with respect to exceptions. Exceptions that have higher priority than a debug condition prevent the debug condition from being recorded as a debug event. In internal debug mode (IDM), debug events cause a debug interrupt if the debug enable bit is set (MSR[DE] = 1).

Debug events are used to cause debug exceptions to be recorded in the DBSR (see Section 3.13.4, “Debug Status Register (DBSR/DBSRWR)”). For a debug event to be enabled to set a DBSR bit and thereby cause a debug exception, the corresponding event type must be enabled in a DBCR*n* register (see Section 3.13.2, “Debug Control Registers (DBCR0–DBCR6)”). The unconditional debug event (UDE) is an exception to this rule and requires no corresponding enable bit in DBCR*n*. If a DBSR*n* bit is set and debug interrupts are enabled (MSR[DE] = 1), a debug interrupt is generated.

NOTE: Software Considerations

> Software should not depend on any DBSR bit other than UDE being set when MSR[DE] = 0.

Debug interrupts are ordered with respect to other interrupt types (Section 7.10, “Interrupt Ordering and Masking”). Debug exceptions are prioritized with respect to other exceptions (Section 7.11, “Exception Priorities”).

There are eight types of debug events defined:

1. Instruction address compare debug events
2. Data address compare debug events
3. Trap debug events
4. Branch taken debug events
5. Instruction complete debug
6. Interrupt taken debug events
7. Critical interrupt taken debug events <E.ED>
8. Interrupt return debug events
9. Critical interrupt return debug events <E.ED>
10. Unconditional debug events

9.7.1 Suppressing Debug Events in Hypervisor Mode

Synchronous debug conditions (and thus events) can be suppressed when executing in hypervisor state. This prevents debug events from being recorded (and subsequent debug interrupts from occurring) when executing in hypervisor state when the guest operating system is using the debug facility.

When EPCR[DUVD] = 1 and MSR[GS] = 0, all debug conditions, except the unconditional debug conditions, are suppressed and are not posted in the DBSR and the associated exceptions do not occur.

NOTE: Virtualization

> The EPCR[DUVD] bit is provided for the hypervisor to give debug control to the guest operating system because the guest operating system wishes to use the debug facilities. This prevents the hypervisor from recording false and spurious debug events (and interrupts) that would be taken by the hypervisor and filtered out for the guest. While it is possible for the hypervisor to do this, it is impractical and requires tricky code which may be in important performance paths.

9.7.2 Instruction Address Compare Debug Event

One or more instruction address compare debug conditions (IAC*n*) occur if they are enabled and execution is attempted of an instruction at an address that meets the criteria specified in the DBCR*n* and IAC*n* registers.

9.7.2.1 Instruction Address Compare User/Supervisor Mode

DBCR1[IAC*n*US] specifies whether corresponding IAC*n* debug conditions can occur in user or supervisor mode, or both.

9.7.2.2 Effective/Real Address Mode

DBCR1[IAC*n*ER] specifies whether effective addresses, physical addresses, effective addresses and MSR[IS] = 0, or effective addresses and MSR[IS] = 1 are used in determining an address match on corresponding IAC*n* debug conditions.

Many processors do not support real address comparisons. Consult the core reference manual.

9.7.2.3 Instruction Address Compare Mode

DBCR1[IAC12M] specifies whether all or some of the bits of the address of the instruction fetch must match the contents of the IAC1 or IAC2, whether the address must be inside a specific range specified by the IAC1 and IAC2 or outside a specific range specified by the IAC1 and IAC2 for an IAC1 or IAC2 debug condition to occur.

DBCR1[IAC34M] specifies whether all or some of the bits of the address of the instruction fetch must match the contents of the IAC3 or IAC4, whether the address must be inside a specific range specified by IAC3 and IAC4 or outside a specific range specified by IAC3 and IAC4 for an IAC3 or IAC4 debug condition to occur.

There are four instruction address compare modes, as follows:

- Exact address compare mode.
  If the address of the instruction fetch matches the value in the enabled IAC*n*, an instruction address match occurs. For 64-bit implementations, the addresses are masked to compare only bits 32–63 when the processor is executing in 32-bit mode.
- Address bit match mode.
  For IAC1 and IAC2 debug conditions, if the address of the instruction fetch access, ANDed with the contents of the IAC2, match the contents of the IAC1, also ANDed with the contents of the IAC2, an instruction address match occurs.
  For IAC3 and IAC4 debug conditions, if the address of the instruction fetch, ANDed with the contents of the IAC4, are equal to the contents of the IAC3, also ANDed with the contents of the IAC4, an instruction address match occurs.
  For 64-bit implementations, the addresses are masked to compare only bits 32–63 when the processor is executing in 32-bit mode.
- Inclusive address range compare mode.
  For IAC1 and IAC2 debug conditions, if the address of the instruction fetch is greater than or equal

to the contents of the IAC1 and less than the contents of the IAC2, an instruction address match occurs.
For IAC3 and IAC4 debug conditions, if the address of the instruction fetch is greater than or equal to the contents of the IAC3 and less than the contents of the IAC4, an instruction address match occurs.
For 64-bit implementations, addresses are masked to compare only bits 32–63 when the processor is executing in 32-bit mode.

- Exclusive address range compare mode.
  For IAC1 and IAC2 debug conditions, if the address of the instruction fetch is less than the contents of the IAC1 or greater than or equal to the contents of the IAC2, an instruction address match occurs.
  For IAC3 and IAC4 debug conditions, if the address of the instruction fetch is less than the contents of the IAC3 or greater than or equal to the contents of the IAC4, an instruction address match occurs.
  For 64-bit implementations, the addresses are masked to compare only bits 32–63 when the processor is executing in 32-bit mode.

9.7.2.4 Instruction Address Compare Debug Event Considerations

<Embedded.Hypervisor>:
When MSR[GS] = 0 and EPCR[DUVD] = 1, instruction address compare debug conditions are suppressed.

Some processors implement additional IAC*n* registers and additional DBCR*n* registers for additional instruction address compare debug conditions.

When an instruction address compare debug condition occurs, if DBCR0[IDM] and MSR[DE] are set and the instruction address compare debug exception is the highest priority exception, the debug event is said to occur and the corresponding DBSR[IAC*n*] bit or bits are set to record the debug exception.

When the instruction address compare mode is anything other than exact address compare mode, it is implementation-dependent if both or either associated DBSR[IAC*n*] bits may be set. For IAC1 and IAC2 debug events, either or both DBSR[IAC1] and DBSR[IAC2] bits may be set. For IAC3 and IAC4 debug events, either or both DBSR[IAC3] and DBSR[IAC4] bits may be set.

If MSR[DE] = 1 at the time of the instruction address compare debug exception, a debug interrupt occurs immediately and execution of the instruction causing the exception is suppressed, and CSRR0 (DSRR0 <E.ED>) is set to the address of the excepting instruction.

If a instruction address compare event causes a delayed debug exception, a debug interrupt does not occur, and the instruction completes execution (provided the instruction is not causing some other exception which generates an enabled interrupt).

Later, if the delayed debug exception has not been reset by clearing the corresponding DBSR[IAC*n*] bit(s) and MSR[DE] is set, a delayed debug interrupt occurs. In this case, CSRR0 (DSRR0 <E.ED>) contains the address of the instruction after the one which enabled the debug interrupt by setting MSR[DE]. Software in the debug interrupt handler can observe DBSR[IDE] to determine how to interpret the value in CSRR0 (DSRR0 <E.ED>).

9.7.3 Data Address Compare Debug Event

One or more data address compare debug conditions (DAC1R, DAC1W, DAC2R, DAC2W) occur if they are enabled, execution is attempted of a data storage access instruction, and the type and address of the data storage access meet the criteria specified in the DBCR0, DBCR2, and the DAC*n* registers.

9.7.3.1 Data Address Compare Read/Write Enable

DBCR0[DAC*n*] specifies whether the corresponding DAC*n*R debug conditions can occur on read-type data storage accesses and whether DAC*n*W debug conditions can occur on write-type data storage accesses.

NOTE: Implementation

> If a processor implements indexed string instructions (**lswx**, **stswx**) for which the XER field specifies zero bytes as the length of the string, these should be treated as no-ops and are not allowed to cause data address compare debug events.

All load instructions are considered reads with respect to debug conditions, while all store instructions are considered writes with respect to debug conditions. Additionally, the cache management instructions, and certain special cases, are handled as follows.

- **dcbt**, **dcbtls** <E.CL>, **dcbtep** <E.PD>, **dcbtst**, **dcbtstls** <E.CL>, **dcbtstep** <E.PD>, **icbt**, **icbtls** <E.CL>, **icbi, icblc** <E.CL>, **dcblc** <E.CL>, and **icbiep** <E.PD> are all considered reads with respect to debug conditions. Note that **dcbt**, **dcbtep** <E.PD>, **dcbtst**, **dcbtstep** <E.PD>, and **icbt** are treated as no-ops when they report data storage or data TLB miss exceptions, instead of being allowed to cause interrupts. However, these instructions are allowed to cause debug conditions, even when they would otherwise have been no-op'ed due to a data storage or data TLB miss exception. It is implementation-dependent if such no-ops cause debug conditions.
- **dcbz**, **dcbzl** <DEO>, **dcbzep** <E.PD>, **dcbzlep** <DEO,E.PD> **dcbi, dcbf**, **dcbfep** <E.PD>, **dcba**, **dcbal** <DEO>, **dcbst,** and **dcbstep** <E.PD> are all considered writes with respect to debug conditions. Note that **dcbf, dcbfep** <E.PD>, **dcbst**, and **dcbstep** <E.PD> are considered reads with respect to data storage exceptions, since they do not actually change the data at a given address. However, since the execution of these instructions may result in write activity on the processor's data bus, they are treated as writes with respect to debug events.

9.7.3.2 Data Address Compare User/Supervisor Mode

DBCR2[DAC*n*US] specifies whether DAC*n*R and DAC*n*W debug conditions can occur in user mode or supervisor mode, or both.

9.7.3.3 Effective/Real Address Mode

DBCR2[DAC*n*ER] specifies whether effective addresses, physical addresses, effective addresses and MSR[DS] = 0, or effective addresses and MSR[DS] = 1 are used to in determining an address match on DAC*n*R and DAC*n*W debug conditions.

Many processors do not support real address comparisons. Consult the core reference manual.

9.7.3.4 Data Address Compare Mode

DBCR2[DAC12M] specifies whether all or some of the bits of the address of the data storage access must match the contents of the DAC1 or DAC2, whether the address must be inside a specific range specified by the DAC1 and DAC2 or outside a specific range specified by the DAC1 and DAC2 for a DAC1R, DAC1W, DAC2R or DAC2W debug condition to occur.

There are four data address compare modes.

- Exact address compare mode. If the 64-bit address of the data storage access matches the value in the DAC*n* register, a data address match occurs. For 64-bit implementations, the addresses are masked to compare only bits 32–63 when the processor is executing in 32-bit mode.

  The address must exactly match and not merely be affected by the storage access. Instructions whose data address and associated access size encompasses the address specified by a DAC*n* register for which the address specified by the instruction does not exactly match the DAC*n* register, will not cause a data address compare condition.

  NOTE: Software Considerations

  For example, in exact address compare mode, a storage access of a word (4 bytes) to address 0x1000 will not cause a data address compare debug condition for DAC1 if DAC1 contains the value 0x1001. The same constraints apply to any storage access including cache management instructions and load/store multiple instructions.

- Address bit match mode. If the address of the data storage access, ANDed with the contents of the DAC2, are equal to the contents of the DAC1, also ANDed with the contents of the DAC2, a data address match occurs. For 64-bit implementations, addresses are masked to compare only bits 32–63 when the processor is executing in 32-bit mode.
- Inclusive address range compare mode. If the 64-bit address of the data storage access is greater than or equal to the contents of the DAC1 and less than the contents of the DAC2, a data address match occurs. For 64-bit implementations, the addresses are masked to compare only bits 32–63 when the processor is executing in 32-bit mode.
- Exclusive address range compare mode. If the 64-bit address of the data storage access is less than the contents of the DAC1 or greater than or equal to the contents of the DAC2, a data address match occurs. For 64-bit implementations, the addresses are masked to compare only bits 32–63 when the processor is executing in 32-bit mode.

9.7.3.5 Data Value Compare Mode

$\mathrm{DBCR2}_{\mathrm{DVC1M}}$ and $\mathrm{DBCR2}_{\mathrm{DVC1BE}}$ specify whether and how the data value being accessed by the storage access must match the contents of the DVC1 for a DAC1R or DAC1W debug condition to occur.

$\mathrm{DBCR2}_{\mathrm{DVC2M}}$ and $\mathrm{DBCR2}_{\mathrm{DVC2BE}}$ specify whether and how the data value being accessed by the storage access must match the contents of the DVC2 for a DAC2R or DAC2W debug condition to occur.

9.7.3.6 Data Address Compare Debug Event Considerations

The description of DBCR0 (see Section 3.13.2.1, “Debug Control Register 0 (DBCR0)”) and DBCR2 (see Section 3.13.2.3, “Debug Control Register 2 (DBCR2)”) control the modes for detecting data address compare debug events.

<Embedded.Hypervisor>:
When MSR[GS] = 0 and EPCR[DUVD] = 1, data address compare debug conditions are suppressed.

<Embedded.Hypervisor, Embedded.External PID>:
When MSR[GS] = 0 and EPCR[DUVD] = 1, data address debug compare conditions are suppressed when external PID instructions are used, even if the external PID instructions target a context where EPSC[EGS] or EPLC[EGS] is set to 1.

When a data address compare debug condition occurs, if DBCR0[IDM] and MSR[DE] are set and the data address compare debug exception is the highest priority exception, the debug event is said to occur and the corresponding DBSR[DAC*n*R,DAC*n*W] bit or bits are set to record the debug exception. On processors that support delayed debug interrupts, if MSR[DE] = 0, the corresponding DBSR[IAC*n*] bit or bits are set to record the debug exception and DBSR[IDE] is set to record the imprecise debug event and a delayed debug exception is pending until MSR[DE] is later set.

When the data address compare mode is anything other than exact address compare mode, it is implementation-dependent if both or either associated DBSR[DAC*n*] bits may be set. For DAC1 and DAC2 debug events, either or both DBSR[DAC1] and DBSR[DAC2] bits may be set.

If MSR[DE] = 1 at the time of the data address compare debug exception, a debug interrupt occurs immediately and execution of the instruction causing the exception is suppressed, and CSRR0 (DSRR0 <E.ED>) is set to the address of the excepting instruction. Depending on the type of instruction and/or the alignment of the data access, the instruction causing the exception may have been partially executed (see Section 7.9, “Partially Executed Instructions”).

If a data address compare event causes a delayed debug exception, a debug interrupt does not occur, and the instruction completes execution (provided the instruction is not causing some other exception which generates an enabled interrupt).

Later, if the delayed debug exception has not been reset by clearing the corresponding DBSR[DAC*n*R,DAC*n*W] bit(s) and MSR[DE] is set, a delayed debug interrupt occurs. In this case, CSRR0 (DSRR0 <E.ED>) contains the address of the instruction after the one which enabled the debug interrupt by setting MSR[DE]. Software in the debug interrupt handler can observe DBSR[IDE] to determine how to interpret the value in CSRR0 (DSRR0 <E.ED>).

9.7.4 Trap Debug Event

A trap debug condition (TRAP) occurs if DBCR0[TRAP] = 1 (that is trap debug events are enabled) and a trap instruction (**tw**, **twi**, **td** <64>, **tdi** <64>) is executed and the conditions specified by the instruction for the trap are met.

<Embedded.Hypervisor>:
When MSR[GS] = 0 and EPCR[DUVD] = 1, trap debug conditions are suppressed.

When a trap debug condition occurs, if DBCR0[IDM] and MSR[DE] are set and the trap debug exception is the highest priority exception, the debug event is said to occur and the corresponding DBSR[TRAP] bit is set to record the debug exception.

When a trap debug condition occurs, if DBCR0[IDM] = 1 and MSR[DE] = 0, a debug interrupt does not occur and a trap exception type program interrupt occurs instead. On processors that support delayed debug interrupts, the corresponding DBSR[TRAP] bit is set to record the debug exception and DBSR[IDE] is set to record the imprecise debug event and a delayed debug exception is pending until MSR[DE] is later set.

If MSR[DE] = 1 at the time of the trap debug exception, a debug interrupt occurs immediately and execution of the instruction causing the exception is suppressed, and CSRR0 (DSRR0 <E.ED>) is set to the address of the excepting instruction.

If a trap debug event causes a delayed debug exception, a debug interrupt does not occur, and the instruction completes execution (provided the instruction is not causing some other exception which generates an enabled interrupt).

Later, if the delayed debug exception has not been reset by clearing the corresponding DBSR[TRAP] bit and MSR[DE] is set, a delayed debug interrupt occurs. In this case, CSRR0 (DSRR0 <E.ED>) contains the address of the instruction after the one which enabled the debug interrupt by setting MSR[DE]. Software in the debug interrupt handler can observe DBSR[IDE] to determine how to interpret the value in CSRR0 (DSRR0 <E.ED>).

9.7.5 Branch Taken Debug Event

A branch taken debug condition (BRT) occurs if DBCR0[BRT] = 1 (that is branch taken debug events are enabled), execution is attempted of a branch instruction whose direction is taken (that is, either an unconditional branch, or a conditional branch whose branch condition is met), and MSR[DE] = 1.

<Embedded.Hypervisor>:
When MSR[GS] = 0 and EPCR[DUVD] = 1, branch taken debug conditions are suppressed.

When a branch taken debug condition occurs, if DBCR0[IDM] and MSR[DE] are set and the branch taken debug exception is the highest priority exception, the debug event is said to occur and the corresponding DBSR[BRT] bit is set to record the debug exception.

If DBCR0[IDM] and MSR[DE] are not set when a branch taken debug condition occurs, no event is recorded in DBSR[BRT]. DBSR[IDE] cannot be set by a branch taken debug event. This is because branch instructions occur very frequently. Allowing these common events to be recorded as exceptions in the DBSR while debug interrupts are disabled via MSR[DE] has little value.

If MSR[DE] = 1 at the time of the branch taken debug exception, a debug interrupt occurs immediately and execution of the instruction causing the exception is suppressed, and CSRR0 (DSRR0 <E.ED>) is set to the address of the excepting instruction.

9.7.6 Instruction Complete Debug Event

An instruction complete debug condition (ICMP) occurs if DBCR0[ICMP] = 1 (that is instruction complete debug events are enabled), execution of any instruction is completed, and MSR[DE] = 1.

<Embedded.Hypervisor>:
When MSR[GS] = 0 and EPCR[DUVD] = 1, instruction complete debug conditions are suppressed.

When an instruction complete debug condition occurs, if DBCR0[IDM] and MSR[DE] are set and the instruction complete debug exception is the highest priority exception, the debug event is said to occur and the corresponding DBSR[ICMP] bit is set to record the debug exception. Note that if execution of an instruction is suppressed due to the instruction causing some other exception which is enabled to generate an interrupt, then the attempted execution of that instruction does not cause an instruction complete debug event. The **sc** instruction does not fall into the type of an instruction whose execution is suppressed, since the instruction actually completes execution and then generates a system call interrupt. In this case, the instruction complete debug exception is also set.

If DBCR0[IDM] and MSR[DE] are not set when an instruction complete debug condition occurs, no event is recorded in DBSR[ICMP]. DBSR[IDE] cannot be set by an instruction complete debug event. This is because instruction completion occurs on most processor cycles. Allowing these common events to be recorded as exceptions in the DBSR while debug interrupts are disabled via MSR[DE] has little value.

If MSR[DE] = 1 at the time of the instruction complete debug exception, a debug interrupt occurs immediately and execution of the instruction causing the exception is suppressed, and CSRR0 (DSRR0 <E.ED>) is set to the address of the excepting instruction.

NOTE: Software Considerations

When an return from interrupt instruction or **mtmsr** is executed, the state of MSR[DE] at the time the instruction begins execution is used to determine if a debug event occurs. While the return from interrupt or **mtmsr** instruction modifies the MSR[DE] bit and the instruction complete debug event occurs after the execution of the instruction, the instruction complete debug event uses MSR[DE] at the beginning of instruction execution.

This behavior is useful since the debug interrupt handler will have DBCR0[ICMP] set and will execute a return from interrupt instruction that transitions MSR[DE] from 0 to 1 to single step an instruction. The instruction complete interrupt will not occur immediately after the return from interrupt, but instead after the first instruction after the return has been executed.

9.7.7 Interrupt Taken Debug Event

An interrupt taken debug condition (IRPT) occurs if DBCR0[IRPT] = 1 (that is interrupt taken debug events are enabled) and a base class interrupt (an interrupt that sets SRR0/1 (or GSRR0/1 <E.ED>)) occurs.

<Embedded.Hypervisor>:
When MSR[GS] = 0 and EPCR[DUVD] = 1, interrupt taken debug conditions are suppressed.

When an interrupt taken debug condition occurs, if DBCR0[IDM] and MSR[DE] are set and the interrupt taken debug exception is the highest priority exception, the debug event is said to occur and the corresponding DBSR[IRPT] bit is set to record the debug exception.

When an interrupt taken debug condition occurs, if DBCR0[IDM] = 1 and MSR[DE] = 0, a debug interrupt does not occur. On processors that support delayed debug interrupts, the corresponding DBSR[IRPT] bit is set to record the debug exception and DBSR[IDE] is set to record the imprecise debug event and a delayed debug exception is pending until MSR[DE] is later set.

If MSR[DE] = 1 at the time of the interrupt taken debug exception, a debug interrupt occurs immediately and CSRR0 (DSRR0 <E.ED>) is set to the address of the first instruction of the base class interrupt that occurred.

If an interrupt taken debug event causes a delayed debug exception, a debug interrupt does not occur, and the instruction completes execution (provided the instruction is not causing some other exception which generates an enabled interrupt).

Later, if the delayed debug exception has not been reset by clearing the corresponding DBSR[IRPT] bit and MSR[DE] is set, a delayed debug interrupt occurs. In this case, CSRR0 (DSRR0 <E.ED>) contains the address of the instruction after the one which enabled the debug interrupt by setting MSR[DE]. Software in the debug interrupt handler can observe DBSR[IDE] to determine how to interpret the value in CSRR0 (DSRR0 <E.ED>).\

Note that only base class interrupts set the interrupt taken debug condition. Other classes of interrupts (critical, debug <E.ED>, or machine check) do not cause interrupt taken debug conditions.

9.7.8 Critical Interrupt Taken Debug Event <E.ED>

A critical interrupt taken debug condition (IRPT) occurs if DBCR0[CIRPT] = 1 (that is critical interrupt taken debug events are enabled) and a critical class interrupt (an interrupt that sets CSRR0/1) occurs.

<Embedded.Hypervisor>:
When MSR[GS] = 0 and EPCR[DUVD] = 1, critical interrupt taken debug conditions are suppressed.

When a critical interrupt taken debug condition occurs, if DBCR0[IDM] and MSR[DE] are set and the critical interrupt taken debug exception is the highest priority exception, the debug event is said to occur and the corresponding DBSR[CIRPT] bit is set to record the debug exception.

When a critical interrupt taken debug condition occurs, if DBCR0[IDM] = 1 and MSR[DE] = 0, a debug interrupt does not occur. On processors that support delayed debug interrupts, the corresponding DBSR[CIRPT] bit is set to record the debug exception and DBSR[IDE] is set to record the imprecise debug event and a delayed debug exception is pending until MSR[DE] is later set.

If MSR[DE] = 1 at the time of the critical interrupt taken debug exception, a debug interrupt occurs immediately and DSRR0 is set to the address of the first instruction of the critical class interrupt that occurred.

If a critical interrupt taken debug event causes a delayed debug exception, a debug interrupt does not occur, and the instruction completes execution (provided the instruction is not causing some other exception which generates an enabled interrupt).

Later, if the delayed debug exception has not been reset by clearing the corresponding DBSR[CIRPT] bit and MSR[DE] is set, a delayed debug interrupt occurs. In this case, DSRR0 contains the address of the instruction after the one which enabled the debug interrupt by setting MSR[DE]. Software in the debug interrupt handler can observe DBSR[IDE] to determine how to interpret the value in DSRR0.

Note that only critical class interrupts set the critical interrupt taken debug condition. Other classes of interrupts (base, debug or machine check) do not cause critical interrupt taken debug conditions.

9.7.9 Return Debug Event

A return debug condition (RET) occurs if DBCR0[RET] = 1 (that is return debug events are enabled) and an attempt is made to execute an **rfi** (or **rfgi** <E.HV>) instruction.

<Embedded.Hypervisor>:
When MSR[GS] = 0 and EPCR[DUVD] = 1, return debug conditions are suppressed.

When a return debug condition occurs, if DBCR0[IDM] and MSR[DE] are set and the return debug exception is the highest priority exception, the debug event is said to occur and the corresponding DBSR[RET] bit is set to record the debug exception.

When a return debug condition occurs, if DBCR0[IDM] = 1 and MSR[DE] = 0, a debug interrupt does not occur. On processors that support delayed debug interrupts, the corresponding DBSR[RET] bit is set to record the debug exception and DBSR[IDE] is set to record the imprecise debug event and a delayed debug exception is pending until MSR[DE] is later set.

If MSR[DE] = 1 at the time of the return debug exception, a debug interrupt occurs immediately and execution of the instruction causing the exception is suppressed, and CSRR0 (DSRR0 <E.ED>) is set to the address of the **rfi** (or **rfgi** <E.HV>).

If a return debug event causes a delayed debug exception, a debug interrupt does not occur, and the instruction completes execution (provided the instruction is not causing some other exception which generates an enabled interrupt).

Later, if the delayed debug exception has not been reset by clearing the corresponding DBSR[RET] bit and MSR[DE] is set, a delayed debug interrupt occurs. In this case, CSRR0 (DSRR0 <E.ED>) contains the address of the instruction after the one which enabled the debug interrupt by setting MSR[DE]. Software in the debug interrupt handler can observe DBSR[IDE] to determine how to interpret the value in CSRR0 (DSRR0 <E.ED>) .

9.7.10 Critical Interrupt Return Debug Event <E.ED>

A critical interrupt return debug condition (CRET) occurs if DBCR0[CRET] = 1 (that is critical interrupt return debug events are enabled) and an attempt is made to execute an **rfci** instruction.

<Embedded.Hypervisor>:
When MSR[GS] = 0 and EPCR[DUVD] = 1, critical interrupt return debug conditions are suppressed.

When a critical interrupt return debug condition occurs, if DBCR0[IDM] and MSR[DE] are set and the critical interrupt return debug exception is the highest priority exception, the debug event is said to occur and the corresponding DBSR[CRET] bit is set to record the debug exception.

When a critical interrupt return debug condition occurs, if DBCR0[IDM] = 1 and MSR[DE] = 0, a debug interrupt does not occur. On processors that support delayed debug interrupts, the corresponding DBSR[CRET] bit is set to record the debug exception and DBSR[IDE] is set to record the imprecise debug event and a delayed debug exception is pending until MSR[DE] is later set.

If MSR[DE] = 1 at the time of the critical interrupt return debug exception, a debug interrupt occurs immediately and execution of the instruction causing the exception is suppressed, and DSRR0 is set to the address of the **rfci**.

If a critical interrupt return debug event causes a delayed debug exception, a debug interrupt does not occur, and the instruction completes execution (provided the instruction is not causing some other exception which generates an enabled interrupt).

Later, if the delayed debug exception has not been reset by clearing the corresponding DBSR[CRET] bit and MSR[DE] is set, a delayed debug interrupt occurs. In this case, DSRR0 contains the address of the instruction after the one which enabled the debug interrupt by setting MSR[DE]. Software in the debug interrupt handler can observe DBSR[IDE] to determine how to interpret the value in DSRR0.

9.7.11 Unconditional Debug Event (UDE)

An unconditional debug condition (UDE) occurs when the UDE signal is activated by the debug mechanism. The exact definition of the UDE signal and how it is activated is implementation-dependent. The unconditional debug condition is the only debug event which does not have a corresponding enable bit for the event in DBCR0 (hence the name of the event). The unconditional debug condition can occur regardless of the setting of MSR[DE].

When an unconditional debug condition occurs, if DBCR0[IDM] and MSR[DE] are set and the unconditional debug exception is the highest priority exception, the debug event is said to occur and the corresponding DBSR[UDE] bit is set to record the debug exception and DBSR[IDE] is set to record the imprecise debug event. Note that DBSR[UDE] is always set regardless of MSR[DE] because UDE is an asynchronous interrupt and is not associated with an instruction in the instruction stream.

When an unconditional debug condition occurs, if DBCR0[IDM] = 1 and MSR[DE] = 0, a debug interrupt does not occur. On processors that support delayed debug interrupts, the corresponding DBSR[UDE] bit is set to record the debug exception and DBSR[IDE] is set to record the imprecise debug event and a delayed debug exception is pending until MSR[DE] is later set.

If MSR[DE] = 1 at the time of the unconditional debug exception, a debug interrupt occurs immediately and execution of the instruction causing the exception is suppressed, and CSRR0 (DSRR0 <E.ED>) is set to the address of the instruction that would have executed next had the interrupt not occurred.

If an unconditional debug event causes a delayed debug exception, a debug interrupt does not occur, and the instruction completes execution (provided the instruction is not causing some other exception which generates an enabled interrupt).

Later, if the delayed debug exception has not been reset by clearing the corresponding DBSR[UDE] bit and MSR[DE] is set, a delayed debug interrupt occurs. In this case, CSRR0 (DSRR0 <E.ED>) contains the address of the instruction after the one which enabled the debug interrupt by setting MSR[DE].

<Embedded.Hypervisor>:
When MSR[GS] = 0 and EPCR[DUVD] = 1, it is implementation-dependent whether unconditional debug conditions are suppressed.

9.7.12 DAC and IAC Linking

Some processors support the ability to “link” an instruction address compare condition and a data address compare condition such that when both occur, a data address compare event occurs.

If DBCR2[DACLINK1] is set, IAC1 debug events do not occur, and DAC1 debug events occur only if a data address compare 1 condition and an instruction address compare 1 condition occur on the same instruction. Both IAC1 and DAC1 address compare modes must be set to exact address compare mode, or the results are boundedly undefined. Similarly if DBCR2[DACLINK2] is set, IAC2 debug events do not occur, and DAC2 debug events occur only if a data address compare 1 condition and an instruction address compare 1 condition occur on the same instruction. Both IAC2 and DAC2 address compare modes must be set to exact address compare mode, or the results are boundedly undefined.

Chapter 10
Performance Monitor <E.PM>

This chapter provides an overview of the performance monitor facility. Specific details about instructions and registers are provided in Chapter 3, “Register Model,” Chapter 4, “Instruction Model,” and Chapter 7, “Interrupts and Exceptions.”

10.1 Performance Monitor Overview

The performance monitor facility defines resources for managing the counting of events in the processor for monitoring performance and processor activity. The method and programming model for performance monitoring is defined by EIS. Some features, most importantly the events which can be monitored, are implementation-specific. Users wishing to make use of the performance monitor facility should consult the processor core reference manual.

NOTE: Software Considerations

> The performance monitor defined by EIS provides a way to characterize processor behavior by tracking processor-related activities. Most integrated devices also define a similar, but separate performance monitor facility that can be used to characterize the integrated device as a whole, using memory-mapped registers. Consult the device reference manual for details.

10.2 Performance Monitor Features

The performance monitor facility provides the ability to count implementation-specific predefined events and processor clocks associated with particular operations. For example, cache misses, mispredicted branches, or the number of cycles an execution unit stalls may be countable events in a particular processor implementation. The count of such events can be used to trigger the performance monitor interrupt.

The performance monitor facility can be used to do the following:

- Improve system performance by monitoring software execution and then recoding algorithms for more efficiency. For example, memory hierarchy behavior can be monitored and analyzed to optimize task scheduling or data distribution algorithms.
- Characterize processors in environments not easily characterized by benchmarking.
- Help system developers bring up and debug their systems.

The performance monitor facility defines the following resources:

- Performance monitor registers (PMRs) to manage which events are counted and in what conditions they are counted.

- The performance monitor mark bit, MSR[PMM], controls which programs are monitored.
- The move to/from performance monitor registers instructions, **mtpmr** and **mfpmr**.
- The performance monitor interrupt and associated interrupt vector offset register (IVOR35).
- Some processor implementations contain other resources, such as signals from other parts of the integrated device. Those signals are implementation specific and not defined by EIS.

10.3 Performance Monitor Programming Model

10.3.1 Performance Monitor Registers Overview

The registers used for configuring and counting events are defined as performance monitor registers (PMRs). PMRs are similar to SPRs and are accessed using **mtpmr** and **mfpmr** instructions. PMRs are described in Section 3.15, “Performance Monitor Registers (PMRs) <E.PM>.” There is no relationship between SPRs and PMRs.

The SPR IVOR35 is used for indicating the address of the performance monitor interrupt vector. IVOR35 is described in Section 3.8.5, “Interrupt Vector Offset Registers (IVORs).”

The PMRs are used to collect and control performance data collection.

This figure shows the performance monitor register model.

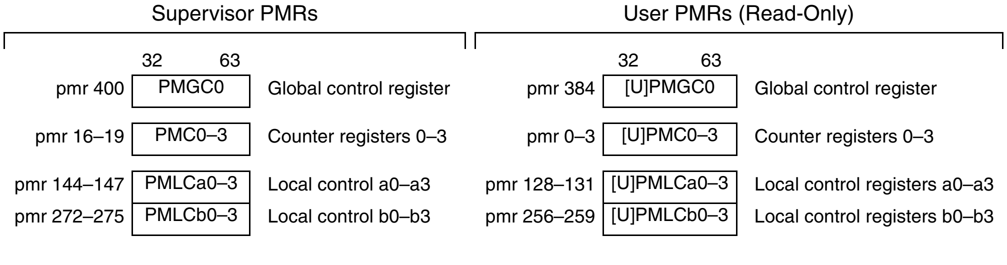


Figure 10-1. Performance Monitor Registers

PMRs provide various controls and access to collected data. PMR numbers allow for different access rights via user and supervisor level access to these registers. User level registers, characterized by PMRs prefaced with a “U” (such as UPMC0), are only allowed read-only access to certain PMRs. The equivalent supervisor level PMR is read/write (such as PMC0). UPMC0 and PMC0 both refer to the same physical register, but the different PMR numbers allow different access for different privilege levels. When the embedded.hypervisor category is implemented, access to PMRs is restricted in guest state if MSRP[PMMP] is set. See Section 3.6.2, “Machine State Register Protect Register (MSRP) <E.HV>.”

10.3.2 Using PMRs to Control and Access Performance Monitoring

PMRs are used to control and access performance monitoring. They are categorized as follows:

- Counter registers ([U]PMC0–[U]PMC15). These registers are 32-bit counters used to count software-selectable events. Reference events are those that should be applicable to most microprocessor microarchitectures and be of general value.

- Global control register ([U]PMGC0). Controls global settings of the performance monitor facility and affects all counters. MSR[PMM] is defined to enable and disable counting.
- Local control registers [U]PMLC*a*0–[U]PMLC*a*15 and [U]PMLC*b*0–[U]PMLC*b*15. Control settings that apply only to a particular counter. Each set of local control registers (PMLCa*n* and PMLCb*n*) contains controls that apply to the associated same numbered counter register (for example PMLCa0 and PMLCb0 contain controls for PMC0 while PMLCa1 and PMLCb1 contain controls for PMC1).

NOTE: Software Considerations

> The counter registers, global controls, and local controls have alias names that cause the assembler to use different PMR numbers. The names PMC0–PMC15, PMGC0, PMLCa0–PMLCa15, and PMLCb0–PMLCb15 cause the assembler to use the supervisor level PMR number.
>
> Names starting with U (UPMC0–UPMC15, UPMGC0, UPMLCa0–UPMLCa15, and UPMLCb0–UPMLCb15) cause the assembler to use the user-level PMR number.

A given implementation may implement fewer counter PMRs (and their associated control PMRs) than are architected. Unimplemented PMRs behave the same as unarchitected PMRs.

Software uses the global and local controls to select which events are counted in the counter registers, when such events should be counted, and what action should be taken when a counter overflows. Software can use the collected information to determine performance attributes of a given segment of code, a process, or the entire software system. The **mfpmr** and **mtpmr** instructions are described in Section 4.6.1.14, “Performance Monitor Instructions.”

Because counters are defined as 32-bit registers, the counting of some events can overflow. The performance monitor interrupt can be programmed to occur in the event of an overflow. The performance monitor interrupt model is described in detail in Section 10.3.3, “Performance Monitor Interrupt Model.”

As with SPRs, bit 5 of the PMR register numbers indicates whether a register is user- or supervisor-accessible. Attempting to read or write supervisor-level registers while in user-mode causes a privilege exception.

The user-level PMRs are read-only and are accessed with the **mfpmr** instruction. Attempting to write user-level registers in either supervisor or user mode causes an illegal instruction exception.

PMRs are fully described in Section 3.15, “Performance Monitor Registers (PMRs) <E.PM>.”

10.3.3 Performance Monitor Interrupt Model

A performance monitor interrupt is triggered by an implementation-defined exception enabled in the PMRs. All implementations contain an overflow condition which triggers the performance monitor exception when counter overflow detection is enabled and a counter overflows. More specifically, for each counter register *n*:

- The counter’s overflow condition is enabled; PMLCa*n*[CE] is set.

- The counter indicates an overflow; PMC*n*[OV] is set.
- The enabling conditions for the interrupt are set; PMGC0[PMIE] is set and MSR[EE] = 1 (or MSR[GS] = 1 <E.HV>).

If PMGC0[PMIE] is set and an overflow condition exists and PMGC0[FCECE] is set, an enabled condition or event also triggers all performance monitor counters to freeze. If the performance monitor interrupt is masked, counters are frozen regardless of whether the interrupt can be taken based on the masking conditions.

The performance monitor exception condition causes a performance monitor interrupt if the exception is the highest priority exception.

The performance monitor exception is level sensitive and the exception condition may cease to exist if any of the required conditions fail to be met. Thus it is possible for a counter to overflow and continue counting events until PMC*n*[OV] becomes 0 without taking a performance monitor interrupt if the interrupt is masked during the overflow condition. To avoid this, software should program the counters to freeze if an overflow condition is detected.

The performance monitor interrupt is fully described in Section 7.8.20, “Performance Monitor Interrupt <E.PM>.”

NOTE: Software Considerations

> When taking a performance monitor interrupt, software should clear the overflow condition by reading the counter register and setting the counter register to a non-overflow value since the normal return from the interrupt sets MSR[EE] back to 1.

10.4 Performance Monitor Use Case

This section describes a performance monitor facility use case. The architecture presents only programming model visible features in conjunction with architecturally defined behavioral features. Much of the selection of events is, by necessity, implementation-dependent and is not described as part of the architecture; however, this document describes typically implemented features.

The performance monitor facility use case provides the ability to monitor and count predefined events such as processor clocks, misses in the instruction cache or data cache, types of instructions decoded, or mispredicted branches. The count of such events can be used to trigger the performance monitor exception. While most of the specific events are not architected, the mechanism of controlling data collection is.

10.4.1 Event Counting

Event counting can be configured in several different ways. This section describes configurability and specific unconditional counting modes.

10.4.2 Processor Context Configurability

When conditions in the processor state match a software-specified condition, counting can be enabled. Because a software task scheduler may switch a processor’s execution among multiple processes and because statistics on only a particular process may be of interest, a facility is provided to mark a process. The performance monitor mark bit, MSR[PMM], is used for this purpose. System software may set this bit when a marked process is running. This enables statistics to be gathered only during the execution of the marked process. The states of MSR[PR] and MSR[PMM] together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified by the PMLCa*n*[FCS], PMLCa*n*[FCU], PMLCa*n*[FCM1] and PMLCa*n*[FCM0] fields in PMLCa*n* (the state for which monitoring is enabled), counting is enabled for PMC*n*.

Each event, on an implementation basis, may count regardless of the value of MSR[PMM]. The counting behavior of each event is documented in the implementation reference manual.

10.4.2.1 Enabling the Monitor of Processor States

This table shows the PMLCa*n*[FCS], PMLCa*n*[FCU], PMLCa*n*[FCM1], PMLCa*n*[FCM0], PMLCa*n*[FCGS1] <E.HV>, and PMLCa*n*[FCGS0] <E.HV> settings necessary to enable monitoring of each processor state.

Table 10-1. Processor States and PMLCa*n* Bit Settings

FCS	FCU	FCM1	FCM0	FCGS1 <E.HV>	FCGS0 <E.HV>	Processor State
0	0	0	1	0	0	Marked
0	0	1	0	0	0	Not marked
0	1	0	0	0	0	Supervisor
1	0	0	0	0	0	User
0	1	0	0	0	1	Guest Supervisor
0	1	0	0	1	0	Hypervisor
0	1	0	1	0	0	Marked and supervisor
1	0	0	1	0	0	Marked and user
0	1	1	0	0	0	Not marked and supervisor
1	0	1	0	0	0	Not mark and user
0	0	0	0	0	0	All
X	X	1	1	1	1	None
1	1	X	X	X	X	None

10.4.2.2 Specifying Unconditional Counting Modes

Two unconditional counting modes may be specified:

- Counting is unconditionally enabled regardless of the states of MSR[PMM] and MSR[PR]. This is done by clearing PMLCa*n*[FCS], PMLCa*n*[FCU], PMLCa*n*[FCM1], and PMLCa*n*[FCM0] for each counter control.
- Counting is unconditionally disabled regardless of the states of MSR[PMM] and MSR[PR]. This is done by setting PMGC0[FAC] or by setting PMLCa*n*[FC] for each counter control.

Alternatively, this can be done by setting PMLCa*n*[FCM1] and PMLCa*n*[FCM0] for each counter control or by setting PMLCa*n*[FCS] and PMLCa*n*[FCU] for each counter control.

NOTE: Software Considerations

Events may be counted in a fuzzy manner. That is, events may not be counted precisely due to the nature of an implementation. Users of the performance monitor facility should be aware that an event may be counted even if it was precisely filtered, though it should not have been. In general such discrepancies are statistically unimportant and users should not assume that counts are explicitly accurate.

10.4.3 Event Selection

Events to count are determined by placing an implementation-defined event value into PMLCa*n*[EVENT]. Which events may be programmed into which counter are implementation specific and should be defined in the implementation document. In general, most events may be programmed into any of the implementation available counters. Programming a counter with an event that is not supported for that counter gives boundedly undefined results.

NOTE: Software Considerations

Event name and event numbers differ greatly across implementations and software should not expect that events and event names to be consistent.

10.4.4 Chaining Counters

An implementation may contain events that are used to chain counters together to provide a larger range of event counts. This is accomplished by programming the desired event into one counter and programming another counter with an event that occurs when the first counter transitions from 1 to 0 in the most significant bit.

The counter chaining feature can be used to decrease the processing pollution caused by performance monitor interrupts, (things like cache contamination, and pipeline effects), by allowing a higher event count than is possible with a single counter. Chaining two counters together effectively adds 32 bits to a counter register where the first counter's carry-out event acts like a carry-out feeding the second counter. By defining the event of interest to be another PMC's overflow generation, the chained counter increments each time the first counter rolls over to zero. Multiple counters may be chained together.

Because the entire chained value cannot be read in a single instruction, an overflow may occur between counter reads, producing an inaccurate value. A sequence like the following is necessary to read the complete chained value when it spans multiple counters and the counters are not frozen. The example shown is for a two-counter case.

```
loop:
    mfpmr   Rx,pmctr1   #load from upper counter
    mfpmr   Ry,pmctr0   #load from lower counter
    mfpmr   Rz,pmctr1   #load from upper counter
    cmp     cr0,0,Rz,Rx #see if ‘old’ = ‘new’
    bc      4,2,loop    #loop if carry occurred between reads
```

The comparison and loop are necessary to ensure that a consistent set of values has been obtained. If the counters are frozen, the above sequence is not necessary.

10.4.5 Thresholds

Thresholds are values that must be exceeded for an event to be counted. Threshold values are programmed in PMLCb*n*[THRESHOLD]. Events which may be thresholded and the units of each event that may be thresholded are implementation-dependent. Programming a threshold value for an event that is not defined to use a threshold gives boundedly undefined results.

Threshold event measurement enables the counting of duration and usage events. Assume an example event, data cache load miss cycles, requires a threshold value. A data cache load miss cycles event is counted only when the number of cycles spent recovering from the miss is greater than the threshold. If the event is counted on two counters and each counter has an individual threshold, one execution of a performance monitor program can sample two different threshold values. Measuring code performance with multiple concurrent thresholds expedites code profiling significantly.

Appendix A
Revision History

This appendix provides a list of the major differences between the *EREF 2.0: A Programmer's Reference Manual for Freescale Power Architecture® Processors*, Revision 0. Because this is the initial version, there are no differences.

Appendix B
Instruction Set Listings

This appendix lists the instructions by both mnemonic and opcode. The tables in the chapter are organized as follows:

- Section B.1, "Instructions Sorted by Mnemonic (Decimal)"
- Section B.2, "Instructions Sorted by Opcodes (Decimal)"
- Section B.3, "Instructions Sorted by Mnemonic (Binary)"
- Section B.4, "Instructions Sorted by Opcode (Binary)"

Note that this appendix does not include instructions defined by the VLE extension. These instructions are listed in the VLE PEM.

B.1 Instructions Sorted by Mnemonic (Decimal)

The following table lists instructions in alphabetical order by mnemonic, showing decimal values of the primary opcode (0–5) and binary values of the secondary opcode (21–31). This list also includes simplified mnemonics and their equivalents using standard mnemonics.

Table B-1. Instructions Sorted by Mnemonic (Decimal)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
add	31	rD	rA	rB	266	0	X	
add.	31	rD	rA	rB	266	1	X	
addc	31	rD	rA	rB	10	0	X	
addc.	31	rD	rA	rB	10	1	X	
addco	31	rD	rA	rB	522	0	X	
addco.	31	rD	rA	rB	522	1	X	
adde	31	rD	rA	rB	138	0	X	
adde.	31	rD	rA	rB	138	1	X	
addeo	31	rD	rA	rB	650	0	X	
addeo.	31	rD	rA	rB	650	1	X	
addi	14	rD	rA	SIMM			D	
addic	12	rD	rA	SIMM			D	

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8	9	10	11 12 13 14 15	16 17 18	19 20	21 22 23 24 25 26 27 28 29	30	31	Form	Category
addic.	13	rD			rA	SIMM					D	
addis	15	rD			rA	SIMM					D	
addme	31	rD			rA	///		234		0	X	
addme.	31	rD			rA	///		234		1	X	
addmeo	31	rD			rA	///		746		0	X	
addmeo.	31	rD			rA	///		746		1	X	
addo	31	rD			rA	rB		778		0	X	
addo.	31	rD			rA	rB		778		1	X	
addze	31	rD			rA	///		202		0	X	
addze.	31	rD			rA	///		202		1	X	
addzeo	31	rD			rA	///		714		0	X	
addzeo.	31	rD			rA	///		714		1	X	
and	31	rS			rA	rB		28		0	X	
and.	31	rS			rA	rB		28		1	X	
andc	31	rS			rA	rB		60		0	X	
andc.	31	rS			rA	rB		60		1	X	
andi.	28	rS			rA	UIMM					D	
andis.	29	rS			rA	UIMM					D	
b	18	LI							0	0	I	
ba	18	LI							1	0	I	
bc	16	BO			BI	BD			0	0	B	
bca	16	BO			BI	BD			1	0	B	
bcctr	19	BO			BI	///	BH	528		0	XL	
bcctrl	19	BO			BI	///	BH	528		1	XL	
bcl	16	BO			BI	BD			0	1	B	
bcla	16	BO			BI	BD			1	1	B	
bclr	19	BO			BI	///	BH	16		0	XL	
bclrl	19	BO			BI	///	BH	16		1	XL	
bl	18	LI							0	1	I	
bla	18	LI							1	1	I	
bpermd	31	rS			rA	rB		252		/	X	**64**
brinc	4	rD			rA	rB		527			EVX	**SP**
cmp	31	crD	/	L	rA	rB		0		/	X	

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
cmpb	31	rS	rA	rB	508	/	X	
cmpi	11	crD / L	rA	SIMM			D	
cmpl	31	/ L	rA	rB	/// 32	/	X	
cmpli	10	crD / L	rA	UIMM			D	
cntlzd	31	rS	rA	///	58	0	**X**	**64**
cntlzd.	31	rS	rA	///	58	1	**X**	**64**
cntlzw	31	rS	rA	///	26	0	X	
cntlzw.	31	rS	rA	///	26	1	X	
crand	19	crbD	crbA	crbB	257	/	XL	
crandc	19	crbD	crbA	crbB	129	/	XL	
creqv	19	crbD	crbA	crbB	289	/	XL	
crnand	19	crbD	crbA	crbB	225	/	XL	
crnor	19	crbD	crbA	crbB	33	/	XL	
cror	19	crbD	crbA	crbB	449	/	XL	
crorc	19	crbD	crbA	crbB	417	/	XL	
crxor	19	crbD	crbA	crbB	193	/	XL	
dcba	31	/// 0	rA	rB	758	/	X	
dcbal	31	/// 0	rA	rB	758	/	X	**DEO**
dcbf	31	///	rA	rB	86	/	X	
dcbfep	31	///	rA	rB	127	/	X	**E.PD**
dcbi	31	///	rA	rB	470	/	X	**Embedded**
dcblc	31	CT	rA	rB	390	/	X	**E.CL**
dcbst	31	///	rA	rB	54	/	X	
dcbstep	31	///	rA	rB	63	/	X	**E.PD**
dcbt	31	TH	rA	rB	278	/	X	
dcbtep	31	TH	rA	rB	319	/	X	**E.PD**
dcbtls	31	CT	rA	rB	166	/	X	**E.CL**
dcbtst	31	TH	rA	rB	246	/	X	
dcbtstep	31	TH	rA	rB	255	/	X	**E.PD**
dcbtstls	31	CT	rA	rB	134	/	X	**E.CL**
dcbz	31	/// 0	rA	rB	1014	/	X	
dcbzep	31	/// 0	rA	rB	1023	/	X	**E.PD**
dcbzl	31	/// 1	rA	rB	1014	/	X	**DEO**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

<table>
<tr><th>Mnemonic</th><th>0 1 2 3 4 5</th><th>6</th><th>7 8</th><th>9</th><th>10</th><th>11</th><th>12 13 14 15</th><th>16 17 18 19</th><th>20</th><th>21 22 23 24 25 26 27 28 29 30</th><th>31</th><th>Form</th><th>Category</th></tr>
<tr><td>dcbzlep</td><td>31</td><td colspan="3">///</td><td>1</td><td colspan="2">rA</td><td colspan="2">rB</td><td>1023</td><td>/</td><td>X</td><td>DEO, E.PD</td></tr>
<tr><td>divd</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>489</td><td>0</td><td>X</td><td>64</td></tr>
<tr><td>divd.</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>489</td><td>1</td><td>X</td><td>64</td></tr>
<tr><td>divdo</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>1001</td><td>0</td><td>X</td><td>64</td></tr>
<tr><td>divdo.</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>1001</td><td>1</td><td>X</td><td>64</td></tr>
<tr><td>divdu</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>457</td><td>0</td><td>X</td><td>64</td></tr>
<tr><td>divdu.</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>457</td><td>1</td><td>X</td><td>64</td></tr>
<tr><td>divduo</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>969</td><td>0</td><td>X</td><td>64</td></tr>
<tr><td>divduo.</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>969</td><td>1</td><td>X</td><td>64</td></tr>
<tr><td>divw</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>491</td><td>0</td><td>X</td><td></td></tr>
<tr><td>divw.</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>491</td><td>1</td><td>X</td><td></td></tr>
<tr><td>divwo</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>1003</td><td>0</td><td>X</td><td></td></tr>
<tr><td>divwo.</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>1003</td><td>1</td><td>X</td><td></td></tr>
<tr><td>divwu</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>459</td><td>0</td><td>X</td><td></td></tr>
<tr><td>divwu.</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>459</td><td>1</td><td>X</td><td></td></tr>
<tr><td>divwuo</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>971</td><td>0</td><td>X</td><td></td></tr>
<tr><td>divwuo.</td><td>31</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td>971</td><td>1</td><td>X</td><td></td></tr>
<tr><td>dnh</td><td>19</td><td colspan="4">DUI</td><td colspan="2">DCTL</td><td>0</td><td>0</td><td>198</td><td>/</td><td>X</td><td>E.ED</td></tr>
<tr><td>dni</td><td>31</td><td colspan="4">DUI</td><td colspan="2">DCTL</td><td>0</td><td>0</td><td>97</td><td>1</td><td>X</td><td>E.ED</td></tr>
<tr><td>dsn</td><td>31</td><td colspan="4">///</td><td colspan="2">rA</td><td colspan="2">rB</td><td>473</td><td>/</td><td>X</td><td>DS</td></tr>
<tr><td>dss</td><td>31</td><td>0</td><td>//</td><td colspan="2">STRM</td><td colspan="2">///</td><td colspan="2">///</td><td>822</td><td>/</td><td>X</td><td>V</td></tr>
<tr><td>dssall</td><td>31</td><td>1</td><td>//</td><td colspan="2">STRM</td><td colspan="2">///</td><td colspan="2">///</td><td>822</td><td>/</td><td>X</td><td>V</td></tr>
<tr><td>dst</td><td>31</td><td>0</td><td>//</td><td colspan="2">STRM</td><td colspan="2">rA</td><td colspan="2">rB</td><td>342</td><td>/</td><td>X</td><td>V</td></tr>
<tr><td>dstst</td><td>31</td><td>0</td><td>//</td><td colspan="2">STRM</td><td colspan="2">rA</td><td colspan="2">rB</td><td>374</td><td>/</td><td>X</td><td>V</td></tr>
<tr><td>dststt</td><td>31</td><td>1</td><td>//</td><td colspan="2">STRM</td><td colspan="2">rA</td><td colspan="2">rB</td><td>374</td><td>/</td><td>X</td><td>V</td></tr>
<tr><td>dstt</td><td>31</td><td>1</td><td>//</td><td colspan="2">STRM</td><td colspan="2">rA</td><td colspan="2">rB</td><td>342</td><td>/</td><td>X</td><td>V</td></tr>
<tr><td>efdabs</td><td>4</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">///</td><td colspan="2">740</td><td>EVX</td><td>SP.FD</td></tr>
<tr><td>efdadd</td><td>4</td><td colspan="4">rD</td><td colspan="2">rA</td><td colspan="2">rB</td><td colspan="2">736</td><td>EVX</td><td>SP.FD</td></tr>
<tr><td>efdcfs</td><td>4</td><td colspan="4">rD</td><td>0</td><td>0 0 0 0</td><td colspan="2">rB</td><td colspan="2">751</td><td>EVX</td><td>SP.FD</td></tr>
<tr><td>efdcfsf</td><td>4</td><td colspan="4">rD</td><td colspan="2">///</td><td colspan="2">rB</td><td colspan="2">755</td><td>EVX</td><td>SP.FD</td></tr>
<tr><td>efdcfsi</td><td>4</td><td colspan="4">rD</td><td colspan="2">///</td><td colspan="2">rB</td><td colspan="2">753</td><td>EVX</td><td>SP.FD</td></tr>
<tr><td>efdcfuf</td><td>4</td><td colspan="4">rD</td><td colspan="2">///</td><td colspan="2">rB</td><td colspan="2">754</td><td>EVX</td><td>SP.FD</td></tr>
<tr><td>efdcfui</td><td>4</td><td colspan="4">rD</td><td colspan="2">///</td><td colspan="2">rB</td><td colspan="2">752</td><td>EVX</td><td>SP.FD</td></tr>
</table>

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8	9 10	11	12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Category
efdcmpeq	4	crD	/ /	rA		rB	750	EVX	**SP.FD**
efdcmpgt	4	crD	/ /	rA		rB	748	EVX	**SP.FD**
efdcmplt	4	crD	/ /	rA		rB	749	EVX	**SP.FD**
efdctsf	4	rD		///		rB	759	EVX	**SP.FD**
efdctsi	4	rD		///		rB	757	EVX	**SP.FD**
efdctsiz	4	rD		///		rB	762	EVX	**SP.FD**
efdctuf	4	rD		///		rB	758	EVX	**SP.FD**
efdctui	4	rD		///		rB	756	EVX	**SP.FD**
efdctuiz	4	rD		///		rB	760	EVX	**SP.FD**
efddiv	4	rD		rA		rB	745	EVX	**SP.FD**
efdmul	4	rD		rA		rB	744	EVX	**SP.FD**
efdnabs	4	rD		rA		///	741	EVX	**SP.FD**
efdneg	4	rD		rA		///	742	EVX	**SP.FD**
efdsub	4	rD		rA		rB	737	EVX	**SP.FD**
efdtsteq	4	crD	/ /	rA		rB	766	EVX	**SP.FD**
efdtstgt	4	crD	/ /	rA		rB	764	EVX	**SP.FD**
efdtstlt	4	crD	/ /	rA		rB	765	EVX	**SP.FD**
efsabs	4	rD		rA		///	708	EVX	**SP.FS**
efsadd	4	rD		rA		rB	704	EVX	**SP.FS**
efscfd	4	rD		0	0	rB	719	EVX	**SP.FS**
efscfsf	4	rD		///		rB	723	EVX	**SP.FS**
efscfsi	4	rD		///		rB	721	EVX	**SP.FS**
efscfuf	4	rD		///		rB	722	EVX	**SP.FS**
efscfui	4	rD		///		rB	720	EVX	**SP.FS**
efscmpeq	4	crD	/ /	rA		rB	718	EVX	**SP.FS**
efscmpgt	4	crD	/ /	rA		rB	716	EVX	**SP.FS**
efscmplt	4	crD	/ /	rA		rB	717	EVX	**SP.FS**
efsctsf	4	rD		///		rB	727	EVX	**SP.FS**
efsctsi	4	rD		///		rB	725	EVX	**SP.FS**
efsctsiz	4	rD		///		rB	730	EVX	**SP.FS**
efsctuf	4	rD		///		rB	726	EVX	**SP.FS**
efsctui	4	rD		///		rB	724	EVX	**SP.FS**
efsctuiz	4	rD		///		rB	728	EVX	**SP.FS**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
efsdiv	4	rD		rA	rB	713		EVX	**SP.FS**
efsmul	4	rD		rA	rB	712		EVX	**SP.FS**
efsnabs	4	rD		rA	///	709		EVX	**SP.FS**
efsneg	4	rD		rA	///	710		EVX	**SP.FS**
efssub	4	rD		rA	rB	705		EVX	**SP.FS**
efststeq	4	crD	/ /	rA	rB	734		EVX	**SP.FS**
efststgt	4	crD	/ /	rA	rB	732		EVX	**SP.FS**
efststlt	4	crD	/ /	rA	rB	733		EVX	**SP.FS**
ehpriv	31	OC				270	/	XL	**E.HV**
eqv	31	rD		rA	rB	568		X	
eqv.	31	rD		rA	rB	569		X	
evabs	4	rD		rA	///	520		EVX	**SP**
evaddiw	4	rD		UIMM	rB	514		EVX	**SP**
evaddsmiaaw	4	rD		rA	///	1225		EVX	**SP**
evaddssiaaw	4	rD		rA	///	1217		EVX	**SP**
evaddumiaaw	4	rD		rA	///	1224		EVX	**SP**
evaddusiaaw	4	rD		rA	///	1216		EVX	**SP**
evaddw	4	rD		rA	rB	512		EVX	**SP**
evand	4	rD		rA	rB	529		EVX	**SP**
evandc	4	rD		rA	rB	530		EVX	**SP**
evcmpeq	4	crD	/ /	rA	rB	564		EVX	**SP**
evcmpgts	4	crD	/ /	rA	rB	561		EVX	**SP**
evcmpgtu	4	crD	/ /	rA	rB	560		EVX	**SP**
evcmplts	4	crD	/ /	rA	rB	563		EVX	**SP**
evcmpltu	4	crD	/ /	rA	rB	562		EVX	**SP**
evcntlsw	4	rD		rA	///	526		EVX	**SP**
evcntlzw	4	rD		rA	///	525		EVX	**SP**
evdivws	4	rD		rA	rB	1222		EVX	**SP**
evdivwu	4	rD		rA	rB	1223		EVX	**SP**
eveqv	4	rD		rA	rB	537		EVX	**SP**
evextsb	4	rD		rA	///	522		EVX	**SP**
evextsh	4	rD		rA	///	523		EVX	**SP**
evfsabs	4	rD		rA	///	644		EVX	**SP.FV**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
evfsadd	4	rD		rA	rB	640		EVX	**SP.FV**
evfscfsf	4	rD		///	rB	659		EVX	**SP.FV**
evfscfsi	4	rD		///	rB	657		EVX	**SP.FV**
evfscfuf	4	rD		///	rB	658		EVX	**SP.FV**
evfscfui	4	rD		///	rB	656		EVX	**SP.FV**
evfscmpeq	4	crD	/ /	rA	rB	654		EVX	**SP.FV**
evfscmpgt	4	crD	/ /	rA	rB	652		EVX	**SP.FV**
evfscmplt	4	crD	/ /	rA	rB	653		EVX	**SP.FV**
evfsctsf	4	rD		///	rB	663		EVX	**SP.FV**
evfsctsi	4	rD		///	rB	661		EVX	**SP.FV**
evfsctsiz	4	rD		///	rB	666		EVX	**SP.FV**
evfsctuf	4	rD		///	rB	662		EVX	**SP.FV**
evfsctui	4	rD		///	rB	660		EVX	**SP.FV**
evfsctuiz	4	rD		///	rB	664		EVX	**SP.FV**
evfsdiv	4	rD		rA	rB	649		EVX	**SP.FV**
evfsmul	4	rD		rA	rB	648		EVX	**SP.FV**
evfsnabs	4	rD		rA	///	645		EVX	**SP.FV**
evfsneg	4	rD		rA	///	646		EVX	**SP.FV**
evfssub	4	rD		rA	rB	641		EVX	**SP.FV**
evfststeq	4	crD	/ /	rA	rB	670		EVX	**SP.FV**
evfststgt	4	crD	/ /	rA	rB	668		EVX	**SP.FV**
evfststlt	4	crD	/ /	rA	rB	669		EVX	**SP.FV**
evldd	4	rD		rA	UIMM[1]	769		EVX	**SP**
evlddepx	31	rD		rA	rB	31	/	X	**E.PD, SP**
evlddx	4	rD		rA	rB	768		EVX	**SP**
evldh	4	rD		rA	UIMM[2]	773		EVX	**SP**
evldhx	4	rD		rA	rB	772		EVX	**SP**
evldw	4	rD		rA	UIMM[3]	771		EVX	**SP**
evldwx	4	rD		rA	rB	770		EVX	**SP**
evlhhesplat	4	rD		rA	UIMM[2]	388		EVX	**SP**
evlhhesplatx	4	rD		rA	rB	776		EVX	**SP**
evlhhossplat	4	rD		rA	UIMM[2]	391		EVX	**SP**
evlhhossplatx	4	rD		rA	rB	782		EVX	**SP**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Category
evlhhousplat	4	rD	rA	UIMM[2]	392	EVX	**SP**
evlhhousplatx	4	rD	rA	rB	780	EVX	**SP**
evlwhe	4	rD	rA	UIMM[2]	392	EVX	**SP**
evlwhex	4	rD	rA	rB	784	EVX	**SP**
evlwhos	4	rD	rA	UIMM[2]	395	EVX	**SP**
evlwhosx	4	rD	rA	rB	790	EVX	**SP**
evlwhou	4	rD	rA	UIMM[2]	394	EVX	**SP**
evlwhoux	4	rD	rA	rB	788	EVX	**SP**
evlwhsplat	4	rD	rA	UIMM[2]	398	EVX	**SP**
evlwhsplatx	4	rD	rA	rB	796	EVX	**SP**
evlwwsplat	4	rD	rA	UIMM[3]	1420	EVX	**SP**
evlwwsplatx	4	rD	rA	rB	1420	EVX	**SP**
evmergehi	4	rD	rA	rB	556	EVX	**SP**
evmergehilo	4	rD	rA	rB	558	EVX	**SP**
evmergelo	4	rD	rA	rB	557	EVX	**SP**
evmergelohi	4	rD	rA	rB	559	EVX	**SP**
evmhegsmfaa	4	rD	rA	rB	1323	EVX	**SP**
evmhegsmfan	4	rD	rA	rB	1451	EVX	**SP**
evmhegsmiaa	4	rD	rA	rB	1321	EVX	**SP**
evmhegsmian	4	rD	rA	rB	1449	EVX	**SP**
evmhegumiaa	4	rD	rA	rB	1320	EVX	**SP**
evmhegumian	4	rD	rA	rB	1448	EVX	**SP**
evmhesmf	4	rD	rA	rB	1035	EVX	**SP**
evmhesmfa	4	rD	rA	rB	1067	EVX	**SP**
evmhesmfaaw	4	rD	rA	rB	1291	EVX	**SP**
evmhesmfanw	4	rD	rA	rB	1419	EVX	**SP**
evmhesmi	4	rD	rA	rB	1033	EVX	**SP**
evmhesmia	4	rD	rA	rB	1065	EVX	**SP**
evmhesmiaaw	4	rD	rA	rB	1289	EVX	**SP**
evmhesmianw	4	rD	rA	rB	1417	EVX	**SP**
evmhessf	4	rD	rA	rB	1027	EVX	**SP**
evmhessfa	4	rD	rA	rB	1059	EVX	**SP**
evmhessfaaw	4	rD	rA	rB	1283	EVX	**SP**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Category
evmhessfanw	4	rD	rA	rB	1411	EVX	**SP**
evmhessiaaw	4	rD	rA	rB	1281	EVX	**SP**
evmhessianw	4	rD	rA	rB	1409	EVX	**SP**
evmheumi	4	rD	rA	rB	1032	EVX	**SP**
evmheumia	4	rD	rA	rB	1064	EVX	**SP**
evmheumiaaw	4	rD	rA	rB	1288	EVX	**SP**
evmheumianw	4	rD	rA	rB	1416	EVX	**SP**
evmheusiaaw	4	rD	rA	rB	1280	EVX	**SP**
evmheusianw	4	rD	rA	rB	1408	EVX	**SP**
evmhogsmfaa	4	rD	rA	rB	1327	EVX	**SP**
evmhogsmfan	4	rD	rA	rB	1455	EVX	**SP**
evmhogsmiaa	4	rD	rA	rB	1325	EVX	**SP**
evmhogsmian	4	rD	rA	rB	1453	EVX	**SP**
evmhogumiaa	4	rD	rA	rB	1324	EVX	**SP**
evmhogumian	4	rD	rA	rB	1452	EVX	**SP**
evmhosmf	4	rD	rA	rB	1039	EVX	**SP**
evmhosmfa	4	rD	rA	rB	1071	EVX	**SP**
evmhosmfaaw	4	rD	rA	rB	1295	EVX	**SP**
evmhosmfanw	4	rD	rA	rB	1423	EVX	**SP**
evmhosmi	4	rD	rA	rB	1037	EVX	**SP**
evmhosmia	4	rD	rA	rB	1069	EVX	**SP**
evmhosmiaaw	4	rD	rA	rB	1293	EVX	**SP**
evmhosmianw	4	rD	rA	rB	1421	EVX	**SP**
evmhossf	4	rD	rA	rB	1031	EVX	**SP**
evmhossfa	4	rD	rA	rB	1063	EVX	**SP**
evmhossfaaw	4	rD	rA	rB	1287	EVX	**SP**
evmhossfanw	4	rD	rA	rB	1415	EVX	**SP**
evmhossiaaw	4	rD	rA	rB	1285	EVX	**SP**
evmhossianw	4	rD	rA	rB	1413	EVX	**SP**
evmhoumi	4	rD	rA	rB	1036	EVX	**SP**
evmhoumia	4	rD	rA	rB	1068	EVX	**SP**
evmhoumiaaw	4	rD	rA	rB	1292	EVX	**SP**
evmhoumianw	4	rD	rA	rB	1420	EVX	**SP**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Category
evmhousiaaw	4	rD	rA	rB	1284	EVX	**SP**
evmhousianw	4	rD	rA	rB	1412	EVX	**SP**
evmra	4	rD	rA	///	1220	EVX	**SP**
evmwhsmf	4	rD	rA	rB	1103	EVX	**SP**
evmwhsmfa	4	rD	rA	rB	1135	EVX	**SP**
evmwhsmi	4	rD	rA	rB	1101	EVX	**SP**
evmwhsmia	4	rD	rA	rB	1133	EVX	**SP**
evmwhssf	4	rD	rA	rB	1095	EVX	**SP**
evmwhssfa	4	rD	rA	rB	1127	EVX	**SP**
evmwhssmaaw	4	rD	rA	rB	1349	EVX	**SP**
evmwhumi	4	rD	rA	rB	1100	EVX	**SP**
evmwhumia	4	rD	rA	rB	1132	EVX	**SP**
evmwlsmiaaw	4	rD	rA	rB	1353	EVX	**SP**
evmwlsmianw	4	rD	rA	rB	1481	EVX	**SP**
evmwlssiaaw	4	rD	rA	rB	1345	EVX	**SP**
evmwlssianw	4	rD	rA	rB	1473	EVX	**SP**
evmwlumi	4	rD	rA	rB	1096	EVX	**SP**
evmwlumia	4	rD	rA	rB	1128	EVX	**SP**
evmwlumiaaw	4	rD	rA	rB	1352	EVX	**SP**
evmwlumianw	4	rD	rA	rB	1480	EVX	**SP**
evmwlusiaaw	4	rD	rA	rB	1344	EVX	**SP**
evmwlusianw	4	rD	rA	rB	1472	EVX	**SP**
evmwsmf	4	rD	rA	rB	1115	EVX	**SP**
evmwsmfa	4	rD	rA	rB	1147	EVX	**SP**
evmwsmfaa	4	rD	rA	rB	1371	EVX	**SP**
evmwsmfan	4	rD	rA	rB	1499	EVX	**SP**
evmwsmi	4	rD	rA	rB	1113	EVX	**SP**
evmwsmia	4	rD	rA	rB	1145	EVX	**SP**
evmwsmiaa	4	rD	rA	rB	1369	EVX	**SP**
evmwsmian	4	rD	rA	rB	1497	EVX	**SP**
evmwssf	4	rD	rA	rB	1107	EVX	**SP**
evmwssfa	4	rD	rA	rB	1139	EVX	**SP**
evmwssfaa	4	rD	rA	rB	1363	EVX	**SP**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31		Form	Category
evmwssfan	4	rD	rA	rB	1491		EVX	**SP**
evmwumi	4	rD	rA	rB	1112		EVX	**SP**
evmwumia	4	rD	rA	rB	1144		EVX	**SP**
evmwumiaa	4	rD	rA	rB	1368		EVX	**SP**
evmwumian	4	rD	rA	rB	1496		EVX	**SP**
evnand	4	rD	rA	rB	542		EVX	**SP**
evneg	4	rD	rA	///	521		EVX	**SP**
evnor	4	rD	rA	rB	536		EVX	**SP**
evor	4	rD	rA	rB	535		EVX	**SP**
evorc	4	rD	rA	rB	539		EVX	**SP**
evrlw	4	rD	rA	rB	552		EVX	**SP**
evrlwi	4	rD	rA	UIMM	554		EVX	**SP**
evrndw	4	rD	rA	UIMM	524		EVX	**SP**
evsel	4	rD	rA	rB	79	crfS	EVX	**SP**
evslw	4	rD	rA	rB	548		EVX	**SP**
evslwi	4	rD	rA	UIMM	550		EVX	**SP**
evsplatfi	4	rD	SIMM	///	555		EVX	**SP**
evsplati	4	rD	SIMM	///	553		EVX	**SP**
evsrwis	4	rD	rA	UIMM	547		EVX	**SP**
evsrwiu	4	rD	rA	UIMM	546		EVX	**SP**
evsrws	4	rD	rA	rB	545		EVX	**SP**
evsrwu	4	rD	rA	rB	544		EVX	**SP**
evstdd	4	rD	rA	UIMM[1]	1424		EVX	**SP**
evstddepx	31	rS	rA	rB	927	/	X	**E.PD, SP**
evstddx	4	rS	rA	rB	800		EVX	**SP**
evstdh	4	rS	rA	UIMM[2]	402		EVX	**SP**
evstdhx	4	rS	rA	rB	804		EVX	**SP**
evstdw	4	rS	rA	UIMM[3]	1425		EVX	**SP**
evstdwx	4	rS	rA	rB	802		EVX	**SP**
evstwhe	4	rS	rA	UIMM[2]	408		EVX	**SP**
evstwhex	4	rS	rA	rB	816		EVX	**SP**
evstwho	4	rS	rA	UIMM[2]	410		EVX	**SP**
evstwhox	4	rS	rA	rB	820		EVX	**SP**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31	Form	Category
evstwwe	4	rS		rA	UIMM^3	1436			EVX	**SP**
evstwwex	4	rS		rA	rB	824			EVX	**SP**
evstwwo	4	rS		rA	UIMM^3	1438			EVX	**SP**
evstwwox	4	rS		rA	rB	828			EVX	**SP**
evsubfsmiaaw	4	rD		rA	///	1227			EVX	**SP**
evsubfssiaaw	4	rD		rA	///	1219			EVX	**SP**
evsubfumiaaw	4	rD		rA	///	1226			EVX	**SP**
evsubfusiaaw	4	rD		rA	///	1218			EVX	**SP**
evsubfw	4	rD		rA	rB	516			EVX	**SP**
evsubifw	4	rD		UIMM	rB	518			EVX	**SP**
evxor	4	rD		rA	rB	534			EVX	**SP**
extsb	31	rS		rA	///	954		0	X	
extsb.	31	rS		rA	///	954		1	X	
extsh	31	rS		rA	///	922		0	X	
extsh.	31	rS		rA	///	922		1	X	
extsw	31	rS		rA	///	986		0	X	**64**
extsw.	31	rS		rA	///	986		1	X	**64**
fabs	63	frD		///	frB	264		0	X	**FP**
fabs.	63	frD		///	frB	264		1	X	**FP.R**
fadd	63	frD		frA	frB	///	21	0	A	**FP**
fadd.	63	frD		frA	frB	///	21	1	A	**FP.R**
fadds	59	frD		frA	frB	///	21	0	A	**FP**
fadds.	59	frD		frA	frB	///	21	1	A	**FP.R**
fcfid	63	frD		///	frB	846		0	X	**FP**
fcfid.	63	frD		///	frB	846		1	X	**FP.R**
fcmpo	63	crD	//	frA	frB	32		/	X	**FP**
fcmpu	63	crD	//	frA	frB	0		/	X	**FP**
fctid	63	frD		///	frB	814		0	X	**FP**
fctid.	63	frD		///	frB	814		1	X	**FP.R**
fctidz	63	frD		///	frB	815		0	X	**FP**
fctidz.	63	frD		///	frB	815		1	X	**FP.R**
fctiw	63	frD		///	frB	14		0	X	**FP**
fctiw.	63	frD		///	frB	14		1	X	**FP.R**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31	Form	Category
fctiwz	63	frD	///	frB	15		0	X	**FP**
fctiwz.	63	frD	///	frB	15		1	X	**FP.R**
fdiv	63	frD	frA	frB	///	18	0	A	**FP**
fdiv.	63	frD	frA	frB	///	18	1	A	**FP.R**
fdivs	59	frD	frA	frB	///	18	0	A	**FP**
fdivs.	59	frD	frA	frB	///	18	1	A	**FP.R**
fmadd	63	frD	frA	frB	frC	29	0	A	**FP**
fmadd.	63	frD	frA	frB	frC	29	1	A	**FP.R**
fmadds	59	frD	frA	frB	frC	29	0	A	**FP**
fmadds.	59	frD	frA	frB	frC	29	1	A	**FP.R**
fmr	63	frD	///	frB	72		0	X	**FP**
fmr.	63	frD	///	frB	72		1	X	**FP.R**
fmsub	63	frD	frA	frB	frC	28	0	A	**FP**
fmsub.	63	frD	frA	frB	frC	28	1	A	**FP.R**
fmsubs	59	frD	frA	frB	frC	28	0	A	**FP**
fmsubs.	59	frD	frA	frB	frC	28	1	A	**FP.R**
fmul	63	frD	frA	///	frC	25	0	A	**FP**
fmul.	63	frD	frA	///	frC	25	1	A	**FP.R**
fmuls	59	frD	frA	///	frC	25	0	A	**FP**
fmuls.	59	frD	frA	///	frC	25	1	A	**FP.R**
fnabs	63	frD	///	frB	136		0	X	**FP**
fnabs.	63	frD	///	frB	136		1	X	**FP.R**
fneg	63	frD	///	frB	40		0	X	**FP**
fneg.	63	frD	///	frB	40		1	X	**FP.R**
fnmadd	63	frD	frA	frB	frC	31	0	A	**FP**
fnmadd.	63	frD	frA	frB	frC	31	1	A	**FP.R**
fnmadds	59	frD	frA	frB	frC	31	0	A	**FP**
fnmadds.	59	frD	frA	frB	frC	31	1	A	**FP.R**
fnmsub	63	frD	frA	frB	frC	30	0	A	**FP**
fnmsub.	63	frD	frA	frB	frC	30	1	A	**FP.R**
fnmsubs	59	frD	frA	frB	frC	30	0	A	**FP**
fnmsubs.	59	frD	frA	frB	frC	30	1	A	**FP.R**
fres	59	frD	///	frB	///	24	0	A	**FP**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31	Form	Category
fres.	59	frD	///	frB	///	24	1	A	**FP.R**
frsp	63	frD	///	frB	12		0	X	**FP**
frsp.	63	frD	///	frB	12		1	X	**FP.R**
frsqrte	63	frD	///	frB	///	26	0	A	**FP**
frsqrte.	63	frD	///	frB	///	26	1	A	**FP.R**
fsel	63	frD	frA	frB	frC	23	0	A	**FP**
fsel.	63	frD	frA	frB	frC	23	1	A	**FP.R**
fsub	63	frD	frA	frB	///	20	0	A	**FP**
fsub.	63	frD	frA	frB	///	20	1	A	**FP.R**
fsubs	59	frD	frA	frB	///	20	0	A	**FP**
fsubs.	59	frD	frA	frB	///	20	1	A	**FP.R**
icbi	31	///	rA	rB	982		/	X	
icbiep	31	///	rA	rB	991		/	X	**E>PD**
icblc	31	CT	rA	rB	230		/	X	**E.CL**
icbt	31	CT	rA	rB	22		/	X	**Embedded**
icbtls	31	CT	rA	rB	486		/	X	**E.CL**
isel	31	rD	rA	rB	crb	0 1 1 1 1	0	A	
isync	19	///			150		/	XL	
lbarx	31	rD	rA	rB	52		/	X	**ER**
lbdx	31	rD	rA	rB	515		/	X	**DS**
lbepx	31	rD	rA	rB	95		/	X	**E.PD**
lbz	34	rD	rA	D				D	
lbzu	35	rD	rA	D				D	
lbzux	31	rD	rA	rB	119		/	X	
lbzx	31	rD	rA	rB	87		/	X	
ld	58	rD	rA	DS		0	0	DS	**64**
ldarx	31	rD	rA	rB	84		/	X	**64**
ldbrx	31	rD	rA	rB	532		/	X	**64**
lddx	31	rD	rA	rB	611		/	X	**DS, 64**
ldepx	31	rD	rA	rB	31		/	X	**E.PD, 64**
ldu	58	rD	rA	DS		0	1	DS	**64**
ldux	31	rD	rA	rB	53		/	X	**64**
ldx	31	rD	rA	rB	21		/	X	**64**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29	30	31	Form	Category
lfd	50	frD	rA	D				D	**FP**
lfddx	31	frD	rA	rB	803		/	X	**DS, FP**
lfdepx	31	frD	rA	rB	607		/	X	**E.PD, FP**
lfdu	51	frD	rA	D				D	**FP**
lfdux	31	frD	rA	rB	631		/	X	**FP**
lfdx	31	frD	rA	rB	599		/	X	**FP**
lfs	48	frD	rA	D				D	**FP**
lfsu	49	frD	rA	D				D	**FP**
lfsux	31	frD	rA	rB	567		/	X	**FP**
lfsx	31	frD	rA	rB	535		/	X	
lha	42	rD	rA	D				D	
lharx	31	rD	rA	rB	116		/	X	**ER**
lhau	43	rD	rA	D				D	
lhaux	31	rD	rA	rB	375		/	X	
lhax	31	rD	rA	rB	343		/	X	
lhbrx	31	rD	rA	rB	790		/	X	
lhdx	31	rD	rA	rB	790		/	X	**DS**
lhepx	31	rD	rA	rB	287		/	X	**E.PD**
lhz	40	rD	rA	D				D	
lhzu	41	rD	rA	D				D	
lhzux	31	rD	rA	rB	311		/	X	
lhzx	31	rD	rA	rB	279		/		
lmw	46	rD	rA	D				D	
lvebx	31	vD	rA	rB	7		/	X	**V**
lvehx	31	vD	rA	rB	39		/	X	**V**
lvepx	31	vD	rA	rB	295		/	X	**E.PD, V**
lvepxl	31	vD	rA	rB	263		/	X	**E.PD, V**
lvewx	31	vD	rA	rB	71		/	X	**V**
lvsl	31	vD	rA	rB	6		/	X	**V**
lvsr	31	vD	rA	rB	38		/	X	**V**
lvx	31	vD	rA	rB	103		/	X	**V**
lvxl	31	vD	rA	rB	359		/	X	**V**
lwa	58	rD	rA	DS		1	0	DS	**64**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

<table>
<tr><th>Mnemonic</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th><th>Form</th><th>Category</th></tr>
<tr><td>lwarx</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">20</td><td>/</td><td>X</td><td></td></tr>
<tr><td>lwaux</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">373</td><td>/</td><td>X</td><td>64</td></tr>
<tr><td>lwax</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">341</td><td>/</td><td>X</td><td>64</td></tr>
<tr><td>lwbrx</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">534</td><td>/</td><td>X</td><td></td></tr>
<tr><td>lwdx</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">579</td><td>/</td><td>X</td><td>DS</td></tr>
<tr><td>lwepx</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">31</td><td>/</td><td>X</td><td>E.PD</td></tr>
<tr><td>lwz</td><td colspan="6">32</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="16">D</td><td>D</td><td></td></tr>
<tr><td>lwzu</td><td colspan="6">33</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="16">D</td><td>D</td><td></td></tr>
<tr><td>lwzux</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">55</td><td>/</td><td>X</td><td></td></tr>
<tr><td>lwzx</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">23</td><td>/</td><td>X</td><td></td></tr>
<tr><td>mbar</td><td colspan="6">31</td><td colspan="5">MO</td><td colspan="10">///</td><td colspan="10">854</td><td>/</td><td>X</td><td>Embedded</td></tr>
<tr><td>mcrf</td><td colspan="6">19</td><td colspan="3">crD</td><td colspan="2">//</td><td colspan="3">crfS</td><td colspan="7">///</td><td colspan="10">0</td><td>/</td><td>XL</td><td></td></tr>
<tr><td>mcrfs</td><td colspan="6">63</td><td colspan="3">crD</td><td colspan="2">//</td><td colspan="3">crfS</td><td colspan="7">///</td><td colspan="10">64</td><td>/</td><td>X</td><td>FP</td></tr>
<tr><td>mcrxr</td><td colspan="6">31</td><td colspan="3">crD</td><td colspan="12">///</td><td colspan="10">512</td><td>/</td><td>X</td><td></td></tr>
<tr><td>mfcr</td><td colspan="6">31</td><td colspan="5">rD</td><td>0</td><td colspan="9">///</td><td colspan="10">19</td><td>/</td><td>X</td><td></td></tr>
<tr><td>mfdcr</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="5">DCRN5–9</td><td colspan="5">DCRN0–4</td><td colspan="10">161</td><td>/</td><td>XFX</td><td>E.DC</td></tr>
<tr><td>mffs</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="10">///</td><td colspan="10">583</td><td>0</td><td>X</td><td>FP</td></tr>
<tr><td>mffs.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="10">///</td><td colspan="10">583</td><td>1</td><td>X</td><td>FP.R</td></tr>
<tr><td>mfmsr</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="10">///</td><td colspan="10">83</td><td>/</td><td>X</td><td></td></tr>
<tr><td>mfocrf</td><td colspan="6">31</td><td colspan="5">rD</td><td>1</td><td colspan="8">CRM</td><td>/</td><td colspan="10">19</td><td>/</td><td>X</td><td></td></tr>
<tr><td>mfpmr</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="5">PMRN5–9</td><td colspan="5">PMRN0–4</td><td colspan="10">167</td><td>/</td><td>XFX</td><td>E.PM</td></tr>
<tr><td>mfspr</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="5">SPR[5–9]</td><td colspan="5">SPR[0–4]</td><td colspan="10">339</td><td>/</td><td>XFX</td><td></td></tr>
<tr><td>mftb</td><td colspan="6">31</td><td colspan="5">rD</td><td colspan="5">TBR[5–9]</td><td colspan="5">TBR[0–4]</td><td colspan="11">742</td><td>XFX</td><td></td></tr>
<tr><td>mfvscr</td><td colspan="6">4</td><td colspan="5">vD</td><td colspan="5">///</td><td colspan="5">///</td><td colspan="11">1540</td><td>VX</td><td>V</td></tr>
<tr><td>msgclr</td><td colspan="6">31</td><td colspan="5">///</td><td colspan="5">///</td><td colspan="5">rB</td><td colspan="10">238</td><td>/</td><td>X</td><td>E.PC</td></tr>
<tr><td>msgsnd</td><td colspan="6">31</td><td colspan="5">///</td><td colspan="5">///</td><td colspan="5">rB</td><td colspan="10">206</td><td>/</td><td>X</td><td>E.PC</td></tr>
<tr><td>mtcrf</td><td colspan="6">31</td><td colspan="5">rS</td><td>0</td><td colspan="8">CRM</td><td>/</td><td colspan="10">144</td><td>/</td><td>XFX</td><td></td></tr>
<tr><td>mtdcr</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">DCRN5–9</td><td colspan="5">DCRN0–4</td><td colspan="10">225</td><td>/</td><td>XFX</td><td>E.DC</td></tr>
<tr><td>mtfsb0</td><td colspan="6">63</td><td colspan="5">crbD</td><td colspan="10">///</td><td colspan="10">70</td><td>0</td><td>X</td><td>FP</td></tr>
<tr><td>mtfsb0.</td><td colspan="6">63</td><td colspan="5">crbD</td><td colspan="10">///</td><td colspan="10">70</td><td>1</td><td>X</td><td>FP.R</td></tr>
<tr><td>mtfsb1</td><td colspan="6">63</td><td colspan="5">crbD</td><td colspan="10">///</td><td colspan="10">38</td><td>0</td><td>X</td><td>FP</td></tr>
<tr><td>mtfsb1.</td><td colspan="6">63</td><td colspan="5">crbD</td><td colspan="10">///</td><td colspan="10">38</td><td>1</td><td>X</td><td>FP.R</td></tr>
<tr><td>mtfsf</td><td colspan="6">63</td><td>L</td><td colspan="8">FM</td><td>W</td><td colspan="5">frB</td><td colspan="10">711</td><td>0</td><td>XFX</td><td>FP</td></tr>
</table>

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31	Form	Category
mtfsf.	63	L FM	W	frB	711		1	XFX	**FP.R**
mtfsfi	63	crD ///	/// W	IMM /	134		0	X	**FP**
mtfsfi.	63	crD ///	/// W	IMM /	134		1	X	**FP.R**
mtmsr	31	rS	///		146		/	X	
mtocrf	31	rS	1 CRM	/	144		/	XFX	
mtpmr	31	rS	PMRN5–9	PMRN0–4	231		/	XFX	**E.PM**
mtspr	31	rS	SPR[5–9]	SPR[0–4]	467		/	XFX	
mtvscr	4	///	///	vB	802		0	VX	**V**
mulhd	31	rD	rA	rB	/	73	0	X	**64**
mulhd.	31	rD	rA	rB	/	73	1	X	**64**
mulhdu	31	rD	rA	rB	/	9	0	X	**64**
mulhdu.	31	rD	rA	rB	/	9	1	X	**64**
mulhw	31	rD	rA	rB	/	75	0	X	
mulhw.	31	rD	rA	rB	/	75	1	X	
mulhwu	31	rD	rA	rB	/	11	0	X	
mulhwu.	31	rD	rA	rB	/	11	1	X	
mulld	31	rD	rA	rB	0	233	0	X	**64**
mulld.	31	rD	rA	rB	0	233	1	X	**64**
mulldo	31	rD	rA	rB	1	233	0	X	**64**
mulldo.	31	rD	rA	rB	1	233	1	X	**64**
mulli	7	rD	rA	SIMM				D	
mullw	31	rD	rA	rB	0	235	0	X	
mullw.	31	rD	rA	rB	0	235	1	X	
mullwo	31	rD	rA	rB	1	235	0	X	
mullwo.	31	rD	rA	rB	1	235	1	X	
nand	31	rS	rA	rB	476		0	X	
nand.	31	rS	rA	rB	476		1	X	
neg	31	rD	rA	///	104		0	X	
neg.	31	rD	rA	///	104		1	X	
nego	31	rD	rA	///	616		0	X	
nego.	31	rD	rA	///	616		1	X	
nor	31	rS	rA	rB	124		0	X	
nor.	31	rS	rA	rB	124		1	X	

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26	27 28 29	30	31	Form	Category
or	31	rS	rA	rB	444				0	X	
or.	31	rS	rA	rB	444				1	X	
orc	31	rS	rA	rB	412				0	X	
orc.	31	rS	rA	rB	412				1	X	
ori	24	rS	rA	UIMM						D	
oris	25	rS	rA	UIMM						D	
popcntb	31	rS	rA	///	122				/	X	
popcntd	31	rS	rA	///	506				/	X	**64**
popcntw	31	rS	rA	///	378				/	X	
prtyd	31	rS	rA	///	186				/	X	**64**
prtyw	31	rS	rA	///	154				/	X	
rfci	19	///			51				/	XL	**Embedded**
rfdi	19	///			39				/	X	**E.ED**
rfgi	19	///			50				/	X	**E.HV**
rfi	19	///			50				/	XL	**Embedded**
rfmci	19	///			38				/	XL	**Embedded**
rldcl	30	rS	rA	rB	mb1–5	mb0	8		0	MDS	**64**
rldcl.	30	rS	rA	rB	mb1–5	mb0	8		1	MDS	**64**
rldcr	30	rS	rA	rB	me1–5	me0	9		0	MDS	**64**
rldcr.	30	rS	rA	rB	me1–5	me0	9		1	MDS	**64**
rldic	30	rS	rA	sh1–5	mb1–5	mb0	2	sh0	0	MD	**64**
rldic.	30	rS	rA	sh1–5	mb1–5	mb0	2	sh0	1	MD	**64**
rldicl	30	rS	rA	sh1–5	mb1–5	mb0	0	sh0	0	MD	**64**
rldicl.	30	rS	rA	sh1–5	mb1–5	mb0	0	sh0	1	MD	**64**
rldicr	30	rS	rA	sh1–5	me1–5	me0	1	sh0	0	MD	**64**
rldicr.	30	rS	rA	sh1–5	me1–5	me0	1	sh0	1	MD	**64**
rldimi	30	rS	rA	sh1–5	mb1–5	mb0	3	sh0	0	MD	**64**
rldimi.	30	rS	rA	sh1–5	mb1–5	mb0	3	sh0	1	MD	**64**
rlwimi	20	rS	rA	SH	MB	ME			0	M	
rlwimi.	20	rS	rA	SH	MB	ME			1	M	
rlwinm	21	rS	rA	SH	MB	ME			0	M	
rlwinm.	21	rS	rA	SH	MB	ME			1	M	
rlwnm	23	rS	rA	rB	MB	ME			0	M	

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

<table>
<tr><th>Mnemonic</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th><th>Form</th><th>Category</th></tr>
<tr><td>rlwnm.</td><td colspan="6">23</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="5">MB</td><td colspan="5">ME</td><td>1</td><td>M</td><td></td></tr>
<tr><td>sc</td><td colspan="6">17</td><td colspan="14">///</td><td colspan="7">LEV</td><td colspan="3">///</td><td>1</td><td>/</td><td>SC</td><td></td></tr>
<tr><td>sld</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">27</td><td>0</td><td>X</td><td>64</td></tr>
<tr><td>sld.</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">27</td><td>1</td><td>X</td><td>64</td></tr>
<tr><td>slw</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">24</td><td>0</td><td>X</td><td></td></tr>
<tr><td>slw.</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">24</td><td>1</td><td>X</td><td></td></tr>
<tr><td>srad</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">794</td><td>0</td><td>X</td><td>64</td></tr>
<tr><td>srad.</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">794</td><td>1</td><td>X</td><td>64</td></tr>
<tr><td>sradi</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">sh1–5</td><td colspan="9">413</td><td>sh0</td><td>0</td><td>XS</td><td>64</td></tr>
<tr><td>sradi.</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">sh1–5</td><td colspan="9">413</td><td>sh0</td><td>1</td><td>XS</td><td>64</td></tr>
<tr><td>sraw</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">792</td><td>0</td><td>X</td><td></td></tr>
<tr><td>sraw.</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">792</td><td>1</td><td>X</td><td></td></tr>
<tr><td>srawi</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">SH</td><td colspan="10">824</td><td>0</td><td>X</td><td></td></tr>
<tr><td>srawi.</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">SH</td><td colspan="10">824</td><td>1</td><td>X</td><td></td></tr>
<tr><td>srd</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">539</td><td>0</td><td>X</td><td>64</td></tr>
<tr><td>srd.</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">539</td><td>1</td><td>X</td><td>64</td></tr>
<tr><td>srw</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">536</td><td>0</td><td>X</td><td></td></tr>
<tr><td>srw.</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">536</td><td>1</td><td>X</td><td></td></tr>
<tr><td>stb</td><td colspan="6">38</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="16">D</td><td>D</td><td></td></tr>
<tr><td>stbcx.</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">694</td><td>1</td><td>X</td><td>ER</td></tr>
<tr><td>stbdx</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">643</td><td>/</td><td>X</td><td>DS</td></tr>
<tr><td>stbepx</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">223</td><td>/</td><td>X</td><td>E.PD</td></tr>
<tr><td>stbu</td><td colspan="6">39</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="16">D</td><td>D</td><td></td></tr>
<tr><td>stbux</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">247</td><td>/</td><td>X</td><td></td></tr>
<tr><td>stbx</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">215</td><td>/</td><td>X</td><td></td></tr>
<tr><td>std</td><td colspan="6">62</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="14">DS</td><td colspan="2">0</td><td>DS</td><td>64</td></tr>
<tr><td>stdbrx</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">660</td><td>/</td><td>X</td><td>64</td></tr>
<tr><td>stdcx.</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">214</td><td>1</td><td>X</td><td>64</td></tr>
<tr><td>stddx</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">739</td><td>/</td><td>X</td><td>DS, 64</td></tr>
<tr><td>stdepx</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">157</td><td>/</td><td>X</td><td>E.PD, 64</td></tr>
<tr><td>stdu</td><td colspan="6">62</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="14">DS</td><td colspan="2">1</td><td>DS</td><td>64</td></tr>
<tr><td>stdux</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">181</td><td>/</td><td>X</td><td>64</td></tr>
<tr><td>stdx</td><td colspan="6">31</td><td colspan="5">rS</td><td colspan="5">rA</td><td colspan="5">rB</td><td colspan="10">149</td><td>/</td><td>X</td><td>64</td></tr>
</table>

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
stfd	54	frS	rA	D			D	**FP**
stfddx	31	frS	rA	rB	931	/	X	**DS, FP**
stfdepx	31	frS	rA	rB	735	/	X	**E.PD, FP**
stfdu	55	frS	rA	D			D	**FP**
stfdux	31	frS	rA	rB	759	/	X	**FP**
stfdx	31	frS	rA	rB	727	/	X	**FP**
stfiwx	31	frS	rA	rB	983	/	X	**FP**
stfs	52	frS	rA	D			D	**FP**
stfsu	53	frS	rA	D			D	**FP**
stfsux	31	frS	rA	rB	695	/	X	**FP**
stfsx	31	frS	rA	rB	663	/	X	**FP**
sth	44	rS	rA	D			D	
sthbrx	31	rS	rA	rB	918	/	X	
sthcx.	31	rS	rA	rB	726	1	X	**ER**
sthdx	31	rS	rA	rB	675	/	X	**DS**
sthepx	31	rS	rA	rB	223	/	X	**E.PD**
sthu	45	rS	rA	D			D	
sthux	31	rS	rA	rB	439	/	X	
sthx	31	rS	rA	rB	407	/	X	
stmw	47	rS	rA	D			D	
stvebx	31	vS	rA	rB	135	/	X	**V**
stvehx	31	vS	rA	rB	167	/	X	**V**
stvepx	31	vS	rA	rB	807	/	X	**E.PD, V**
stvepxl	31	vS	rA	rB	775	/	X	**E.PD, V**
stvewx	31	vS	rA	rB	199	/	X	**V**
stvx	31	vS	rA	rB	231	/	X	**V**
stvxl	31	vS	rA	rB	487	/	X	**V**
stw	36	rS	rA	D			D	
stwbrx	31	rS	rA	rB	662	/	X	
stwcx.	31	rS	rA	rB	150	1	X	
stwdx	31	rS	rA	rB	707	/	X	**DS**
stwepx	31	rS	rA	rB	159	/	X	**E.PD**
stwu	37	rS	rA	D			D	

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
stwux	31	rS	rA	rB	183	/	D	
stwx	31	rS	rA	rB	151	/	D	
subf	31	rD	rA	rB	40	0	X	
subf.	31	rD	rA	rB	40	1	X	
subfc	31	rD	rA	rB	8	0	X	
subfc.	31	rD	rA	rB	8	1	X	
subfco	31	rD	rA	rB	520	0	X	
subfco.	31	rD	rA	rB	520	1	X	
subfe	31	rD	rA	rB	136	0	X	
subfe.	31	rD	rA	rB	136	1	X	
subfeo	31	rD	rA	rB	648	0	X	
subfeo.	31	rD	rA	rB	648	1	X	
subfic	8	rD	rA	SIMM			D	
subfme	31	rD	rA	///	232	0	X	
subfme.	31	rD	rA	///	232	1	X	
subfmeo	31	rD	rA	///	744	0	X	
subfmeo.	31	rD	rA	///	744	1	X	
subfo	31	rD	rA	rB	552	0	X	
subfo.	31	rD	rA	rB	552	1	X	
subfze	31	rD	rA	///	200	0	X	
subfze.	31	rD	rA	///	200	1	X	
subfzeo	31	rD	rA	///	712	0	X	
subfzeo.	31	rD	rA	///	712	1	X	
sync	31	/// L	/ E	///	598	/	X	
td	31	TO	rA	rB	68	/	X	**64**
tdi	2	TO	rA	SIMM			D	**64**
tlbilx	31	0 /// T	rA	rB	18	/	X	**Embedded**
tlbivax	31	0 ///	rA	rB	786	/	X	**Embedded**
tlbre	31	0 ///			946	/	X	**Embedded**
tlbsx	31	0 ///	rA	rB	914	/	X	**Embedded**
tlbsync	31	0 ///			566	/	X	**Embedded**
tlbwe	31	0 ///			978	/	X	**Embedded**
tw	31	TO	rA	rB	4	/	X	

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30 31	Form	Category
twi	3	TO	rA	SIMM			D	
vaddcuw	4	vD	vA	vB	384		VX	**V**
vaddfp	4	vD	vA	vB	10		VX	**V**
vaddsbs	4	vD	vA	vB	768		VX	**V**
vaddshs	4	vD	vA	vB	832		VX	**V**
vaddsws	4	vD	vA	vB	896		VX	**V**
vaddubm	4	vD	vA	vB	0		VX	**V**
vaddubs	4	vD	vA	vB	512		VX	**V**
vadduhm	4	vD	vA	vB	64		VX	**V**
vadduhs	4	vD	vA	vB	576		VX	**V**
vadduwm	4	vD	vA	vB	128		VX	**V**
vadduws	4	vD	vA	vB	640		VX	**V**
vand	4	vD	vA	vB	1028		VX	**V**
vandc	4	vD	vA	vB	1092		VX	**V**
vavgsb	4	vD	vA	vB	1282		VX	**V**
vavgsh	4	vD	vA	vB	1346		VX	**V**
vavgsw	4	vD	vA	vB	1410		VX	**V**
vavgub	4	vD	vA	vB	1026		VX	**V**
vavguh	4	vD	vA	vB	1090		VX	**V**
vavguw	4	vD	vA	vB	1154		VX	**V**
vcfsx	4	vD	UIMM	vB	842		VX	**V**
vcfux	4	vD	UIMM	vB	778		VX	**V**
vcmpbfp	4	vD	vA	vB	0	966	VC	**V**
vcmpbfp.	4	vD	vA	vB	1	966	VC	**V**
vcmpeqfp	4	vD	vA	vB	0	198	VC	**V**
vcmpeqfp.	4	vD	vA	vB	1	198	VC	**V**
vcmpequb	4	vD	vA	vB	0	6	VC	**V**
vcmpequb.	4	vD	vA	vB	1	6	VC	**V**
vcmpequh	4	vD	vA	vB	0	70	VC	**V**
vcmpequh.	4	vD	vA	vB	1	70	VC	**V**
vcmpequw	4	vD	vA	vB	0	134	VC	**V**
vcmpequw.	4	vD	vA	vB	1	134	VC	**V**
vcmpgefp	4	vD	vA	vB	0	454	VC	**V**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25	26 27 28 29 30 31	Form	Category
vcmpgefp.	4	vD	vA	vB	1	454		VC	**V**
vcmpgtfp	4	vD	vA	vB	0	710		VC	**V**
vcmpgtfp.	4	vD	vA	vB	1	710		VC	**V**
vcmpgtsb	4	vD	vA	vB	0	774		VC	**V**
vcmpgtsb.	4	vD	vA	vB	1	774		VC	**V**
vcmpgtsh	4	vD	vA	vB	0	838		VC	**V**
vcmpgtsh.	4	vD	vA	vB	1	838		VC	**V**
vcmpgtsw	4	vD	vA	vB	0	902		VC	**V**
vcmpgtsw.	4	vD	vA	vB	1	902		VC	**V**
vcmpgtub	4	vD	vA	vB	0	518		VC	**V**
vcmpgtub.	4	vD	vA	vB	1	518		VC	**V**
vcmpgtuh	4	vD	vA	vB	0	582		VC	**V**
vcmpgtuh.	4	vD	vA	vB	1	582		VC	**V**
vcmpgtuw	4	vD	vA	vB	0	646		VC	**V**
vcmpgtuw.	4	vD	vA	vB	1	646		VC	**V**
vctsxs	4	vD	UIMM	vB	970			VX	**V**
vctuxs	4	vD	UIMM	vB	906			VX	**V**
vexptefp	4	vD	///	vB	394			VX	**V**
vlogefp	4	vD	///	vB	458			VX	**V**
vmaddfp	4	vD	vA	vB	vC		46	VA	**V**
vmaxfp	4	vD	vA	vB	1034			VX	**V**
vmaxsb	4	vD	vA	vB	258			VX	**V**
vmaxsh	4	vD	vA	vB	322			VX	**V**
vmaxsw	4	vD	vA	vB	386			VX	**V**
vmaxub	4	vD	vA	vB	2			VX	**V**
vmaxuh	4	vD	vA	vB	66			VX	**V**
vmaxuw	4	vD	vA	vB	130			VX	**V**
vmhaddshs	4	vD	vA	vB	vC		32	VA	**V**
vmhraddshs	4	vD	vA	vB	vC		33	VA	**V**
vminfp	4	vD	vA	vB	1098			VX	**V**
vminsb	4	vD	vA	vB	770			VX	**V**
vminsh	4	vD	vA	vB	834			VX	**V**
vminsw	4	vD	vA	vB	898			VX	**V**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30 31	Form	Category
vminub	4	vD	vA	vB	514		VX	**V**
vminuh	4	vD	vA	vB	578		VX	**V**
vminuw	4	vD	vA	vB	642		VX	**V**
vmladduhm	4	vD	vA	vB	vC	34	VA	**V**
vmrghb	4	vD	vA	vB	12		VX	**V**
vmrghh	4	vD	vA	vB	76		VX	**V**
vmrghw	4	vD	vA	vB	140		VX	**V**
vmrglb	4	vD	vA	vB	268		VX	**V**
vmrglh	4	vD	vA	vB	332		VX	**V**
vmrglw	4	vD	vA	vB	396		VX	**V**
vmsummbm	4	vD	vA	vB	vC	37	VA	**V**
vmsumshm	4	vD	vA	vB	vC	40	VA	**V**
vmsumshs	4	vD	vA	vB	vC	41	VA	**V**
vmsumubm	4	vD	vA	vB	vC	36	VA	**V**
vmsumuhm	4	vD	vA	vB	vC	38	VA	**V**
vmsumuhs	4	vD	vA	vB	vC	39	VA	**V**
vmulesb	4	vD	vA	vB	776		VX	**V**
vmulesh	4	vD	vA	vB	840		VX	**V**
vmuleub	4	vD	vA	vB	520		VX	**V**
vmuleuh	4	vD	vA	vB	584		VX	**V**
vmulosb	4	vD	vA	vB	264		VX	**V**
vmulosh	4	vD	vA	vB	328		VX	**V**
vmuloub	4	vD	vA	vB	8		VX	**V**
vmulouh	4	vD	vA	vB	72		VX	**V**
vnmsubfp	4	vD	vA	vB	vC	47	VA	**V**
vnor	4	vD	vA	vB	1284		VX	**V**
vor	4	vD	vA	vB	1156		VX	**V**
vperm	4	vD	vA	vB	vC	43	VA	**V**
vpkpx	4	vD	vA	vB	782		VX	**V**
vpkshss	4	vD	vA	vB	398		VX	**V**
vpkshus	4	vD	vA	vB	270		VX	**V**
vpkswss	4	vD	vA	vB	462		VX	**V**
vpkswus	4	vD	vA	vB	334		VX	**V**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25	26 27 28 29 30 31	Form	Category
vpkuhum	4	vD	vA	vB	14			VX	**V**
vpkuhus	4	vD	vA	vB	142			VX	**V**
vpkuwum	4	vD	vA	vB	78			VX	**V**
vpkuwus	4	vD	vA	vB	206			VX	**V**
vrefp	4	vD	///	vB	266			VX	**V**
vrfim	4	vD	///	vB	714			VX	**V**
vrfin	4	vD	///	vB	522			VX	**V**
vrfip	4	vD	///	vB	650			VX	**V**
vrfiz	4	vD	///	vB	586			VX	**V**
vrlb	4	vD	vA	vB	4			VX	**V**
vrlh	4	vD	vA	vB	68			VX	**V**
vrlw	4	vD	vA	vB	132			VX	**V**
vrsqrtefp	4	vD	///	vB	330			VX	**V**
vsel	4	vD	vA	vB	vC		42	VA	**V**
vsl	4	vD	vA	vB	452			VX	**V**
vslb	4	vD	vA	vB	260			VX	**V**
vsldoi	4	vD	vA	vB	/	SH	44	VX	**V**
vslh	4	vD	vA	vB	324			VX	**V**
vslo	4	vD	vA	vB	1036			VX	**V**
vslw	4	vD	vA	vB	388			VX	**V**
vspltb	4	vD	UIMM	vB	524			VX	**V**
vsplth	4	vD	UIMM	vB	588			VX	**V**
vspltisb	4	vD	SIMM	///	780			VX	**V**
vspltish	4	vD	SIMM	///	844			VX	**V**
vspltisw	4	vD	SIMM	///	908			VX	**V**
vspltw	4	vD	UIMM	vB	652			VX	**V**
vsr	4	vD	vA	vB	708			VX	**V**
vsrab	4	vD	vA	vB	772			VX	**V**
vsrah	4	vD	vA	vB	836			VX	**V**
vsraw	4	vD	vA	vB	900			VX	**V**
vsrb	4	vD	vA	vB	516			VX	**V**
vsrh	4	vD	vA	vB	580			VX	**V**
vsro	4	vD	vA	vB	1100			VX	**V**

Table B-1. Instructions Sorted by Mnemonic (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
vsrw	4	vD	vA	vB	644		VX	**V**
vsubcuw	4	vD	vA	vB	1408		VX	**V**
vsubfp	4	vD	vA	vB	74		VX	**V**
vsubsbs	4	vD	vA	vB	1792		VX	**V**
vsubshs	4	vD	vA	vB	1856		VX	**V**
vsubsws	4	vD	vA	vB	1920		VX	**V**
vsububm	4	vD	vA	vB	1024		VX	**V**
vsububs	4	vD	vA	vB	1536		VX	**V**
vsubuhm	4	vD	vA	vB	1088		VX	**V**
vsubuhs	4	vD	vA	vB	1600		VX	**V**
vsubuwm	4	vD	vA	vB	1152		VX	**V**
vsubuws	4	vD	vA	vB	1664		VX	**V**
vsum2sws	4	vD	vA	vB	1672		VX	**V**
vsum4sbs	4	vD	vA	vB	1800		VX	**V**
vsum4shs	4	vD	vA	vB	1608		VX	**V**
vsum4ubs	4	vD	vA	vB	1544		VX	**V**
vsumsws	4	vD	vA	vB	1928		VX	**V**
vupkhpx	4	vD	///	vB	846		VX	**V**
vupkhsb	4	vD	///	vB	526		VX	**V**
vupkhsh	4	vD	///	vB	590		VX	**V**
vupklpx	4	vD	///	vB	974		VX	**V**
vupklsb	4	vD	///	vB	654		VX	**V**
vupklsh	4	vD	///	vB	718		VX	**V**
vxor	4	vD	vA	vB	1220		VX	**V**
wait	31	/// WC	WH ///		62	/	X	**WT**
wrtee	31	rS	///		131	/	X	**Embedded**
wrteei	31	///		E ///	163	/	X	**Embedded**
xor	31	rS	rA	rB	632		X	
xor.	31	rS	rA	rB	633		X	
xori	26	rS	rA	UIMM			D	
xoris	27	rS	rA	UIMM			D	

[1] d = UIMM * 8

[2] d = UIMM * 2

[3] d = UIMM * 4

B.2 Instructions Sorted by Opcodes (Decimal)

The following table lists instructions by their primary and extended opcodes in decimal format.

Table B-2. Instructions Sorted by Opcode (Decimal)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30 31	Form	Category
tdi	2	TO	rA	SIMM			D	**64**
twi	3	TO	rA	SIMM			D	
vaddubm	4	vD	vA	vB	0		VX	**V**
vmaxub	4	vD	vA	vB	2		VX	**V**
vrlb	4	vD	vA	vB	4		VX	**V**
vcmpequb	4	vD	vA	vB	0	6	VC	**V**
vmuloub	4	vD	vA	vB	8		VX	**V**
vaddfp	4	vD	vA	vB	10		VX	**V**
vmrghb	4	vD	vA	vB	12		VX	**V**
vpkuhum	4	vD	vA	vB	14		VX	**V**
vadduhm	4	vD	vA	vB	64		VX	**V**
vmaxuh	4	vD	vA	vB	66		VX	**V**
vrlh	4	vD	vA	vB	68		VX	**V**
vcmpequh	4	vD	vA	vB	0	70	VC	**V**
vmulouh	4	vD	vA	vB	72		VX	**V**
vsubfp	4	vD	vA	vB	74		VX	**V**
vmrghh	4	vD	vA	vB	76		VX	**V**
vpkuwum	4	vD	vA	vB	78		VX	**V**
vadduwm	4	vD	vA	vB	128		VX	**V**
vmaxuw	4	vD	vA	vB	130		VX	**V**
vrlw	4	vD	vA	vB	132		VX	**V**
vcmpequw	4	vD	vA	vB	0	134	VC	**V**
vmrghw	4	vD	vA	vB	140		VX	**V**
vpkuhus	4	vD	vA	vB	142		VX	**V**
vcmpeqfp	4	vD	vA	vB	0	198	VC	**V**
vpkuwus	4	vD	vA	vB	206		VX	**V**
vmaxsb	4	vD	vA	vB	258		VX	**V**
vslb	4	vD	vA	vB	260		VX	**V**
vmulosb	4	vD	vA	vB	264		VX	**V**
vrefp	4	vD	///	vB	266		VX	**V**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30 31	Form	Category
vmrglb	4	vD	vA	vB	268		VX	**V**
vpkshus	4	vD	vA	vB	270		VX	**V**
vmaxsh	4	vD	vA	vB	322		VX	**V**
vslh	4	vD	vA	vB	324		VX	**V**
vmulosh	4	vD	vA	vB	328		VX	**V**
vrsqrtefp	4	vD	///	vB	330		VX	**V**
vmrglh	4	vD	vA	vB	332		VX	**V**
vpkswus	4	vD	vA	vB	334		VX	**V**
vaddcuw	4	vD	vA	vB	384		VX	**V**
vmaxsw	4	vD	vA	vB	386		VX	**V**
vslw	4	vD	vA	vB	388		VX	**V**
vexptefp	4	vD	///	vB	394		VX	**V**
vmrglw	4	vD	vA	vB	396		VX	**V**
vpkshss	4	vD	vA	vB	398		VX	**V**
vsl	4	vD	vA	vB	452		VX	**V**
vcmpgefp	4	vD	vA	vB	0	454	VC	**V**
vlogefp	4	vD	///	vB	458		VX	**V**
vpkswss	4	vD	vA	vB	462		VX	**V**
evaddw	4	rD	rA	rB	512		EVX	**SP**
vaddubs	4	vD	vA	vB	512		VX	**V**
evaddiw	4	rD	UIMM	rB	514		EVX	**SP**
vminub	4	vD	vA	vB	514		VX	**V**
evsubfw	4	rD	rA	rB	516		EVX	**SP**
vsrb	4	vD	vA	vB	516		VX	**V**
evsubifw	4	rD	UIMM	rB	518		EVX	**SP**
vcmpgtub	4	vD	vA	vB	0	518	VC	**V**
evabs	4	rD	rA	///	520		EVX	**SP**
vmuleub	4	vD	vA	vB	520		VX	**V**
evneg	4	rD	rA	///	521		EVX	**SP**
evextsb	4	rD	rA	///	522		EVX	**SP**
vrfin	4	vD	///	vB	522		VX	**V**
evextsh	4	rD	rA	///	523		EVX	**SP**
evrndw	4	rD	rA	UIMM	524		EVX	**SP**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Category
vspltb	4	vD		UIMM	vB	524	VX	**V**
evcntlzw	4	rD		rA	///	525	EVX	**SP**
evcntlsw	4	rD		rA	///	526	EVX	**SP**
vupkhsb	4	vD		///	vB	526	VX	**V**
brinc	4	rD		rA	rB	527	EVX	**SP**
evand	4	rD		rA	rB	529	EVX	**SP**
evandc	4	rD		rA	rB	530	EVX	**SP**
evxor	4	rD		rA	rB	534	EVX	**SP**
evor	4	rD		rA	rB	535	EVX	**SP**
evnor	4	rD		rA	rB	536	EVX	**SP**
eveqv	4	rD		rA	rB	537	EVX	**SP**
evorc	4	rD		rA	rB	539	EVX	**SP**
evnand	4	rD		rA	rB	542	EVX	**SP**
evsrwu	4	rD		rA	rB	544	EVX	**SP**
evsrws	4	rD		rA	rB	545	EVX	**SP**
evsrwiu	4	rD		rA	UIMM	546	EVX	**SP**
evsrwis	4	rD		rA	UIMM	547	EVX	**SP**
evslw	4	rD		rA	rB	548	EVX	**SP**
evslwi	4	rD		rA	UIMM	550	EVX	**SP**
evrlw	4	rD		rA	rB	552	EVX	**SP**
evsplati	4	rD		SIMM	///	553	EVX	**SP**
evrlwi	4	rD		rA	UIMM	554	EVX	**SP**
evsplatfi	4	rD		SIMM	///	555	EVX	**SP**
evmergehi	4	rD		rA	rB	556	EVX	**SP**
evmergelo	4	rD		rA	rB	557	EVX	**SP**
evmergehilo	4	rD		rA	rB	558	EVX	**SP**
evmergelohi	4	rD		rA	rB	559	EVX	**SP**
evcmpgtu	4	crD	/ /	rA	rB	560	EVX	**SP**
evcmpgts	4	crD	/ /	rA	rB	561	EVX	**SP**
evcmpltu	4	crD	/ /	rA	rB	562	EVX	**SP**
evcmplts	4	crD	/ /	rA	rB	563	EVX	**SP**
evcmpeq	4	crD	/ /	rA	rB	564	EVX	**SP**
vadduhs	4	vD		vA	vB	576	VX	**V**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28	29 30 31	Form	Category
vminuh	4	vD		vA	vB	578			VX	**V**
vsrh	4	vD		vA	vB	580			VX	**V**
vcmpgtuh	4	vD		vA	vB	0	582		VC	**V**
vmuleuh	4	vD		vA	vB	584			VX	**V**
vrfiz	4	vD		///	vB	586			VX	**V**
vsplth	4	vD		UIMM	vB	588			VX	**V**
vupkhsh	4	vD		///	vB	590			VX	**V**
evsel	4	rD		rA	rB	79		crfS	EVX	**SP**
evfsadd	4	rD		rA	rB	640			EVX	**SP.FV**
vadduws	4	vD		vA	vB	640			VX	**V**
evfssub	4	rD		rA	rB	641			EVX	**SP.FV**
vminuw	4	vD		vA	vB	642			VX	**V**
evfsabs	4	rD		rA	///	644			EVX	**SP.FV**
vsrw	4	vD		vA	vB	644			VX	**V**
evfsnabs	4	rD		rA	///	645			EVX	**SP.FV**
evfsneg	4	rD		rA	///	646			EVX	**SP.FV**
vcmpgtuw	4	vD		vA	vB	0	582		VC	**V**
evfsmul	4	rD		rA	rB	648			EVX	**SP.FV**
evfsdiv	4	rD		rA	rB	649			EVX	**SP.FV**
vrfip	4	vD		///	vB	650			VX	**V**
evfscmpgt	4	crD	/ /	rA	rB	652			EVX	**SP.FV**
vspltw	4	vD		UIMM	vB	652			VX	**V**
evfscmplt	4	crD	/ /	rA	rB	653			EVX	**SP.FV**
evfscmpeq	4	crD	/ /	rA	rB	654			EVX	**SP.FV**
vupklsb	4	vD		///	vB	654			VX	**V**
evfscfui	4	rD		///	rB	656			EVX	**SP.FV**
evfscfsi	4	rD		///	rB	657			EVX	**SP.FV**
evfscfuf	4	rD		///	rB	658			EVX	**SP.FV**
evfscfsf	4	rD		///	rB	659			EVX	**SP.FV**
evfsctui	4	rD		///	rB	660			EVX	**SP.FV**
evfsctsi	4	rD		///	rB	661			EVX	**SP.FV**
evfsctuf	4	rD		///	rB	662			EVX	**SP.FV**
evfsctsf	4	rD		///	rB	663			EVX	**SP.FV**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Category
evfsctuiz	4	rD	///	rB	664	EVX	**SP.FV**
evfsctsiz	4	rD	///	rB	666	EVX	**SP.FV**
evfststgt	4	crD / /	rA	rB	668	EVX	**SP.FV**
evfststlt	4	crD / /	rA	rB	669	EVX	**SP.FV**
evfststeq	4	crD / /	rA	rB	670	EVX	**SP.FV**
efsadd	4	rD	rA	rB	704	EVX	**SP.FS**
efssub	4	rD	rA	rB	705	EVX	**SP.FS**
efsabs	4	rD	rA	///	708	EVX	**SP.FS**
vsr	4	vD	vA	vB	708	VX	**V**
efsnabs	4	rD	rA	///	709	EVX	**SP.FS**
efsneg	4	rD	rA	///	710	EVX	**SP.FS**
vcmpgtfp	4	vD	vA	vB	0 710	VC	**V**
efsmul	4	rD	rA	rB	712	EVX	**SP.FS**
efsdiv	4	rD	rA	rB	713	EVX	**SP.FS**
vrfim	4	vD	///	vB	714	VX	**V**
efscmpgt	4	crD / /	rA	rB	716	EVX	**SP.FS**
efscmplt	4	crD / /	rA	rB	717	EVX	**SP.FS**
efscmpeq	4	crD / /	rA	rB	718	EVX	**SP.FS**
vupklsh	4	vD	///	vB	718	VX	**V**
efscfd	4	rD	0 0 0 0 0	rB	719	EVX	**SP.FS**
efscfui	4	rD	///	rB	720	EVX	**SP.FS**
efscfsi	4	rD	///	rB	721	EVX	**SP.FS**
efscfuf	4	rD	///	rB	722	EVX	**SP.FS**
efscfsf	4	rD	///	rB	723	EVX	**SP.FS**
efsctui	4	rD	///	rB	724	EVX	**SP.FS**
efsctsi	4	rD	///	rB	725	EVX	**SP.FS**
efsctuf	4	rD	///	rB	726	EVX	**SP.FS**
efsctsf	4	rD	///	rB	727	EVX	**SP.FS**
efsctuiz	4	rD	///	rB	728	EVX	**SP.FS**
efsctsiz	4	rD	///	rB	730	EVX	**SP.FS**
efststgt	4	crD / /	rA	rB	732	EVX	**SP.FS**
efststlt	4	crD / /	rA	rB	733	EVX	**SP.FS**
efststeq	4	crD / /	rA	rB	734	EVX	**SP.FS**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8	9 10	11	12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Category
efdadd	4	rD		rA		rB	736	EVX	**SP.FD**
efdsub	4	rD		rA		rB	737	EVX	**SP.FD**
efdabs	4	rD		rA		///	740	EVX	**SP.FD**
efdnabs	4	rD		rA		///	741	EVX	**SP.FD**
efdneg	4	rD		rA		///	742	EVX	**SP.FD**
efdmul	4	rD		rA		rB	744	EVX	**SP.FD**
efddiv	4	rD		rA		rB	745	EVX	**SP.FD**
efdcmpgt	4	crD	/ /	rA		rB	748	EVX	**SP.FD**
efdcmplt	4	crD	/ /	rA		rB	749	EVX	**SP.FD**
efdcmpeq	4	crD	/ /	rA		rB	750	EVX	**SP.FD**
efdcfs	4	rD		0	0 0 0 0	rB	751	EVX	**SP.FD**
efdcfui	4	rD		///		rB	752	EVX	**SP.FD**
efdcfsi	4	rD		///		rB	753	EVX	**SP.FD**
efdcfuf	4	rD		///		rB	754	EVX	**SP.FD**
efdcfsf	4	rD		///		rB	755	EVX	**SP.FD**
efdctui	4	rD		///		rB	756	EVX	**SP.FD**
efdctsi	4	rD		///		rB	757	EVX	**SP.FD**
efdctuf	4	rD		///		rB	758	EVX	**SP.FD**
efdctsf	4	rD		///		rB	759	EVX	**SP.FD**
efdctuiz	4	rD		///		rB	760	EVX	**SP.FD**
efdctsiz	4	rD		///		rB	762	EVX	**SP.FD**
efdtstgt	4	crD	/ /	rA		rB	764	EVX	**SP.FD**
efdtstlt	4	crD	/ /	rA		rB	765	EVX	**SP.FD**
efdtsteq	4	crD	/ /	rA		rB	766	EVX	**SP.FD**
evlddx	4	rD		rA		rB	768	EVX	**SP**
vaddsbs	4	vD		vA		vB	768	VX	**V**
evldd	4	rD		rA		UIMM[1]	769	EVX	**SP**
evldwx	4	rD		rA		rB	770	EVX	**SP**
vminsb	4	vD		vA		vB	770	VX	**V**
evldw	4	rD		rA		UIMM[2]	771	EVX	**SP**
evldhx	4	rD		rA		rB	772	EVX	**SP**
vsrab	4	vD		vA		vB	772	VX	**V**
evldh	4	rD		rA		UIMM[3]	772	EVX	**SP**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30 31	Form	Category
vcmpgtsb	4	vD	vA	vB	0	774	VC	**V**
evlhhesplatx	4	rD	rA	rB	776		EVX	**SP**
vmulesb	4	vD	vA	vB	776		VX	**V**
evlhhesplat	4	rD	rA	UIMM[2]	777		EVX	**SP**
vcfux	4	vD	UIMM	vB	778		VX	**V**
evlhhousplatx	4	rD	rA	rB	780		EVX	**SP**
vspltisb	4	vD	SIMM	///	780		VX	**V**
evlhhousplat	4	rD	rA	UIMM[2]	781		EVX	**SP**
evlhhossplatx	4	rD	rA	rB	782		EVX	**SP**
vpkpx	4	vD	vA	vB	782		VX	**V**
evlhhossplat	4	rD	rA	UIMM[2]	783		EVX	**SP**
evlwhex	4	rD	rA	rB	784		EVX	**SP**
evlwhe	4	rD	rA	UIMM[2]	785		EVX	**SP**
evlwhoux	4	rD	rA	rB	788		EVX	**SP**
evlwhou	4	rD	rA	UIMM[2]	789		EVX	**SP**
evlwhosx	4	rD	rA	rB	790		EVX	**SP**
evlwhos	4	rD	rA	UIMM[2]	791		EVX	**SP**
evlwwsplatx	4	rD	rA	rB	792		EVX	**SP**
evlwwsplat	4	rD	rA	UIMM[3]	792		EVX	**SP**
evlwhsplatx	4	rD	rA	rB	793		EVX	**SP**
evlwhsplat	4	rD	rA	UIMM[2]	796		EVX	**SP**
evstddx	4	rS	rA	rB	800		EVX	**SP**
evstdd	4	rD	rA	UIMM[1]	801		EVX	**SP**
evstdwx	4	rS	rA	rB	802		EVX	**SP**
evstdw	4	rS	rA	UIMM[3]	803		EVX	**SP**
evstdhx	4	rS	rA	rB	804		EVX	**SP**
evstdh	4	rS	rA	UIMM[2]	805		EVX	**SP**
evstwhex	4	rS	rA	rB	816		EVX	**SP**
evstwhe	4	rS	rA	UIMM[2]	817		EVX	**SP**
evstwhox	4	rS	rA	rB	820		EVX	**SP**
evstwho	4	rS	rA	UIMM[2]	821		EVX	**SP**
evstwwex	4	rS	rA	rB	824		EVX	**SP**
evstwwe	4	rS	rA	UIMM[3]	825		EVX	**SP**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30 31	Form	Category
evstwwox	4	rS	rA	rB	828		EVX	**SP**
evstwwo	4	rS	rA	UIMM[3]	829		EVX	**SP**
vaddshs	4	vD	vA	vB	832		VX	**V**
vminsh	4	vD	vA	vB	834		VX	**V**
vsrah	4	vD	vA	vB	836		VX	**V**
vcmpgtsh	4	vD	vA	vB	0	840	VC	**V**
vmulesh	4	vD	vA	vB	840		VX	**V**
vcfsx	4	vD	UIMM	vB	842		VX	**V**
vspltish	4	vD	SIMM	///	844		VX	**V**
vupkhpx	4	vD	///	vB	846		VX	**V**
vaddsws	4	vD	vA	vB	896		VX	**V**
vminsw	4	vD	vA	vB	898		VX	**V**
vsraw	4	vD	vA	vB	900		VX	**V**
vcmpgtsw	4	vD	vA	vB	0	902	VC	**V**
vctuxs	4	vD	UIMM	vB	906		VX	**V**
vspltisw	4	vD	SIMM	///	908		VX	**V**
vcmpbfp	4	vD	vA	vB	0	902	VC	**V**
vctsxs	4	vD	UIMM	vB	970		VX	**V**
vupklpx	4	vD	///	vB	974		VX	**V**
vsububm	4	vD	vA	vB	1024		VX	**V**
vavgub	4	vD	vA	vB	1026		VX	**V**
evmhessf	4	rD	rA	rB	1027		EVX	**SP**
vand	4	vD	vA	vB	1028		VX	**V**
vcmpequb.	4	vD	vA	vB	1	6	VC	**V**
evmhossf	4	rD	rA	rB	1031		EVX	**SP**
evmheumi	4	rD	rA	rB	1032		EVX	**SP**
evmhesmi	4	rD	rA	rB	1033		EVX	**SP**
vmaxfp	4	vD	vA	vB	1034		VX	**V**
evmhesmf	4	rD	rA	rB	1035		EVX	**SP**
evmhoumi	4	rD	rA	rB	1036		EVX	**SP**
vslo	4	vD	vA	vB	1037		VX	**V**
evmhosmi	4	rD	rA	rB	1039		EVX	**SP**
evmhosmf	4	rD	rA	rB	1059		EVX	**SP**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30 31	Form	Category
evmhessfa	4	rD	rA	rB	1063		EVX	**SP**
evmhossfa	4	rD	rA	rB	1064		EVX	**SP**
evmheumia	4	rD	rA	rB	1065		EVX	**SP**
evmhesmia	4	rD	rA	rB	1065		EVX	**SP**
evmhesmfa	4	rD	rA	rB	1067		EVX	**SP**
evmhoumia	4	rD	rA	rB	1068		EVX	**SP**
evmhosmia	4	rD	rA	rB	1069		EVX	**SP**
evmhosmfa	4	rD	rA	rB	1071		EVX	**SP**
vsubuhm	4	vD	vA	vB	1088		VX	**V**
vavguh	4	vD	vA	vB	1090		VX	**V**
vandc	4	vD	vA	vB	1092		VX	**V**
vcmpequh.	4	vD	vA	vB	1	70	VC	**V**
evmwhssf	4	rD	rA	rB	1095		EVX	**SP**
evmwlumi	4	rD	rA	rB	1096		EVX	**SP**
vminfp	4	vD	vA	vB	1098		VX	**V**
evmwhumi	4	rD	rA	rB	1100		EVX	**SP**
vsro	4	vD	vA	vB	1100		VX	**V**
evmwhsmi	4	rD	rA	rB	1101		EVX	**SP**
evmwhsmf	4	rD	rA	rB	1103		EVX	**SP**
evmwssf	4	rD	rA	rB	1107		EVX	**SP**
evmwumi	4	rD	rA	rB	1112		EVX	**SP**
evmwsmi	4	rD	rA	rB	1113		EVX	**SP**
evmwsmf	4	rD	rA	rB	1115		EVX	**SP**
evmwhssfa	4	rD	rA	rB	1127		EVX	**SP**
evmwlumia	4	rD	rA	rB	1128		EVX	**SP**
evmwhumia	4	rD	rA	rB	1132		EVX	**SP**
evmwhsmia	4	rD	rA	rB	1133		EVX	**SP**
evmwhsmfa	4	rD	rA	rB	1135		EVX	**SP**
evmwssfa	4	rD	rA	rB	1139		EVX	**SP**
evmwumia	4	rD	rA	rB	1144		EVX	**SP**
evmwsmia	4	rD	rA	rB	1145		EVX	**SP**
evmwsmfa	4	rD	rA	rB	1147		EVX	**SP**
vsubuwm	4	vD	vA	vB	1152		VX	**V**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30 31	Form	Category
vavguw	4	vD	vA	vB	1154		VX	**V**
vor	4	vD	vA	vB	1156		VX	**V**
vcmpequw.	4	vD	vA	vB	1	134	VC	**V**
evaddusiaaw	4	rD	rA	///	1216		EVX	**SP**
evaddssiaaw	4	rD	rA	///	1217		EVX	**SP**
evsubfusiaaw	4	rD	rA	///	1218		EVX	**SP**
evsubfssiaaw	4	rD	rA	///	1219		EVX	**SP**
evmra	4	rD	rA	///	1220		EVX	**SP**
vxor	4	vD	vA	vB	1220		VX	**V**
evdivws	4	rD	rA	rB	1222		EVX	**SP**
vcmpeqfp.	4	vD	vA	vB	1	198	VC	**V**
evdivwu	4	rD	rA	rB	1223		EVX	**SP**
evaddumiaaw	4	rD	rA	///	1224		EVX	**SP**
evaddsmiaaw	4	rD	rA	///	1225		EVX	**SP**
evsubfumiaaw	4	rD	rA	///	1226		EVX	**SP**
evsubfsmiaaw	4	rD	rA	///	1227		EVX	**SP**
evmheusiaaw	4	rD	rA	rB	1280		EVX	**SP**
evmhessiaaw	4	rD	rA	rB	1281		EVX	**SP**
vavgsb	4	vD	vA	vB	1282		VX	**V**
evmhessfaaw	4	rD	rA	rB	1283		EVX	**SP**
evmhousiaaw	4	rD	rA	rB	1284		EVX	**SP**
vnor	4	vD	vA	vB	1284		VX	**V**
evmhossiaaw	4	rD	rA	rB	1285		EVX	**SP**
evmhossfaaw	4	rD	rA	rB	1287		EVX	**SP**
evmheumiaaw	4	rD	rA	rB	1288		EVX	**SP**
evmhesmiaaw	4	rD	rA	rB	1289		EVX	**SP**
evmhesmfaaw	4	rD	rA	rB	1291		EVX	**SP**
evmhoumiaaw	4	rD	rA	rB	1292		EVX	**SP**
evmhosmiaaw	4	rD	rA	rB	1293		EVX	**SP**
evmhosmfaaw	4	rD	rA	rB	1295		EVX	**SP**
evmhegumiaa	4	rD	rA	rB	1320		EVX	**SP**
evmhegsmiaa	4	rD	rA	rB	1321		EVX	**SP**
evmhegsmfaa	4	rD	rA	rB	1323		EVX	**SP**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Category
evmhogumiaa	4	rD	rA	rB	1324	EVX	**SP**
evmhogsmiaa	4	rD	rA	rB	1325	EVX	**SP**
evmhogsmfaa	4	rD	rA	rB	1327	EVX	**SP**
evmwlusiaaw	4	rD	rA	rB	1344	EVX	**SP**
evmwlssiaaw	4	rD	rA	rB	1345	EVX	**SP**
vavgsh	4	vD	vA	vB	1346	VX	**V**
evmwhssmaaw	4	rD	rA	rB	1349	EVX	**SP**
evmwlumiaaw	4	rD	rA	rB	1352	EVX	**SP**
evmwlsmiaaw	4	rD	rA	rB	1353	EVX	**SP**
evmwssfaa	4	rD	rA	rB	1363	EVX	**SP**
evmwumiaa	4	rD	rA	rB	1368	EVX	**SP**
evmwsmiaa	4	rD	rA	rB	1369	EVX	**SP**
evmwsmfaa	4	rD	rA	rB	1371	EVX	**SP**
evmheusianw	4	rD	rA	rB	1408	EVX	**SP**
vsubcuw	4	vD	vA	vB	1408	VX	**V**
evmhessianw	4	rD	rA	rB	1409	EVX	**SP**
vavgsw	4	vD	vA	vB	1410	VX	**V**
evmhessfanw	4	rD	rA	rB	1411	EVX	**SP**
evmhousianw	4	rD	rA	rB	1412	EVX	**SP**
evmhossianw	4	rD	rA	rB	1413	EVX	**SP**
evmhossfanw	4	rD	rA	rB	1415	EVX	**SP**
evmheumianw	4	rD	rA	rB	1416	EVX	**SP**
evmhesmianw	4	rD	rA	rB	1417	EVX	**SP**
evmhesmfanw	4	rD	rA	rB	1419	EVX	**SP**
evmhoumianw	4	rD	rA	rB	1420	EVX	**SP**
evmhosmianw	4	rD	rA	rB	1421	EVX	**SP**
evmhosmfanw	4	rD	rA	rB	1423	EVX	**SP**
evmhegumian	4	rD	rA	rB	1448	EVX	**SP**
evmhegsmian	4	rD	rA	rB	1449	EVX	**SP**
evmhegsmfan	4	rD	rA	rB	1451	EVX	**SP**
evmhogumian	4	rD	rA	rB	1452	EVX	**SP**
evmhogsmian	4	rD	rA	rB	1453	EVX	**SP**
evmhogsmfan	4	rD	rA	rB	1455	EVX	**SP**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25	26 27 28 29 30	31	Form	Category
evmwlusianw	4	rD	rA	rB	1472				EVX	**SP**
evmwlssianw	4	rD	rA	rB	1473				EVX	**SP**
vcmpgefp.	4	vD	vA	vB	1	454			VC	**V**
evmwlumianw	4	rD	rA	rB	1480				EVX	**SP**
evmwlsmianw	4	rD	rA	rB	1481				EVX	**SP**
evmwssfan	4	rD	rA	rB	1491				EVX	**SP**
evmwumian	4	rD	rA	rB	1496				EVX	**SP**
evmwsmian	4	rD	rA	rB	1497				EVX	**SP**
evmwsmfan	4	rD	rA	rB	1499				EVX	**SP**
vsububs	4	vD	vA	vB	454			0	VX	**V**
mfvscr	4	vD	///	///	454			0	VX	**V**
vcmpgtub.	4	vD	vA	vB	1	518			VC	**V**
vsum4ubs	4	vD	vA	vB	1544				VX	**V**
vsubuhs	4	vD	vA	vB	1600				VX	**V**
vcmpgtuh.	4	vD	vA	vB	1	582			VC	**V**
vsum4shs	4	vD	vA	vB	1608				VX	**V**
vsubuws	4	vD	vA	vB	1664				VX	**V**
vcmpgtuw.	4	vD	vA	vB	1	646			VC	**V**
vsum2sws	4	vD	vA	vB	1672				VX	**V**
vcmpgtfp.	4	vD	vA	vB	1	710			VC	**V**
vsubsbs	4	vD	vA	vB	1792				VX	**V**
vcmpgtsb.	4	vD	vA	vB	1	774			VC	**V**
vsum4sbs	4	vD	vA	vB	1800				VX	**V**
vsubshs	4	vD	vA	vB	1856				VX	**V**
vcmpgtsh.	4	vD	vA	vB	1	838			VC	**V**
vsubsws	4	vD	vA	vB	1920				VX	**V**
vcmpgtsw.	4	vD	vA	vB	1	902			VC	**V**
vsumsws	4	vD	vA	vB	1928				VX	**V**
vcmpbfp.	4	vD	vA	vB	1	966			VC	**V**
vmhaddshs	4	vD	vA	vB	vC		32		VA	**V**
vmhraddshs	4	vD	vA	vB	vC		33		VA	**V**
vmladduhm	4	vD	vA	vB	vC		34		VA	**V**
vmsumubm	4	vD	vA	vB	vC		36		VA	**V**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

<table>
<tr><th>Mnemonic</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th><th>Form</th><th>Category</th></tr>
<tr><td>vmsummbm</td><td colspan="6">4</td><td colspan="5">vD</td><td colspan="5">vA</td><td colspan="5">vB</td><td colspan="5">vC</td><td colspan="6">37</td><td>VA</td><td>V</td></tr>
<tr><td>vmsumuhm</td><td colspan="6">4</td><td colspan="5">vD</td><td colspan="5">vA</td><td colspan="5">vB</td><td colspan="5">vC</td><td colspan="6">38</td><td>VA</td><td>V</td></tr>
<tr><td>vmsumuhs</td><td colspan="6">4</td><td colspan="5">vD</td><td colspan="5">vA</td><td colspan="5">vB</td><td colspan="5">vC</td><td colspan="6">39</td><td>VA</td><td>V</td></tr>
<tr><td>vmsumshm</td><td colspan="6">4</td><td colspan="5">vD</td><td colspan="5">vA</td><td colspan="5">vB</td><td colspan="5">vC</td><td colspan="6">40</td><td>VA</td><td>V</td></tr>
<tr><td>vmsumshs</td><td colspan="6">4</td><td colspan="5">vD</td><td colspan="5">vA</td><td colspan="5">vB</td><td colspan="5">vC</td><td colspan="6">41</td><td>VA</td><td>V</td></tr>
<tr><td>vsel</td><td colspan="6">4</td><td colspan="5">vD</td><td colspan="5">vA</td><td colspan="5">vB</td><td colspan="5">vC</td><td colspan="6">42</td><td>VA</td><td>V</td></tr>
<tr><td>vperm</td><td colspan="6">4</td><td colspan="5">vD</td><td colspan="5">vA</td><td colspan="5">vB</td><td colspan="5">vC</td><td colspan="6">43</td><td>VA</td><td>V</td></tr>
<tr><td>vsldoi</td><td colspan="6">4</td><td colspan="5">vD</td><td colspan="5">vA</td><td colspan="5">vB</td><td>/</td><td colspan="4">SH</td><td colspan="6">44</td><td>VX</td><td>V</td></tr>
<tr><td>vmaddfp</td><td colspan="6">4</td><td colspan="5">vD</td><td colspan="5">vA</td><td colspan="5">vB</td><td colspan="5">vC</td><td colspan="6">46</td><td>VA</td><td>V</td></tr>
<tr><td>vnmsubfp</td><td colspan="6">4</td><td colspan="5">vD</td><td colspan="5">vA</td><td colspan="5">vB</td><td colspan="5">vC</td><td colspan="6">47</td><td>VA</td><td>V</td></tr>
<tr><td>mulli</td><td colspan="6">7</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="16">SIMM</td><td>D</td><td></td></tr>
<tr><td>subfic</td><td colspan="6">8</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="16">SIMM</td><td>D</td><td></td></tr>
<tr><td>cmpli</td><td colspan="6">10</td><td colspan="3">crD</td><td>/</td><td>L</td><td colspan="5">rA</td><td colspan="16">UIMM</td><td>D</td><td></td></tr>
<tr><td>cmpi</td><td colspan="6">11</td><td colspan="3">crD</td><td>/</td><td>L</td><td colspan="5">rA</td><td colspan="16">SIMM</td><td>D</td><td></td></tr>
<tr><td>addic</td><td colspan="6">12</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="16">SIMM</td><td>D</td><td></td></tr>
<tr><td>addic.</td><td colspan="6">13</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="16">SIMM</td><td>D</td><td></td></tr>
<tr><td>addi</td><td colspan="6">14</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="16">SIMM</td><td>D</td><td></td></tr>
<tr><td>addis</td><td colspan="6">15</td><td colspan="5">rD</td><td colspan="5">rA</td><td colspan="16">SIMM</td><td>D</td><td></td></tr>
<tr><td>bc</td><td colspan="6">16</td><td colspan="5">BO</td><td colspan="5">BI</td><td colspan="14">BD</td><td>0</td><td>0</td><td>B</td><td></td></tr>
<tr><td>bcl</td><td colspan="6">16</td><td colspan="5">BO</td><td colspan="5">BI</td><td colspan="14">BD</td><td>0</td><td>1</td><td>B</td><td></td></tr>
<tr><td>bca</td><td colspan="6">16</td><td colspan="5">BO</td><td colspan="5">BI</td><td colspan="14">BD</td><td>1</td><td>0</td><td>B</td><td></td></tr>
<tr><td>bcla</td><td colspan="6">16</td><td colspan="5">BO</td><td colspan="5">BI</td><td colspan="14">BD</td><td>1</td><td>1</td><td>B</td><td></td></tr>
<tr><td>sc</td><td colspan="6">17</td><td colspan="14">///</td><td colspan="7">LEV</td><td colspan="3">///</td><td>1</td><td>/</td><td>SC</td><td></td></tr>
<tr><td>b</td><td colspan="6">18</td><td colspan="24">LI</td><td>0</td><td>0</td><td>I</td><td></td></tr>
<tr><td>bl</td><td colspan="6">18</td><td colspan="24">LI</td><td>0</td><td>1</td><td>I</td><td></td></tr>
<tr><td>ba</td><td colspan="6">18</td><td colspan="24">LI</td><td>1</td><td>0</td><td>I</td><td></td></tr>
<tr><td>bla</td><td colspan="6">18</td><td colspan="24">LI</td><td>1</td><td>1</td><td>I</td><td></td></tr>
<tr><td>mcrf</td><td colspan="6">19</td><td colspan="3">crD</td><td colspan="2">//</td><td colspan="3">crfS</td><td colspan="7">///</td><td colspan="10">0</td><td>/</td><td>XL</td><td></td></tr>
<tr><td>bclr</td><td colspan="6">19</td><td colspan="5">BO</td><td colspan="5">BI</td><td colspan="3">///</td><td colspan="2">BH</td><td colspan="10">16</td><td>0</td><td>XL</td><td></td></tr>
<tr><td>bclrl</td><td colspan="6">19</td><td colspan="5">BO</td><td colspan="5">BI</td><td colspan="3">///</td><td colspan="2">BH</td><td colspan="10">16</td><td>1</td><td>XL</td><td></td></tr>
<tr><td>crnor</td><td colspan="6">19</td><td colspan="5">crbD</td><td colspan="5">crbA</td><td colspan="5">crbB</td><td colspan="10">33</td><td>/</td><td>XL</td><td></td></tr>
<tr><td>rfmci</td><td colspan="6">19</td><td colspan="15">///</td><td colspan="10">39</td><td>/</td><td>XL</td><td>Embedded</td></tr>
<tr><td>rfdi</td><td colspan="6">19</td><td colspan="15">///</td><td colspan="10">39</td><td>/</td><td>X</td><td>E.ED</td></tr>
</table>

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18	19 20	21 22 23 24 25	26	27	28	29	30	31	Form	Category
rfi	19	///				50						/	XL	**Embedded**
rfci	19	///				51						/	XL	**Embedded**
rfgi	19	///				102						/	X	**E.HV**
crandc	19	crbD	crbA	crbB		129						/	XL	
isync	19	///				150						/	XL	
crxor	19	crbD	crbA	crbB		193						/	XL	
dnh	19	DUI	DCTL	0		6						/	X	**E.ED**
crnand	19	crbD	crbA	crbB		225						/	XL	
crand	19	crbD	crbA	crbB		257						/	XL	
creqv	19	crbD	crbA	crbB		289						/	XL	
crorc	19	crbD	crbA	crbB		417						/	XL	
cror	19	crbD	crbA	crbB		449						/	XL	
bcctr	19	BO	BI	///	BH	528						0	XL	
bcctrl	19	BO	BI	///	BH	528						1	XL	
rlwimi	20	rS	rA	SH		MB	ME					0	M	
rlwimi.	20	rS	rA	SH		MB	ME					1	M	
rlwinm	21	rS	rA	SH		MB	ME					0	M	
rlwinm.	21	rS	rA	SH		MB	ME					1	M	
rlwnm	22	rS	rA	rB		MB	ME					0	M	
rlwnm.	23	rS	rA	rB		MB	ME					1	M	
ori	23	rS	rA	UIMM									D	
oris	23	rS	rA	UIMM									D	
xori	24	rS	rA	UIMM									D	
xoris	25	rS	rA	UIMM									D	
andi.	28	rS	rA	UIMM									D	
andis.	29	rS	rA	UIMM									D	
rldicl	30	rS	rA	sh1–5		mb1–5	mb0	0	0	0	sh0	0	MD	**64**
rldicl.	30	rS	rA	sh1–5		mb1–5	mb0	0	0	0	sh0	1	MD	**64**
rldicr	30	rS	rA	sh1–5		me1–5	me0	0	0	1	sh0	0	MD	**64**
rldicr.	30	rS	rA	sh1–5		me1–5	me0	0	0	1	sh0	1	MD	**64**
rldic	30	rS	rA	sh1–5		mb1–5	mb0	0	1	0	sh0	0	MD	**64**
rldic.	30	rS	rA	sh1–5		mb1–5	mb0	0	1	0	sh0	1	MD	**64**
rldimi	30	rS	rA	sh1–5		mb1–5	mb0	0	1	1	sh0	0	MD	**64**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26	27	28	29	30	31	Form	Category
rldimi.	30	rS	rA	sh1–5	mb1–5	mb0	0	1	1	sh0	1	MD	**64**
rldcl	30	rS	rA	rB	mb1–5	mb0	1	0	0	0	0	MDS	**64**
rldcl.	30	rS	rA	rB	mb1–5	mb0	1	0	0	0	1	MDS	**64**
rldcr	30	rS	rA	rB	me1–5	me0	1	0	0	1	0	MDS	**64**
rldcr.	30	rS	rA	rB	me1–5	me0	1	0	0	1	1	MDS	**64**
cmp	31	crD / L	rA	rB	0						/	X	
tw	31	TO	rA	rB	4						/	X	
lvsl	31	vD	rA	rB	6						/	X	**V**
lvebx	31	vD	rA	rB	7						/	X	**V**
subfc	31	rD	rA	rB	8						0	X	
subfc.	31	rD	rA	rB	8						1	X	
mulhdu	31	rD	rA	rB	/ 9						0	X	**64**
mulhdu.	31	rD	rA	rB	/ 9						1	X	**64**
addc	31	rD	rA	rB	10						0	X	
addc.	31	rD	rA	rB	10						1	X	
mulhwu	31	rD	rA	rB	/ 10						0	X	
mulhwu.	31	rD	rA	rB	/ 10						1	X	
isel	31	rD	rA	rB	crb	30						A	
tlbilx	31	0 /// T	rA	rB	18						/	X	**Embedded**
mfcr	31	rD	0 ///		19						/	X	
mfocrf	31	rD	1 CRM	/	19						/	X	
lwarx	31	rD	rA	rB	20						/	X	
ldx	31	rD	rA	rB	21						/	X	**64**
icbt	31	CT	rA	rB	22						/	X	**Embedded**
lwzx	31	rD	rA	rB	23						/	X	
slw	31	rS	rA	rB	24						0	X	
slw.	31	rS	rA	rB	24						1	X	
cntlzw	31	rS	rA	///	26						0	X	
cntlzw.	31	rS	rA	///	26						1	X	
sld	31	rS	rA	rB	27						0	X	**64**
sld.	31	rS	rA	rB	27						1	X	**64**
and	31	rS	rA	rB	28						0	X	
and.	31	rS	rA	rB	28						1	X	

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
ldepx	31	rD	rA	rB	29	/	X	**E.PD, 64**
lwepx	31	rD	rA	rB	29	/	X	**E.PD**
cmpl	31	/ L	rA	rB ///	32	/	X	
lvsr	31	vD	rA	rB	38	/	X	**V**
lvehx	31	vD	rA	rB	39	/	X	**V**
subf	31	rD	rA	rB	40	0	X	
subf.	31	rD	rA	rB	40	1	X	
lbarx	31	rD	rA	rB	52	/	X	**ER**
ldux	31	rD	rA	rB	53	/	X	**64**
dcbst	31	///	rA	rB	54	/	X	
lwzux	31	rD	rA	rB	55	/	X	
cntlzd	31	rS	rA	///	58	0	**X**	**64**
cntlzd.	31	rS	rA	///	58	1	**X**	**64**
andc	31	rS	rA	rB	60	0	X	
andc.	31	rS	rA	rB	60	1	X	
wait	31	/// WC	WH	///	62	/	X	**WT**
dcbstep	31	///	rA	rB	63	/	X	**E.PD**
td	31	TO	rA	rB	68	/	X	**64**
lvewx	31	vD	rA	rB	71	/	X	**V**
mulhd	31	rD	rA	rB	/ 73	0	X	**64**
mulhd.	31	rD	rA	rB	/ 73	1	X	**64**
mulhw	31	rD	rA	rB	/ 74	0	X	
mulhw.	31	rD	rA	rB	/ 74	1	X	
mfmsr	31	rD	///		83	/	X	
ldarx	31	rD	rA	rB	84	/	X	**64**
dcbf	31	///	rA	rB	86	/	X	
lbzx	31	rD	rA	rB	87	/	X	
lbepx	31	rD	rA	rB	95	/	X	**E.PD**
dni	31	DUI	DCTL	0	5	1	X	**E.ED**
lvx	31	vD	rA	rB	103	/	X	**V**
neg	31	rD	rA	///	104	0	X	
neg.	31	rD	rA	///	104	1	X	
lharx	31	rD	rA	rB	116	/	X	**ER**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11	12 13 14 15	16	17 18 19	20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
lbzux	31	rD	rA		rB			119	/	X	
popcntb	31	rS	rA		///			122	/	X	
nor	31	rS	rA		rB			124	0	X	
nor.	31	rS	rA		rB			124	1	X	
dcbfep	31	///	rA		rB			127	/	X	**E.PD**
wrtee	31	rS	///					131	/	X	**Embedded**
dcbtstls	31	CT	rA		rB			134	/	X	**E.CL**
stvebx	31	vS	rA		rB			135	/	X	**V**
subfe	31	rD	rA		rB			136	0	X	
subfe.	31	rD	rA		rB			136	1	X	
adde	31	rD	rA		rB			138	0	X	
adde.	31	rD	rA		rB			138	1	X	
mtcrf	31	rS	0	CRM			/	144	/	XFX	
mtocrf	31	rS	1	CRM			/	144	/	XFX	
mtmsr	31	rS	///					146	/	X	
stdx	31	rS	rA		rB			149	/	X	**64**
stwcx.	31	rS	rA		rB			150	1	X	
stwx	31	rS	rA		rB			151	/	D	
prtyw	31	rS	rA		///			154	/	X	
stdepx	31	rS	rA		rB			157	/	X	**E.PD, 64**
stwepx	31	rS	rA		rB			159	/	X	**E.PD**
wrteei	31	///			E	///		163	/	X	**Embedded**
dcbtls	31	CT	rA		rB			166	/	X	**E.CL**
stvehx	31	vS	rA		rB			167	/	X	**V**
stdux	31	rS	rA		rB			181	/	X	**64**
stwux	31	rS	rA		rB			183	/	D	
prtyd	31	rS	rA		///			186	/	X	**64**
stvewx	31	vS	rA		rB			199	/	X	**V**
subfze	31	rD	rA		///			200	0	X	
subfze.	31	rD	rA		///			200	1	X	
addze	31	rD	rA		///			202	0	X	
addze.	31	rD	rA		///			202	1	X	
msgsnd	31	///	///		rB			206	/	X	**E.PC**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31	Form	Category
stdcx.	31	rS	rA	rB	214		1	X	**64**
stbx	31	rS	rA	rB	215		/	X	
stbepx	31	rS	rA	rB	223		/	X	**E.PD**
sthepx	31	rS	rA	rB	223		/	X	**E.PD**
icblc	31	CT	rA	rB	230		/	X	**E.CL**
stvx	31	vS	rA	rB	231		/	X	**V**
mulld	31	rD	rA	rB	0	233	0	X	**64**
mulld.	31	rD	rA	rB	0	233	1	X	**64**
subfme	31	rD	rA	///	232		0	X	
subfme.	31	rD	rA	///	232		1	X	
addme	31	rD	rA	///	234		0	X	
addme.	31	rD	rA	///	234		1	X	
mullw	31	rD	rA	rB	0	235	0	X	
mullw.	31	rD	rA	rB	0	235	1	X	
msgclr	31	///	///	rB	238		/	X	**E.PC**
dcbtst	31	TH	rA	rB	246		/	X	
stbux	31	rS	rA	rB	247		/	X	
bpermd	31	rS	rA	rB	252		/	X	**64**
dcbtstep	31	TH	rA	rB	255		/	X	**E.PD**
lvepxl	31	vD	rA	rB	263		/	X	**E.PD, V**
add	31	rD	rA	rB	266		0	X	
add.	31	rD	rA	rB	266		1	X	
ehpriv	31	OC			270		/	XL	**E.HV**
dcbt	31	TH	rA	rB	278		/	X	
lhzx	31	rD	rA	rB	279		/		
eqv	31	rD	rA	rB	568			X	
eqv.	31	rD	rA	rB	569			X	
lhepx	31	rD	rA	rB	287		/	X	**E.PD**
lvepx	31	vD	rA	rB	295		/	X	**E.PD, V**
lhzux	31	rD	rA	rB	311		/	X	
xor	31	rS	rA	rB	632			X	
xor.	31	rS	rA	rB	633			X	
dcbtep	31	TH	rA	rB	319		/	X	**E.PD**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6	7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
mfdcr	31	rD			DCRN5–9	DCRN0–4	323	/	XFX	**E.DC**
mfpmr	31	rD			PMRN5–9	PMRN0–4	334	/	XFX	**E.PM**
mfspr	31	rD			SPRN[5–9]	SPRN[0–4]	339	/	XFX	
lwax	31	rD			rA	rB	341	/	X	**64**
dst	31	0	//	STRM	rA	rB	342	/	X	**V**
dstt	31	1	//	STRM	rA	rB	342	/	X	**V**
lhax	31	rD			rA	rB	343	/	X	
lvxl	31	vD			rA	rB	359	/	X	**V**
mftb	31	rD			TBRN[5–9]	TBRN[0–4]	742		XFX	
lwaux	31	rD			rA	rB	373	/	X	**64**
dstst	31	0	//	STRM	rA	rB	374	/	X	**V**
dststt	31	1	//	STRM	rA	rB	374	/	X	**V**
lhaux	31	rD			rA	rB	375	/	X	
popcntw	31	rS			rA	///	378	/	X	
dcblc	31	CT			rA	rB	390	/	X	**E.CL**
sthx	31	rS			rA	rB	407	/	X	
orc	31	rS			rA	rB	412	0	X	
orc.	31	rS			rA	rB	412	1	X	
sthux	31	rS			rA	rB	439	/	X	
or	31	rS			rA	rB	444	0	X	
or.	31	rS			rA	rB	444	1	X	
mtdcr	31	rS			DCRN5–9	DCRN0–4	451	/	XFX	**E.DC**
divdu	31	rD			rA	rB	457	0	**X**	**64**
divdu.	31	rD			rA	rB	457	1	**X**	**64**
divwu	31	rD			rA	rB	459	0	X	
divwu.	31	rD			rA	rB	459	1	X	
mtpmr	31	rS			PMRN5–9	PMRN0–4	462	/	XFX	**E.PM**
mtspr	31	rS			SPRN[5–9]	SPRN[0–4]	467	/	XFX	
dcbi	31	///			rA	rB	470	/	X	**Embedded**
dsn	31	///			rA	rB	473	/	X	**DS**
nand	31	rS			rA	rB	476	0	X	
nand.	31	rS			rA	rB	476	1	X	
icbtls	31	CT			rA	rB	486	/	X	**E.CL**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

<table>
<tr><th>Mnemonic</th><th>0 1 2 3 4 5</th><th>6</th><th>7 8</th><th>9 10</th><th>11</th><th>12 13 14 15</th><th>16 17 18 19 20</th><th>21 22 23 24 25 26 27 28 29 30</th><th>31</th><th>Form</th><th>Category</th></tr>
<tr><td>stvxl</td><td>31</td><td colspan="3">vS</td><td colspan="2">rA</td><td>rB</td><td>487</td><td>/</td><td>X</td><td>V</td></tr>
<tr><td>divd</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>489</td><td>0</td><td>X</td><td>64</td></tr>
<tr><td>divd.</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>489</td><td>1</td><td>X</td><td>64</td></tr>
<tr><td>divw</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>491</td><td>0</td><td>X</td><td></td></tr>
<tr><td>divw.</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>491</td><td>1</td><td>X</td><td></td></tr>
<tr><td>popcntd</td><td>31</td><td colspan="3">rS</td><td colspan="2">rA</td><td>///</td><td>506</td><td>/</td><td>X</td><td>64</td></tr>
<tr><td>cmpb</td><td>31</td><td colspan="3">rS</td><td colspan="2">rA</td><td>rB</td><td>508</td><td>/</td><td>X</td><td></td></tr>
<tr><td>mcrxr</td><td>31</td><td colspan="2">crD</td><td colspan="4">///</td><td>512</td><td>/</td><td>X</td><td></td></tr>
<tr><td>lbdx</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>515</td><td>/</td><td>X</td><td>DS</td></tr>
<tr><td>subfco</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>520</td><td>0</td><td>X</td><td></td></tr>
<tr><td>subfco.</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>520</td><td>1</td><td>X</td><td></td></tr>
<tr><td>addco</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>522</td><td>0</td><td>X</td><td></td></tr>
<tr><td>addco.</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>522</td><td>1</td><td>X</td><td></td></tr>
<tr><td>ldbrx</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>532</td><td>/</td><td>X</td><td>64</td></tr>
<tr><td>lfsx</td><td>31</td><td colspan="3">frD</td><td colspan="2">rA</td><td>rB</td><td>535</td><td>/</td><td>X</td><td></td></tr>
<tr><td>lwbrx</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>534</td><td>/</td><td>X</td><td></td></tr>
<tr><td>srw</td><td>31</td><td colspan="3">rS</td><td colspan="2">rA</td><td>rB</td><td>536</td><td>0</td><td>X</td><td></td></tr>
<tr><td>srw.</td><td>31</td><td colspan="3">rS</td><td colspan="2">rA</td><td>rB</td><td>536</td><td>1</td><td>X</td><td></td></tr>
<tr><td>srd</td><td>31</td><td colspan="3">rS</td><td colspan="2">rA</td><td>rB</td><td>539</td><td>0</td><td>X</td><td>64</td></tr>
<tr><td>srd.</td><td>31</td><td colspan="3">rS</td><td colspan="2">rA</td><td>rB</td><td>539</td><td>1</td><td>X</td><td>64</td></tr>
<tr><td>lhdx</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>546</td><td>/</td><td>X</td><td>DS</td></tr>
<tr><td>lvrx</td><td>31</td><td colspan="3">vD</td><td colspan="2">rA</td><td>rB</td><td>549</td><td>/</td><td>X</td><td>V</td></tr>
<tr><td>subfo</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>552</td><td>0</td><td>X</td><td></td></tr>
<tr><td>subfo.</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>552</td><td>1</td><td>X</td><td></td></tr>
<tr><td>tlbsync</td><td>31</td><td>0</td><td colspan="5">///</td><td>566</td><td>/</td><td>X</td><td>Embedded</td></tr>
<tr><td>lfsux</td><td>31</td><td colspan="3">frD</td><td colspan="2">rA</td><td>rB</td><td>567</td><td>/</td><td>X</td><td>FP</td></tr>
<tr><td>lwdx</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>579</td><td>/</td><td>X</td><td>DS</td></tr>
<tr><td>sync</td><td>31</td><td colspan="2">///</td><td>L</td><td>/</td><td>E</td><td>///</td><td>598</td><td>/</td><td>X</td><td></td></tr>
<tr><td>lfdx</td><td>31</td><td colspan="3">frD</td><td colspan="2">rA</td><td>rB</td><td>599</td><td>/</td><td>X</td><td>FP</td></tr>
<tr><td>lfdepx</td><td>31</td><td colspan="3">frD</td><td colspan="2">rA</td><td>rB</td><td>607</td><td>/</td><td>X</td><td>E.PD, FP</td></tr>
<tr><td>lddx</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>rB</td><td>611</td><td>/</td><td>X</td><td>DS, 64</td></tr>
<tr><td>nego</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>///</td><td>616</td><td>0</td><td>X</td><td></td></tr>
<tr><td>nego.</td><td>31</td><td colspan="3">rD</td><td colspan="2">rA</td><td>///</td><td>616</td><td>1</td><td>X</td><td></td></tr>
</table>

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9	10	11 12 13 14 15	16 17 18 19 20	21	22 23 24 25 26 27 28 29 30	31	Form	Category
lfdux	31	frD		rA	rB	631		/	X	**FP**
stbdx	31	rS		rA	rB	643		/	X	**DS**
subfeo	31	rD		rA	rB	648		0	X	
subfeo.	31	rD		rA	rB	648		1	X	
addeo	31	rD		rA	rB	650		0	X	
addeo.	31	rD		rA	rB	650		1	X	
stdbrx	31	rS		rA	rB	660		/	X	**64**
stwbrx	31	rS		rA	rB	662		/	X	
stfsx	31	frS		rA	rB	663		/	X	**FP**
sthdx	31	rS		rA	rB	675		/	X	**DS**
stbcx.	31	rS		rA	rB	694		1	X	**ER**
stfsux	31	frS		rA	rB	695		/	X	**FP**
stwdx	31	rS		rA	rB	707		/	X	**DS**
subfzeo	31	rD		rA	///	712		0	X	
subfzeo.	31	rD		rA	///	712		1	X	
addzeo	31	rD		rA	///	714		0	X	
addzeo.	31	rD		rA	///	714		1	X	
sthcx.	31	rS		rA	rB	726		1	X	**ER**
stfdx	31	frS		rA	rB	727		/	X	**FP**
stfdepx	31	frS		rA	rB	735		/	X	**E.PD, FP**
stddx	31	rS		rA	rB	739		/	X	**DS, 64**
subfmeo	31	rD		rA	///	744		0	X	
subfmeo.	31	rD		rA	///	744		1	X	
mulldo	31	rD		rA	rB	1	233	0	X	**64**
mulldo.	31	rD		rA	rB	1	233	1	X	**64**
addmeo	31	rD		rA	///	746		0	X	
addmeo.	31	rD		rA	///	746		1	X	
mullwo	31	rD		rA	rB	1	235	0	X	
mullwo.	31	rD		rA	rB	1	235	1	X	
dcba	31	///	0	rA	rB	758		/	X	
dcbal	31	///	1	rA	rB	758		/	X	**DEO**
stfdux	31	frS		rA	rB	759		/	X	**FP**
stvepxl	31	vS		rA	rB	775		/	X	**E.PD, V**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
addo	31	rD	rA	rB	778	0	X	
addo.	31	rD	rA	rB	778	1	X	
tlbivax	31	0 ///	rA	rB	786	/	X	**Embedded**
lhbrx	31	rD	rA	rB	790	/	X	
sraw	31	rS	rA	rB	792	0	X	
sraw.	31	rS	rA	rB	792	1	X	
srad	31	rS	rA	rB	794	0	X	**64**
srad.	31	rS	rA	rB	794	1	X	**64**
evlddepx	31	rD	rA	rB	799	/	X	**E.PD, SP**
mtvscr	31	///	///	vB	1604		VX	**V**
lfddx	31	frD	rA	rB	803	/	X	**DS, FP**
stvepx	31	vS	rA	rB	807	/	X	**E.PD, V**
dss	31	0 // STRM	///	///	822	/	X	**V**
dssall	31	1 // STRM	///	///	822	/	X	**V**
srawi	31	rS	rA	SH	824	0	X	
srawi.	31	rS	rA	SH	824	1	X	
sradi	31	rS	rA	sh1–5	413 sh0	0	XS	**64**
sradi.	31	rS	rA	sh1–5	413 sh0	1	XS	**64**
mbar	31	MO	///		854	/	X	**Embedded**
tlbsx	31	0 ///	rA	rB	914	/	X	**Embedded**
sthbrx	31	rS	rA	rB	918	/	X	
extsh	31	rS	rA	///	922	0	X	
extsh.	31	rS	rA	///	922	1	X	
evstddepx	31	rS	rA	rB	927	/	X	**E.PD, SP**
stfddx	31	frS	rA	rB	931	/	X	**DS, FP**
tlbre	31	0	///		946	/	X	**Embedded**
extsb	31	rS	rA	///	954	0	X	
extsb.	31	rS	rA	///	954	1	X	
divduo	31	rD	rA	rB	969	0	**X**	**64**
divduo.	31	rD	rA	rB	969	1	**X**	**64**
divwuo	31	rD	rA	rB	971	0	X	
divwuo.	31	rD	rA	rB	971	1	X	
tlbwe	31	0	///		978	/	X	**Embedded**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8 9	10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
icbi	31	///		rA	rB	982	/	X	
stfiwx	31	frS		rA	rB	983	/	X	**FP**
extsw	31	rS		rA	///	986	0	X	**64**
extsw.	31	rS		rA	///	986	1	X	**64**
icbiep	31	///		rA	rB	991	/	X	**E>PD**
divdo	31	rD		rA	rB	1001	0	**X**	**64**
divdo.	31	rD		rA	rB	1001	1	**X**	**64**
divwo	31	rD		rA	rB	1003	0	X	
divwo.	31	rD		rA	rB	1003	1	X	
dcbz	31	///	0	rA	rB	1014	/	X	
dcbzl	31	///	1	rA	rB	1014	/	X	**DEO**
dcbzep	31	///	0	rA	rB	1023	/	X	**E.PD**
dcbzlep	31	///	1	rA	rB	1023	/	X	**DEO, E.PD**
lwz	32	rD		rA	D			D	
lwzu	33	rD		rA	D			D	
lbz	34	rD		rA	D			D	
lbzu	35	rD		rA	D			D	
stw	36	rS		rA	D			D	
stwu	37	rS		rA	D			D	
stb	38	rS		rA	D			D	
stbu	39	rS		rA	D			D	
lhz	40	rD		rA	D			D	
lhzu	41	rD		rA	D			D	
lha	42	rD		rA	D			D	
lhau	43	rD		rA	D			D	
sth	44	rS		rA	D			D	
sthu	45	rS		rA	D			D	
lmw	46	rD		rA	D			D	
stmw	47	rS		rA	D			D	
lfs	48	frD		rA	D			D	**FP**
lfsu	49	frD		rA	D			D	**FP**
lfd	50	frD		rA	D			D	**FP**
lfdu	51	frD		rA	D			D	**FP**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29	30	31	Form	Category
stfs	52	frS		rA	D					D	**FP**
stfsu	53	frS		rA	D					D	**FP**
stfd	54	frS		rA	D					D	**FP**
stfdu	55	frS		rA	D					D	**FP**
ld	58	rD		rA	DS			0	0	DS	**64**
ldu	58	rD		rA	DS			0	1	DS	**64**
lwa	58	rD		rA	DS			1	0	DS	**64**
fdivs	59	frD		frA	frB	///	18		0	A	**FP**
fdivs.	59	frD		frA	frB	///	18		1	A	**FP.R**
fsubs	59	frD		frA	frB	///	20		0	A	**FP**
fsubs.	59	frD		frA	frB	///	20		1	A	**FP.R**
fadds	59	frD		frA	frB	///	21		0	A	**FP**
fadds.	59	frD		frA	frB	///	21		1	A	**FP.R**
fres	59	frD		///	frB	///	22		0	A	**FP**
fres.	59	frD		///	frB	///	22		1	A	**FP.R**
fmuls	59	frD		frA	///	frC	24		0	A	**FP**
fmuls.	59	frD		frA	///	frC	24		1	A	**FP.R**
fmsubs	59	frD		frA	frB	frC	25		0	A	**FP**
fmsubs.	59	frD		frA	frB	frC	25		1	A	**FP.R**
fmadds	59	frD		frA	frB	frC	29		0	A	**FP**
fmadds.	59	frD		frA	frB	frC	29		1	A	**FP.R**
fnmsubs	59	frD		frA	frB	frC	30		0	A	**FP**
fnmsubs.	59	frD		frA	frB	frC	30		1	A	**FP.R**
fnmadds	59	frD		frA	frB	frC	31		0	A	**FP**
fnmadds.	59	frD		frA	frB	frC	31		1	A	**FP.R**
std	62	rS		rA	DS			0		DS	**64**
stdu	62	rS		rA	DS			1		DS	**64**
fcmpu	63	crD	//	frA	frB	0			/	X	**FP**
frsp	63	frD		///	frB	12			0	X	**FP**
frsp.	63	frD		///	frB	12			1	X	**FP.R**
fctiw	63	frD		///	frB	14			0	X	**FP**
fctiw.	63	frD		///	frB	14			1	X	**FP.R**
fctiwz	63	frD		///	frB	15			0	X	**FP**

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

<table>
<tr><th>Mnemonic</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th><th>Form</th><th>Category</th></tr>
<tr><td>fctiwz.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">///</td><td colspan="5">frB</td><td colspan="10">15</td><td>1</td><td>X</td><td>FP.R</td></tr>
<tr><td>fdiv</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">///</td><td colspan="5">18</td><td>0</td><td>A</td><td>FP</td></tr>
<tr><td>fdiv.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">///</td><td colspan="5">18</td><td>1</td><td>A</td><td>FP.R</td></tr>
<tr><td>fadd</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">///</td><td colspan="5">20</td><td>0</td><td>A</td><td>FP</td></tr>
<tr><td>fsub</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">///</td><td colspan="5">20</td><td>0</td><td>A</td><td>FP</td></tr>
<tr><td>fsub.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">///</td><td colspan="5">21</td><td>1</td><td>A</td><td>FP.R</td></tr>
<tr><td>fadd.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">///</td><td colspan="5">21</td><td>1</td><td>A</td><td>FP.R</td></tr>
<tr><td>fsel</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">frC</td><td colspan="5">23</td><td>0</td><td>A</td><td>FP</td></tr>
<tr><td>fsel.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">frC</td><td colspan="5">23</td><td>1</td><td>A</td><td>FP.R</td></tr>
<tr><td>fmul</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">///</td><td colspan="5">frC</td><td colspan="5">25</td><td>0</td><td>A</td><td>FP</td></tr>
<tr><td>fmul.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">///</td><td colspan="5">frC</td><td colspan="5">25</td><td>1</td><td>A</td><td>FP.R</td></tr>
<tr><td>frsqrte</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">///</td><td colspan="5">frB</td><td colspan="5">///</td><td colspan="5">26</td><td>0</td><td>A</td><td>FP</td></tr>
<tr><td>frsqrte.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">///</td><td colspan="5">frB</td><td colspan="5">///</td><td colspan="5">26</td><td>1</td><td>A</td><td>FP.R</td></tr>
<tr><td>fmsub</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">frC</td><td colspan="5">28</td><td>0</td><td>A</td><td>FP</td></tr>
<tr><td>fmsub.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">frC</td><td colspan="5">28</td><td>1</td><td>A</td><td>FP.R</td></tr>
<tr><td>fmadd</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">frC</td><td colspan="5">29</td><td>0</td><td>A</td><td>FP</td></tr>
<tr><td>fmadd.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">frC</td><td colspan="5">29</td><td>1</td><td>A</td><td>FP.R</td></tr>
<tr><td>fnmsub</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">frC</td><td colspan="5">30</td><td>0</td><td>A</td><td>FP</td></tr>
<tr><td>fnmsub.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">frC</td><td colspan="5">30</td><td>1</td><td>A</td><td>FP.R</td></tr>
<tr><td>fnmadd</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">frC</td><td colspan="5">31</td><td>0</td><td>A</td><td>FP</td></tr>
<tr><td>fnmadd.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="5">frC</td><td colspan="5">31</td><td>1</td><td>A</td><td>FP.R</td></tr>
<tr><td>fcmpo</td><td colspan="6">63</td><td colspan="3">crD</td><td colspan="2">//</td><td colspan="5">frA</td><td colspan="5">frB</td><td colspan="10">32</td><td>/</td><td>X</td><td>FP</td></tr>
<tr><td>mtfsb1</td><td colspan="6">63</td><td colspan="5">crbD</td><td colspan="10">///</td><td colspan="10">38</td><td>0</td><td>X</td><td>FP</td></tr>
<tr><td>mtfsb1.</td><td colspan="6">63</td><td colspan="5">crbD</td><td colspan="10">///</td><td colspan="10">38</td><td>1</td><td>X</td><td>FP.R</td></tr>
<tr><td>fneg</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">///</td><td colspan="5">frB</td><td colspan="10">40</td><td>0</td><td>X</td><td>FP</td></tr>
<tr><td>fneg.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">///</td><td colspan="5">frB</td><td colspan="10">40</td><td>1</td><td>X</td><td>FP.R</td></tr>
<tr><td>mcrfs</td><td colspan="6">63</td><td colspan="3">crD</td><td colspan="2">//</td><td colspan="3">crfS</td><td colspan="7">///</td><td colspan="10">64</td><td>/</td><td>X</td><td>FP</td></tr>
<tr><td>mtfsb0</td><td colspan="6">63</td><td colspan="5">crbD</td><td colspan="10">///</td><td colspan="10">70</td><td>0</td><td>X</td><td>FP</td></tr>
<tr><td>mtfsb0.</td><td colspan="6">63</td><td colspan="5">crbD</td><td colspan="10">///</td><td colspan="10">70</td><td>1</td><td>X</td><td>FP.R</td></tr>
<tr><td>fmr</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">///</td><td colspan="5">frB</td><td colspan="10">72</td><td>0</td><td>X</td><td>FP</td></tr>
<tr><td>fmr.</td><td colspan="6">63</td><td colspan="5">frD</td><td colspan="5">///</td><td colspan="5">frB</td><td colspan="10">72</td><td>1</td><td>X</td><td>FP.R</td></tr>
<tr><td>mtfsfi</td><td colspan="6">63</td><td colspan="3">crD</td><td colspan="2">///</td><td colspan="4">///</td><td>W</td><td colspan="4">IMM</td><td>/</td><td colspan="10">134</td><td>0</td><td>X</td><td>FP</td></tr>
<tr><td>mtfsfi.</td><td colspan="6">63</td><td colspan="3">crD</td><td colspan="2">///</td><td colspan="4">///</td><td>W</td><td colspan="4">IMM</td><td>/</td><td colspan="10">134</td><td>1</td><td>X</td><td>FP.R</td></tr>
</table>

Table B-2. Instructions Sorted by Opcode (Decimal) (continued)

Mnemonic	0 1 2 3 4 5	6	7 8 9 10	11 12 13 14	15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Category
fnabs	63	frD		///		frB	136	0	X	**FP**
fnabs.	63	frD		///		frB	136	1	X	**FP.R**
fabs	63	frD		///		frB	264	0	X	**FP**
fabs.	63	frD		///		frB	264	1	X	**FP.R**
mffs	63	frD		///			583	0	X	**FP**
mffs.	63	frD		///			583	1	X	**FP.R**
mtfsf	63	L	FM		W	frB	711	0	XFX	**FP**
mtfsf.	63	L	FM		W	frB	711	1	XFX	**FP.R**
fctid	63	frD		///		frB	814	0	X	**FP**
fctid.	63	frD		///		frB	814	1	X	**FP.R**
fctidz	63	frD		///		frB	815	0	X	**FP**
fctidz.	63	frD		///		frB	815	1	X	**FP.R**
fcfid	63	frD		///		frB	846	0	X	**FP**
fcfid.	63	frD		///		frB	846	1	X	**FP.R**

[1] d = UIMM * 8

[2] d = UIMM * 4

[3] d = UIMM * 2

B.3 Instructions Sorted by Mnemonic (Binary)

Table B-3 lists instructions (non-VLE) in alphabetical order by mnemonic with binary values.

Table B-3. Instructions Sorted by Mnemonic (Binary)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
add	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	1	0	0	X	
add.	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	1	0	1	X	
addc	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	1	0	0	X	
addc.	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	1	0	1	X	
addco	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	1	0	0	X	
addco.	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	1	0	1	X	
adde	0	1	1	1	1	1	rD					rA					rB					0	0	1	0	0	0	1	0	1	0	0	X	
adde.	0	1	1	1	1	1	rD					rA					rB					0	0	1	0	0	0	1	0	1	0	1	X	
addeo	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	1	0	0	X	
addeo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	1	0	1	X	
addi	0	0	1	1	1	0	rD					rA					SIMM																D	
addic	0	0	1	1	0	0	rD					rA					SIMM																D	
addic.	0	0	1	1	0	1	rD					rA					SIMM																D	
addis	0	0	1	1	1	1	rD					rA					SIMM																D	
addme	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	1	0	0	X	
addme.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	1	0	1	X	
addmeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	1	0	0	X	
addmeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	1	0	1	X	
addo	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	0	1	0	1	0	0	X	
addo.	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	0	1	0	1	0	1	X	
addze	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	1	0	0	X	
addze.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	1	0	1	X	
addzeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	1	0	0	X	
addzeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	1	0	1	X	
and	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	1	0	0	0	X	
and.	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	1	0	0	1	X	
andc	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	1	1	1	1	0	0	0	X	
andc.	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	1	1	1	1	0	0	1	X	
andi.	0	1	1	1	0	0	rS					rA					UIMM																D	
andis.	0	1	1	1	0	1	rS					rA					UIMM																D	
b	0	1	0	0	1	0	LI																								0	0	I	
ba	0	1	0	0	1	0	LI																								1	0	I	

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
bc	0	1	0	0	0	0	BO					BI					BD														0	0	B	
bca	0	1	0	0	0	0	BO					BI					BD														1	0	B	
bcctr	0	1	0	0	1	1	BO					BI					///			BH		1	0	0	0	0	1	0	0	0	0	0	XL	
bcctrl	0	1	0	0	1	1	BO					BI					///			BH		1	0	0	0	0	1	0	0	0	0	1	XL	
bcl	0	1	0	0	0	0	BO					BI					BD														0	1	B	
bcla	0	1	0	0	0	0	BO					BI					BD														1	1	B	
bclr	0	1	0	0	1	1	BO					BI					///			BH		0	0	0	0	0	1	0	0	0	0	0	XL	
bclrl	0	1	0	0	1	1	BO					BI					///			BH		0	0	0	0	0	1	0	0	0	0	1	XL	
bl	0	1	0	0	1	0	LI																								0	1	I	
bla	0	1	0	0	1	0	LI																								1	1	I	
brinc	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	0	0	1	1	1	1	EVX	**SP**
bpermd	0	1	1	1	1	1	rS					rA					rB					0	0	1	1	1	1	1	1	0	0	/	X	**64**
cmp	0	1	1	1	1	1	crD			/	L	rA					rB					0	0	0	0	0	0	0	0	0	0	/	X	
cmpb	0	1	1	1	1	1	rS					rA					rB					0	1	1	1	1	1	1	1	0	0	/	X	
cmpi	0	0	1	0	1	1	crD			/	L	rA					SIMM																D	
cmpl	0	1	1	1	1	1	crD			/	L	rA					rB					0	0	0	0	1	0	0	0	0	0	/	X	
cmpli	0	0	1	0	1	0	crD			/	L	rA					UIMM																D	
cntlzd	0	1	1	1	1	1	rS					rA					///					0	0	0	0	1	1	1	0	1	0	0	**X**	**64**
cntlzd.	0	1	1	1	1	1	rS					rA					///					0	0	0	0	1	1	1	0	1	0	1	**X**	**64**
cntlzw	0	1	1	1	1	1	rS					rA					///					0	0	0	0	0	1	1	0	1	0	0	X	
cntlzw.	0	1	1	1	1	1	rS					rA					///					0	0	0	0	0	1	1	0	1	0	1	X	
crand	0	1	0	0	1	1	crbD					crbA					crbB					0	1	0	0	0	0	0	0	0	1	/	XL	
crandc	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	0	0	0	0	0	0	1	/	XL	
creqv	0	1	0	0	1	1	crbD					crbA					crbB					0	1	0	0	1	0	0	0	0	1	/	XL	
crnand	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	1	1	0	0	0	0	1	/	XL	
crnor	0	1	0	0	1	1	crbD					crbA					crbB					0	0	0	0	1	0	0	0	0	1	/	XL	
cror	0	1	0	0	1	1	crbD					crbA					crbB					0	1	1	1	0	0	0	0	0	1	/	XL	
crorc	0	1	0	0	1	1	crbD					crbA					crbB					0	1	1	0	1	0	0	0	0	1	/	XL	
crxor	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	1	0	0	0	0	0	1	/	XL	
dcba	0	1	1	1	1	1	///				0	rA					rB					1	0	1	1	1	1	0	1	1	0	/	X	
dcbal	0	1	1	1	1	1	///				1	rA					rB					1	0	1	1	1	1	0	1	1	0	/	X	**DEO**
dcbf	0	1	1	1	1	1	///					rA					rB					0	0	0	1	0	1	0	1	1	0	/	X	
dcbfep	0	1	1	1	1	1	///					rA					rB					0	0	0	1	1	1	1	1	1	1	/	X	**E.PD**
dcbi	0	1	1	1	1	1	///					rA					rB					0	1	1	1	0	1	0	1	1	0	/	X	**Embedded**
dcblc	0	1	1	1	1	1	CT					rA					rB					0	1	1	0	0	0	0	1	1	0	/	X	**E.CL**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
dcbst	0	1	1	1	1	1	///					rA					rB					0	0	0	0	1	1	0	1	1	0	/	X	
dcbstep	0	1	1	1	1	1	///					rA					rB					0	0	0	0	1	1	1	1	1	1	/	X	**E.PD**
dcbt	0	1	1	1	1	1	TH					rA					rB					0	1	0	0	0	1	0	1	1	0	/	X	
dcbtep	0	1	1	1	1	1	TH					rA					rB					0	1	0	0	1	1	1	1	1	1	/	X	**E.PD**
dcbtls	0	1	1	1	1	1	CT					rA					rB					0	0	1	0	1	0	0	1	1	0	/	X	**E.CL**
dcbtst	0	1	1	1	1	1	TH					rA					rB					0	0	1	1	1	1	0	1	1	0	/	X	
dcbtstep	0	1	1	1	1	1	TH					rA					rB					0	0	1	1	1	1	1	1	1	1	/	X	**E.PD**
dcbtstls	0	1	1	1	1	1	CT					rA					rB					0	0	1	0	0	0	0	1	1	0	/	X	**E.CL**
dcbz	0	1	1	1	1	1	///				0	rA					rB					1	1	1	1	1	1	0	1	1	0	/	X	
dcbzep	0	1	1	1	1	1	///				0	rA					rB					1	1	1	1	1	1	1	1	1	1	/	X	**E.PD**
dcbzl	0	1	1	1	1	1	///				1	rA					rB					1	1	1	1	1	1	0	1	1	0	/	X	**DEO**
dcbzlep	0	1	1	1	1	1	///				1	rA					rB					1	1	1	1	1	1	1	1	1	1	/	X	**DEO, E.PD**
divd	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	1	0	1	0	0	1	0	**X**	**64**
divd.	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	1	0	1	0	0	1	1	**X**	**64**
divdo	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	1	0	1	0	0	1	0	**X**	**64**
divdo.	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	1	0	1	0	0	1	1	**X**	**64**
divdu	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	0	0	1	0	0	1	0	**X**	**64**
divdu.	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	0	0	1	0	0	1	1	**X**	**64**
divduo	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	0	0	1	0	0	1	0	**X**	**64**
divduo.	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	0	0	1	0	0	1	1	**X**	**64**
divw	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	1	0	1	0	1	1	0	X	
divw.	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	1	0	1	0	1	1	1	X	
divwo	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	1	0	1	0	1	1	0	X	
divwo.	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	1	0	1	0	1	1	1	X	
divwu	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	0	0	1	0	1	1	0	X	
divwu.	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	0	0	1	0	1	1	1	X	
divwuo	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	0	0	1	0	1	1	0	X	
divwuo.	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	0	0	1	0	1	1	1	X	
dnh	0	1	0	0	1	1	DUI					DCTL					0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	/	X	**E.ED**
dni	0	1	1	1	1	1	DUI					DCTL					0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	X	**E.ED**
dsn	0	1	1	1	1	1	///					rA					rB					0	1	1	1	0	1	1	0	0	1	/	X	**DS**
dss	0	1	1	1	1	1	0	//		STRM		///					///					1	1	0	0	1	1	0	1	1	0	/	X	**V**
dssall	0	1	1	1	1	1	1	//		STRM		///					///					1	1	0	0	1	1	0	1	1	0	/	X	**V**
dst	0	1	1	1	1	1	0	//		STRM		rA					rB					0	1	0	1	0	1	0	1	1	0	/	X	**V**
dstst	0	1	1	1	1	1	0	//		STRM		rA					rB					0	1	0	1	1	1	0	1	1	0	/	X	**V**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
dststt	0	1	1	1	1	1	1	//		STRM		rA					rB					0	1	0	1	1	1	0	1	1	0	/	X	**V**
dstt	0	1	1	1	1	1	1	//		STRM		rA					rB					0	1	0	1	0	1	0	1	1	0	/	X	**V**
ehpriv	0	1	1	1	1	1	OC															0	1	0	0	0	0	1	1	1	0	/	XL	**E.HV**
efdabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	0	0	EVX	**SP.FD**
efdadd	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	0	0	0	0	EVX	**SP.FD**
efdcfs	0	0	0	1	0	0	rD					0	0	0	0	0	rB					0	1	0	1	1	1	0	1	1	1	1	EVX	**SP.FD**
efdcfsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	1	1	EVX	**SP.FD**
efdcfsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	0	1	EVX	**SP.FD**
efdcfuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	1	0	EVX	**SP.FD**
efdcfui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	0	0	EVX	**SP.FD**
efdcmpeq	0	0	0	1	0	0	crD			/ /		rA					rB					0	1	0	1	1	1	0	1	1	1	0	EVX	**SP.FD**
efdcmpgt	0	0	0	1	0	0	crD			/ /		rA					rB					0	1	0	1	1	1	0	1	1	0	0	EVX	**SP.FD**
efdcmplt	0	0	0	1	0	0	crD			/ /		rA					rB					0	1	0	1	1	1	0	1	1	0	1	EVX	**SP.FD**
efdctsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	1	1	EVX	**SP.FD**
efdctsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	0	1	EVX	**SP.FD**
efdctsiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	1	0	1	0	EVX	**SP.FD**
efdctuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	1	0	EVX	**SP.FD**
efdctui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	0	0	EVX	**SP.FD**
efdctuiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	1	0	0	0	EVX	**SP.FD**
efddiv	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	1	0	0	1	EVX	**SP.FD**
efdmul	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	1	0	0	0	EVX	**SP.FD**
efdnabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	0	1	EVX	**SP.FD**
efdneg	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	1	0	EVX	**SP.FD**
efdsub	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	0	0	0	1	EVX	**SP.FD**
efdtsteq	0	0	0	1	0	0	crD			/ /		rA					rB					0	1	0	1	1	1	1	1	1	1	0	EVX	**SP.FD**
efdtstgt	0	0	0	1	0	0	crD			/ /		rA					rB					0	1	0	1	1	1	1	1	1	0	0	EVX	**SP.FD**
efdtstlt	0	0	0	1	0	0	crD			/ /		rA					rB					0	1	0	1	1	1	1	1	1	0	1	EVX	**SP.FD**
efsabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	0	0	0	1	0	0	EVX	**SP.FS**
efsadd	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	0	0	0	0	EVX	**SP.FS**
efscfd	0	0	0	1	0	0	rD					0	0	0	0	0	rB					0	1	0	1	1	0	0	1	1	1	1	EVX	**SP.FS**
efscfsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	1	1	EVX	**SP.FS**
efscfsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	0	1	EVX	**SP.FS**
efscfuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	1	0	EVX	**SP.FS**
efscfui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	0	0	EVX	**SP.FS**
efscmpeq	0	0	0	1	0	0	crD			/ /		rA					rB					0	1	0	1	1	0	0	1	1	1	0	EVX	**SP.FS**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
efscmpgt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	0	0	EVX	**SP.FS**
efscmplt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	0	1	EVX	**SP.FS**
efsctsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	1	1	EVX	**SP.FS**
efsctsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	0	1	EVX	**SP.FS**
efsctsiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	1	0	1	0	EVX	**SP.FS**
efsctuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	1	0	EVX	**SP.FS**
efsctui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	0	0	EVX	**SP.FS**
efsctuiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	1	0	0	0	EVX	**SP.FS**
efsdiv	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	1	0	0	1	EVX	**SP.FS**
efsmul	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	1	0	0	0	EVX	**SP.FS**
efsnabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	0	0	0	1	0	1	EVX	**SP.FS**
efsneg	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	0	0	0	1	1	0	EVX	**SP.FS**
efssub	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	0	0	0	1	EVX	**SP.FS**
efststeq	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	1	0	EVX	**SP.FS**
efststgt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	0	0	EVX	**SP.FS**
efststlt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	0	1	EVX	**SP.FS**
eqv	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	1	1	0	0	0	X	
eqv.	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	1	1	0	0	1	X	
evabs	0	0	0	1	0	0	rD					rA					///					0	1	0	0	0	0	0	1	0	0	0	EVX	**SP**
evaddiw	0	0	0	1	0	0	rD					UIMM					rB					0	1	0	0	0	0	0	0	0	1	0	EVX	**SP**
evaddsmiaaw	0	0	0	1	0	0	rD					rA					///					1	0	0	1	1	0	0	1	0	0	1	EVX	**SP**
evaddssiaaw	0	0	0	1	0	0	rD					rA					///					1	0	0	1	1	0	0	0	0	0	1	EVX	**SP**
evaddumiaaw	0	0	0	1	0	0	rD					rA					///					1	0	0	1	1	0	0	1	0	0	0	EVX	**SP**
evaddusiaaw	0	0	0	1	0	0	rD					rA					///					1	0	0	1	1	0	0	0	0	0	0	EVX	**SP**
evaddw	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	0	0	0	0	0	0	EVX	**SP**
evand	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	0	1	0	0	0	1	EVX	**SP**
evandc	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	0	1	0	0	1	0	EVX	**SP**
evcmpeq	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	0	0	1	1	0	1	0	0	EVX	**SP**
evcmpgts	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	0	1	EVX	**SP**
evcmpgtu	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	0	0	EVX	**SP**
evcmplts	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	1	1	EVX	**SP**
evcmpltu	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	0	0	1	1	0	0	1	0	EVX	**SP**
evcntlsw	0	0	0	1	0	0	rD					rA					///					0	1	0	0	0	0	0	1	1	1	0	EVX	**SP**
evcntlzw	0	0	0	1	0	0	rD					rA					///					0	1	0	0	0	0	0	1	1	0	1	EVX	**SP**
evdivws	0	0	0	1	0	0	rD					rA					rB					1	0	0	1	1	0	0	0	1	1	0	EVX	**SP**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
evdivwu	0	0	0	1	0	0	rD					rA					rB					1	0	0	1	1	0	0	0	1	1	1	EVX	**SP**
eveqv	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	0	1	1	0	0	1	EVX	**SP**
evextsb	0	0	0	1	0	0	rD					rA					///					0	1	0	0	0	0	0	1	0	1	0	EVX	**SP**
evextsh	0	0	0	1	0	0	rD					rA					///					0	1	0	0	0	0	0	1	0	1	1	EVX	**SP**
evfsabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	0	0	0	0	1	0	0	EVX	**SP.FV**
evfsadd	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	0	0	0	0	0	0	0	EVX	**SP.FV**
evfscfsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	0	0	1	0	0	1	1	EVX	**SP.FV**
evfscfsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	0	0	1	0	0	0	1	EVX	**SP.FV**
evfscfuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	0	0	1	0	0	1	0	EVX	**SP.FV**
evfscfui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	0	0	1	0	0	0	0	EVX	**SP.FV**
evfscmpeq	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	1	0	EVX	**SP.FV**
evfscmpgt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	0	0	EVX	**SP.FV**
evfscmplt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	0	0	0	1	1	0	1	EVX	**SP.FV**
evfsctsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	0	0	1	0	1	1	1	EVX	**SP.FV**
evfsctsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	0	0	1	0	1	0	1	EVX	**SP.FV**
evfsctsiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	0	0	1	1	0	1	0	EVX	**SP.FV**
evfsctuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	0	0	1	0	1	1	0	EVX	**SP.FV**
evfsctui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	0	0	1	0	1	0	0	EVX	**SP.FV**
evfsctuiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	0	0	1	1	0	0	0	EVX	**SP.FV**
evfsdiv	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	0	0	0	1	0	0	1	EVX	**SP.FV**
evfsmul	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	0	0	0	1	0	0	0	EVX	**SP.FV**
evfsnabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	0	0	0	0	1	0	1	EVX	**SP.FV**
evfsneg	0	0	0	1	0	0	rD					rA					///					0	1	0	1	0	0	0	0	1	1	0	EVX	**SP.FV**
evfssub	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	0	0	0	0	0	0	1	EVX	**SP.FV**
evfststeq	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	1	0	EVX	**SP.FV**
evfststgt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	0	0	EVX	**SP.FV**
evfststlt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	0	0	1	1	1	0	1	EVX	**SP.FV**
evldd	0	0	0	1	0	0	rD					rA					UIMM[1]					0	1	1	0	0	0	0	0	0	0	1	EVX	**SP**
evlddepx	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	1	1	1	1	1	/	X	**E.PD, SP**
evlddx	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	0	0	0	0	0	EVX	**SP**
evldh	0	0	0	1	0	0	rD					rA					UIMM[2]					0	1	1	0	0	0	0	0	1	0	1	EVX	**SP**
evldhx	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	0	0	1	0	0	EVX	**SP**
evldw	0	0	0	1	0	0	rD					rA					UIMM[3]					0	1	1	0	0	0	0	0	0	1	1	EVX	**SP**
evldwx	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	0	0	0	1	0	EVX	**SP**
evlhhesplat	0	0	0	1	0	0	rD					rA					UIMM[2]					0	1	1	0	0	0	0	1	0	0	1	EVX	**SP**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
evlhhesplatx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	0	1	0	0	0	EVX	**SP**
evlhhossplat	0	0	0	1	0	0	rD	rA	UIMM[2]	0	1	1	0	0	0	0	1	1	1	1	EVX	**SP**
evlhhossplatx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	0	1	1	1	0	EVX	**SP**
evlhhousplat	0	0	0	1	0	0	rD	rA	UIMM[2]	0	1	1	0	0	0	0	1	1	0	1	EVX	**SP**
evlhhousplatx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	0	1	1	0	0	EVX	**SP**
evlwhe	0	0	0	1	0	0	rD	rA	UIMM[2]	0	1	1	0	0	0	1	0	0	0	1	EVX	**SP**
evlwhex	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	0	0	0	0	EVX	**SP**
evlwhos	0	0	0	1	0	0	rD	rA	UIMM[2]	0	1	1	0	0	0	1	0	1	1	1	EVX	**SP**
evlwhosx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	0	1	1	0	EVX	**SP**
evlwhou	0	0	0	1	0	0	rD	rA	UIMM[2]	0	1	1	0	0	0	1	0	1	0	1	EVX	**SP**
evlwhoux	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	0	1	0	0	EVX	**SP**
evlwhsplat	0	0	0	1	0	0	rD	rA	UIMM[2]	0	1	1	0	0	0	1	1	1	0	1	EVX	**SP**
evlwhsplatx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	1	1	0	0	EVX	**SP**
evlwwsplat	0	0	0	1	0	0	rD	rA	UIMM[3]	0	1	1	0	0	0	1	1	0	0	1	EVX	**SP**
evlwwsplatx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	1	0	0	0	EVX	**SP**
evmergehi	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	1	0	1	1	0	0	EVX	**SP**
evmergehilo	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	1	0	1	1	1	0	EVX	**SP**
evmergelo	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	1	0	1	1	0	1	EVX	**SP**
evmergelohi	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	1	0	1	1	1	1	EVX	**SP**
evmhegsmfaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	1	0	1	0	1	1	EVX	**SP**
evmhegsmfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	0	1	1	EVX	**SP**
evmhegsmiaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	1	0	1	0	0	1	EVX	**SP**
evmhegsmian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	0	0	1	EVX	**SP**
evmhegumiaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	1	0	1	0	0	0	EVX	**SP**
evmhegumian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	0	0	0	EVX	**SP**
evmhesmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	0	1	1	EVX	**SP**
evmhesmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	0	1	1	EVX	**SP**
evmhesmfaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	1	0	1	1	EVX	**SP**
evmhesmfanw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	0	1	1	EVX	**SP**
evmhesmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	0	0	1	EVX	**SP**
evmhesmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	0	0	1	EVX	**SP**
evmhesmiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	1	0	0	1	EVX	**SP**
evmhesmianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	0	0	1	EVX	**SP**
evmhessf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	0	0	1	1	EVX	**SP**
evmhessfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	0	0	1	1	EVX	**SP**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
evmhessfaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	0	0	1	1	EVX	**SP**
evmhessfanw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	0	0	1	1	EVX	**SP**
evmhessiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	0	0	0	1	EVX	**SP**
evmhessianw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	0	0	0	1	EVX	**SP**
evmheumi	0	0	0	1	0	0	rD					rA					rB					1	0	0	0	0	0	0	1	0	0	0	EVX	**SP**
evmheumia	0	0	0	1	0	0	rD					rA					rB					1	0	0	0	0	1	0	1	0	0	0	EVX	**SP**
evmheumiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	1	0	0	0	EVX	**SP**
evmheumianw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	1	0	0	0	EVX	**SP**
evmheusiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	0	0	0	0	EVX	**SP**
evmheusianw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	0	0	0	0	EVX	**SP**
evmhogsmfaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	1	0	1	1	1	1	EVX	**SP**
evmhogsmfan	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	1	0	1	1	1	1	EVX	**SP**
evmhogsmiaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	1	0	1	1	0	1	EVX	**SP**
evmhogsmian	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	1	0	1	1	0	1	EVX	**SP**
evmhogumiaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	1	0	1	1	0	0	EVX	**SP**
evmhogumian	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	1	0	1	1	0	0	EVX	**SP**
evmhosmf	0	0	0	1	0	0	rD					rA					rB					1	0	0	0	0	0	0	1	1	1	1	EVX	**SP**
evmhosmfa	0	0	0	1	0	0	rD					rA					rB					1	0	0	0	0	1	0	1	1	1	1	EVX	**SP**
evmhosmfaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	1	1	1	1	EVX	**SP**
evmhosmfanw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	1	1	1	1	EVX	**SP**
evmhosmi	0	0	0	1	0	0	rD					rA					rB					1	0	0	0	0	0	0	1	1	0	1	EVX	**SP**
evmhosmia	0	0	0	1	0	0	rD					rA					rB					1	0	0	0	0	1	0	1	1	0	1	EVX	**SP**
evmhosmiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	1	1	0	1	EVX	**SP**
evmhosmianw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	1	1	0	1	EVX	**SP**
evmhossf	0	0	0	1	0	0	rD					rA					rB					1	0	0	0	0	0	0	0	1	1	1	EVX	**SP**
evmhossfa	0	0	0	1	0	0	rD					rA					rB					1	0	0	0	0	1	0	0	1	1	1	EVX	**SP**
evmhossfaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	0	1	1	1	EVX	**SP**
evmhossfanw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	0	1	1	1	EVX	**SP**
evmhossiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	0	1	0	1	EVX	**SP**
evmhossianw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	0	1	0	1	EVX	**SP**
evmhoumi	0	0	0	1	0	0	rD					rA					rB					1	0	0	0	0	0	0	1	1	0	0	EVX	**SP**
evmhoumia	0	0	0	1	0	0	rD					rA					rB					1	0	0	0	0	1	0	1	1	0	0	EVX	**SP**
evmhoumiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	1	1	0	0	EVX	**SP**
evmhoumianw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	1	1	0	0	EVX	**SP**
evmhousiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	0	1	0	0	EVX	**SP**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
evmhousianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	0	1	0	0	EVX	**SP**
evmra	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	0	1	0	0	EVX	**SP**
evmwhsmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	1	1	1	EVX	**SP**
evmwhsmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	1	1	1	EVX	**SP**
evmwhsmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	1	0	1	EVX	**SP**
evmwhsmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	1	0	1	EVX	**SP**
evmwhssf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	0	1	1	1	EVX	**SP**
evmwhssfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	0	1	1	1	EVX	**SP**
evmwhssmaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	0	0	1	0	1	EVX	**SP**
evmwhumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	1	0	0	EVX	**SP**
evmwhumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	1	0	0	EVX	**SP**
evmwlsmiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	0	1	0	0	1	EVX	**SP**
evmwlsmianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	1	0	0	1	EVX	**SP**
evmwlssiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	0	0	0	0	1	EVX	**SP**
evmwlssianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	0	0	0	1	EVX	**SP**
evmwlumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	0	0	0	EVX	**SP**
evmwlumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	0	0	0	EVX	**SP**
evmwlumiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	0	1	0	0	0	EVX	**SP**
evmwlumianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	1	0	0	0	EVX	**SP**
evmwlusiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	0	0	0	0	0	EVX	**SP**
evmwlusianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	0	0	0	0	EVX	**SP**
evmwsmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	1	0	1	1	EVX	**SP**
evmwsmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	1	0	1	1	EVX	**SP**
evmwsmfaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	1	1	0	1	1	EVX	**SP**
evmwsmfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	1	0	1	1	EVX	**SP**
evmwsmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	1	0	0	1	EVX	**SP**
evmwsmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	1	0	0	1	EVX	**SP**
evmwsmiaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	1	1	0	0	1	EVX	**SP**
evmwsmian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	1	0	0	1	EVX	**SP**
evmwssf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	0	0	1	1	EVX	**SP**
evmwssfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	0	0	1	1	EVX	**SP**
evmwssfaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	1	0	0	1	1	EVX	**SP**
evmwssfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	0	0	1	1	EVX	**SP**
evmwumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	1	0	0	0	EVX	**SP**
evmwumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	1	0	0	0	EVX	**SP**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
evmwumiaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	1	0	1	1	0	0	0	EVX	**SP**
evmwumian	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	1	0	1	1	0	0	0	EVX	**SP**
evnand	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	0	1	1	1	1	0	EVX	**SP**
evneg	0	0	0	1	0	0	rD					rA					///					0	1	0	0	0	0	0	1	0	0	1	EVX	**SP**
evnor	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	0	1	1	0	0	0	EVX	**SP**
evor	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	0	1	0	1	1	1	EVX	**SP**
evorc	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	0	1	1	0	1	1	EVX	**SP**
evrlw	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	1	0	1	0	0	0	EVX	**SP**
evrlwi	0	0	0	1	0	0	rD					rA					UIMM					0	1	0	0	0	1	0	1	0	1	0	EVX	**SP**
evrndw	0	0	0	1	0	0	rD					rA					UIMM					0	1	0	0	0	0	0	1	1	0	0	EVX	**SP**
evsel	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	1	1	1	1	crfS			EVX	**SP**
evslw	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	1	0	0	1	0	0	EVX	**SP**
evslwi	0	0	0	1	0	0	rD					rA					UIMM					0	1	0	0	0	1	0	0	1	1	0	EVX	**SP**
evsplatfi	0	0	0	1	0	0	rD					SIMM					///					0	1	0	0	0	1	0	1	0	1	1	EVX	**SP**
evsplati	0	0	0	1	0	0	rD					SIMM					///					0	1	0	0	0	1	0	1	0	0	1	EVX	**SP**
evsrwis	0	0	0	1	0	0	rD					rA					UIMM					0	1	0	0	0	1	0	0	0	1	1	EVX	**SP**
evsrwiu	0	0	0	1	0	0	rD					rA					UIMM					0	1	0	0	0	1	0	0	0	1	0	EVX	**SP**
evsrws	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	1	0	0	0	0	1	EVX	**SP**
evsrwu	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	1	0	0	0	0	0	EVX	**SP**
evstdd	0	0	0	1	0	0	rD					rA					UIMM[1]					0	1	1	0	0	1	0	0	0	0	1	EVX	**SP**
evstddepx	0	1	1	1	1	1	rS					rA					rB					1	1	1	0	0	1	1	1	1	1	/	X	**E.PD, SP**
evstddx	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	0	0	0	0	0	EVX	**SP**
evstdh	0	0	0	1	0	0	rS					rA					UIMM[2]					0	1	1	0	0	1	0	0	1	0	1	EVX	**SP**
evstdhx	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	0	0	1	0	0	EVX	**SP**
evstdw	0	0	0	1	0	0	rS					rA					UIMM[3]					0	1	1	0	0	1	0	0	0	1	1	EVX	**SP**
evstdwx	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	0	0	0	1	0	EVX	**SP**
evstwhe	0	0	0	1	0	0	rS					rA					UIMM[2]					0	1	1	0	0	1	1	0	0	0	1	EVX	**SP**
evstwhex	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	1	0	0	0	0	EVX	**SP**
evstwho	0	0	0	1	0	0	rS					rA					UIMM[2]					0	1	1	0	0	1	1	0	1	0	1	EVX	**SP**
evstwhox	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	1	0	1	0	0	EVX	**SP**
evstwwe	0	0	0	1	0	0	rS					rA					UIMM[3]					0	1	1	0	0	1	1	1	0	0	1	EVX	**SP**
evstwwex	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	1	1	0	0	0	EVX	**SP**
evstwwo	0	0	0	1	0	0	rS					rA					UIMM[3]					0	1	1	0	0	1	1	1	1	0	1	EVX	**SP**
evstwwox	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	1	1	1	0	0	EVX	**SP**
evsubfsmiaaw	0	0	0	1	0	0	rD					rA					///					1	0	0	1	1	0	0	1	0	1	1	EVX	**SP**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
evsubfssiaaw	0	0	0	1	0	0	rD					rA					///					1	0	0	1	1	0	0	0	0	1	1	EVX	**SP**
evsubfumiaaw	0	0	0	1	0	0	rD					rA					///					1	0	0	1	1	0	0	1	0	1	0	EVX	**SP**
evsubfusiaaw	0	0	0	1	0	0	rD					rA					///					1	0	0	1	1	0	0	0	0	1	0	EVX	**SP**
evsubfw	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	0	0	0	1	0	0	EVX	**SP**
evsubifw	0	0	0	1	0	0	rD					UIMM					rB					0	1	0	0	0	0	0	0	1	1	0	EVX	**SP**
evxor	0	0	0	1	0	0	rD					rA					rB					0	1	0	0	0	0	1	0	1	1	0	EVX	**SP**
extsb	0	1	1	1	1	1	rS					rA					///					1	1	1	0	1	1	1	0	1	0	0	X	
extsb.	0	1	1	1	1	1	rS					rA					///					1	1	1	0	1	1	1	0	1	0	1	X	
extsh	0	1	1	1	1	1	rS					rA					///					1	1	1	0	0	1	1	0	1	0	0	X	
extsh.	0	1	1	1	1	1	rS					rA					///					1	1	1	0	0	1	1	0	1	0	1	X	
extsw	0	1	1	1	1	1	rS					rA					///					1	1	1	1	0	1	1	0	1	0	0	X	**64**
extsw.	0	1	1	1	1	1	rS					rA					///					1	1	1	1	0	1	1	0	1	0	1	X	**64**
fabs	1	1	1	1	1	1	frD					///					frB					0	1	0	0	0	0	1	0	0	0	0	X	**FP**
fabs.	1	1	1	1	1	1	frD					///					frB					0	1	0	0	0	0	1	0	0	0	1	X	**FP.R**
fadd	1	1	1	1	1	1	frD					frA					frB					///					1	0	1	0	1	0	A	**FP**
fadd.	1	1	1	1	1	1	frD					frA					frB					///					1	0	1	0	1	1	A	**FP.R**
fadds	1	1	1	0	1	1	frD					frA					frB					///					1	0	1	0	1	0	A	**FP**
fadds.	1	1	1	0	1	1	frD					frA					frB					///					1	0	1	0	1	1	A	**FP.R**
fcfid	1	1	1	1	1	1	frD					///					frB					1	1	0	1	0	0	1	1	1	0	0	X	**FP**
fcfid.	1	1	1	1	1	1	frD					///					frB					1	1	0	1	0	0	1	1	1	0	1	X	**FP.R**
fcmpo	1	1	1	1	1	1	crD			//		frA					frB					0	0	0	0	1	0	0	0	0	0	/	X	**FP**
fcmpu	1	1	1	1	1	1	crD			//		frA					frB					0	0	0	0	0	0	0	0	0	0	/	X	**FP**
fctid	1	1	1	1	1	1	frD					///					frB					1	1	0	0	1	0	1	1	1	0	0	X	**FP**
fctid.	1	1	1	1	1	1	frD					///					frB					1	1	0	0	1	0	1	1	1	0	1	X	**FP.R**
fctidz	1	1	1	1	1	1	frD					///					frB					1	1	0	0	1	0	1	1	1	1	0	X	**FP**
fctidz.	1	1	1	1	1	1	frD					///					frB					1	1	0	0	1	0	1	1	1	1	1	X	**FP.R**
fctiw	1	1	1	1	1	1	frD					///					frB					0	0	0	0	0	0	1	1	1	0	0	X	**FP**
fctiw.	1	1	1	1	1	1	frD					///					frB					0	0	0	0	0	0	1	1	1	0	1	X	**FP.R**
fctiwz	1	1	1	1	1	1	frD					///					frB					0	0	0	0	0	0	1	1	1	1	0	X	**FP**
fctiwz.	1	1	1	1	1	1	frD					///					frB					0	0	0	0	0	0	1	1	1	1	1	X	**FP.R**
fdiv	1	1	1	1	1	1	frD					frA					frB					///					1	0	0	1	0	0	A	**FP**
fdiv.	1	1	1	1	1	1	frD					frA					frB					///					1	0	0	1	0	1	A	**FP.R**
fdivs	1	1	1	0	1	1	frD					frA					frB					///					1	0	0	1	0	0	A	**FP**
fdivs.	1	1	1	0	1	1	frD					frA					frB					///					1	0	0	1	0	1	A	**FP.R**
fmadd	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	0	1	0	A	**FP**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
fmadd.	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	0	1	1	A	**FP.R**
fmadds	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	0	1	0	A	**FP**
fmadds.	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	0	1	1	A	**FP.R**
fmr	1	1	1	1	1	1	frD					///					frB					0	0	0	1	0	0	1	0	0	0	0	X	**FP**
fmr.	1	1	1	1	1	1	frD					///					frB					0	0	0	1	0	0	1	0	0	0	1	X	**FP.R**
fmsub	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	0	0	0	A	**FP**
fmsub.	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	0	0	1	A	**FP.R**
fmsubs	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	0	0	0	A	**FP**
fmsubs.	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	0	0	1	A	**FP.R**
fmul	1	1	1	1	1	1	frD					frA					///					frC					1	1	0	0	1	0	A	**FP**
fmul.	1	1	1	1	1	1	frD					frA					///					frC					1	1	0	0	1	1	A	**FP.R**
fmuls	1	1	1	0	1	1	frD					frA					///					frC					1	1	0	0	1	0	A	**FP**
fmuls.	1	1	1	0	1	1	frD					frA					///					frC					1	1	0	0	1	1	A	**FP.R**
fnabs	1	1	1	1	1	1	frD					///					frB					0	0	1	0	0	0	1	0	0	0	0	X	**FP**
fnabs.	1	1	1	1	1	1	frD					///					frB					0	0	1	0	0	0	1	0	0	0	1	X	**FP.R**
fneg	1	1	1	1	1	1	frD					///					frB					0	0	0	0	1	0	1	0	0	0	0	X	**FP**
fneg.	1	1	1	1	1	1	frD					///					frB					0	0	0	0	1	0	1	0	0	0	1	X	**FP.R**
fnmadd	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	1	1	0	A	**FP**
fnmadd.	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	1	1	1	A	**FP.R**
fnmadds	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	1	1	0	A	**FP**
fnmadds.	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	1	1	1	A	**FP.R**
fnmsub	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	1	0	0	A	**FP**
fnmsub.	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	1	0	1	A	**FP.R**
fnmsubs	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	1	0	0	A	**FP**
fnmsubs.	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	1	0	1	A	**FP.R**
fres	1	1	1	0	1	1	frD					///					frB					///					1	1	0	0	0	0	A	**FP**
fres.	1	1	1	0	1	1	frD					///					frB					///					1	1	0	0	0	1	A	**FP.R**
frsp	1	1	1	1	1	1	frD					///					frB					0	0	0	0	0	0	1	1	0	0	0	X	**FP**
frsp.	1	1	1	1	1	1	frD					///					frB					0	0	0	0	0	0	1	1	0	0	1	X	**FP.R**
frsqrte	1	1	1	1	1	1	frD					///					frB					///					1	1	0	1	0	0	A	**FP**
frsqrte.	1	1	1	1	1	1	frD					///					frB					///					1	1	0	1	0	1	A	**FP.R**
fsel	1	1	1	1	1	1	frD					frA					frB					frC					1	0	1	1	1	0	A	**FP**
fsel.	1	1	1	1	1	1	frD					frA					frB					frC					1	0	1	1	1	1	A	**FP.R**
fsub	1	1	1	1	1	1	frD					frA					frB					///					1	0	1	0	0	0	A	**FP**
fsub.	1	1	1	1	1	1	frD					frA					frB					///					1	0	1	0	0	1	A	**FP.R**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
fsubs	1	1	1	0	1	1	frD					frA					frB					///					1	0	1	0	0	0	A	FP
fsubs.	1	1	1	0	1	1	frD					frA					frB					///					1	0	1	0	0	1	A	FP.R
icbi	0	1	1	1	1	1	///					rA					rB					1	1	1	1	0	1	0	1	1	0	/	X	
icbiep	0	1	1	1	1	1	///					rA					rB					1	1	1	1	0	1	1	1	1	1	/	X	E>PD
icblc	0	1	1	1	1	1	CT					rA					rB					0	0	1	1	1	0	0	1	1	0	/	X	E.CL
icbt	0	1	1	1	1	1	CT					rA					rB					0	0	0	0	0	1	0	1	1	0	/	X	Embedded
icbtls	0	1	1	1	1	1	CT					rA					rB					0	1	1	1	1	0	0	1	1	0	/	X	E.CL
isel	0	1	1	1	1	1	rD					rA					rB					crb					0	1	1	1	1	0	A	
isync	0	1	0	0	1	1	///															0	0	1	0	0	1	0	1	1	0	/	XL	
lbarx	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	1	1	0	1	0	0	/	X	ER
lbdx	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	0	1	1	/	X	DS
lbepx	0	1	1	1	1	1	rD					rA					rB					0	0	0	1	0	1	1	1	1	1	/	X	E.PD
lbz	1	0	0	0	1	0	rD					rA					D																D	
lbzu	1	0	0	0	1	1	rD					rA					D																D	
lbzux	0	1	1	1	1	1	rD					rA					rB					0	0	0	1	1	1	0	1	1	1	/	X	
lbzx	0	1	1	1	1	1	rD					rA					rB					0	0	0	1	0	1	0	1	1	1	/	X	
ld	1	1	1	0	1	0	rD					rA					DS														0	0	DS	64
ldarx	0	1	1	1	1	1	rD					rA					rB					0	0	0	1	0	1	0	1	0	0	/	X	64
ldbrx	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	0	0	/	X	64
lddx	0	1	1	1	1	1	rD					rA					rB					1	0	0	1	1	0	0	0	1	1	/	X	DS, 64
ldepx	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	1	1	1	0	1	/	X	E.PD, 64
ldu	1	1	1	0	1	0	rD					rA					DS														0	1	DS	64
ldux	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	1	1	0	1	0	1	/	X	64
ldx	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	1	0	1	0	1	/	X	64
lfd	1	1	0	0	1	0	frD					rA					D																D	FP
lfddx	0	1	1	1	1	1	frD					rA					rB					1	1	0	0	1	0	0	0	1	1	/	X	DS, FP
lfdepx	0	1	1	1	1	1	frD					rA					rB					1	0	0	1	0	1	1	1	1	1	/	X	E.PD, FP
lfdu	1	1	0	0	1	1	frD					rA					D																D	FP
lfdux	0	1	1	1	1	1	frD					rA					rB					1	0	0	1	1	1	0	1	1	1	/	X	FP
lfdx	0	1	1	1	1	1	frD					rA					rB					1	0	0	1	0	1	0	1	1	1	/	X	FP
lfs	1	1	0	0	0	0	frD					rA					D																D	FP
lfsu	1	1	0	0	0	1	frD					rA					D																D	FP
lfsux	0	1	1	1	1	1	frD					rA					rB					1	0	0	0	1	1	0	1	1	1	/	X	FP
lfsx	0	1	1	1	1	1	frD					rA					rB					1	0	0	0	0	1	0	1	1	1	/	X	
lha	1	0	1	0	1	0	rD					rA					D																D	

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
lharx	0	1	1	1	1	1	rD	rA	rB	0	0	0	1	1	1	0	1	0	0	/	X	**ER**
lhau	1	0	1	0	1	1	rD	rA	D												D	
lhaux	0	1	1	1	1	1	rD	rA	rB	0	1	0	1	1	1	0	1	1	1	/	X	
lhax	0	1	1	1	1	1	rD	rA	rB	0	1	0	1	0	1	0	1	1	1	/	X	
lhbrx	0	1	1	1	1	1	rD	rA	rB	1	1	0	0	0	1	0	1	1	0	/	X	
lhdx	0	1	1	1	1	1	rD	rA	rB	1	0	0	0	1	0	0	0	1	1	/	X	**DS**
lhepx	0	1	1	1	1	1	rD	rA	rB	0	1	0	0	0	1	1	1	1	1	/	X	**E.PD**
lhz	1	0	1	0	0	0	rD	rA	D												D	
lhzu	1	0	1	0	0	1	rD	rA	D												D	
lhzux	0	1	1	1	1	1	rD	rA	rB	0	1	0	0	1	1	0	1	1	1	/	X	
lhzx	0	1	1	1	1	1	rD	rA	rB	0	1	0	0	0	1	0	1	1	1	/		
lmw	1	0	1	1	1	0	rD	rA	D												D	
lvebx	0	1	1	1	1	1	vD	rA	rB	0	0	0	0	0	0	0	1	1	1	/	X	**V**
lvehx	0	1	1	1	1	1	vD	rA	rB	0	0	0	0	1	0	0	1	1	1	/	X	**V**
lvepx	0	1	1	1	1	1	vD	rA	rB	0	1	0	0	1	0	0	1	1	1	/	X	**E.PD, V**
lvepxl	0	1	1	1	1	1	vD	rA	rB	0	1	0	0	0	0	0	1	1	1	/	X	**E.PD, V**
lvewx	0	1	1	1	1	1	vD	rA	rB	0	0	0	1	0	0	0	1	1	1	/	X	**V**
lvsl	0	1	1	1	1	1	vD	rA	rB	0	0	0	0	0	0	0	1	1	0	/	X	**V**
lvsr	0	1	1	1	1	1	vD	rA	rB	0	0	0	0	1	0	0	1	1	0	/	X	**V**
lvx	0	1	1	1	1	1	vD	rA	rB	0	0	0	1	1	0	0	1	1	1	/	X	**V**
lvxl	0	1	1	1	1	1	vD	rA	rB	0	1	0	1	1	0	0	1	1	1	/	X	**V**
lwa	1	1	1	0	1	0	rD	rA	DS										1	0	DS	**64**
lwarx	0	1	1	1	1	1	rD	rA	rB	0	0	0	0	0	1	0	1	0	0	/	X	
lwaux	0	1	1	1	1	1	rD	rA	rB	0	1	0	1	1	1	0	1	0	1	/	X	**64**
lwax	0	1	1	1	1	1	rD	rA	rB	0	1	0	1	0	1	0	1	0	1	/	X	**64**
lwbrx	0	1	1	1	1	1	rD	rA	rB	1	0	0	0	0	1	0	1	1	0	/	X	
lwdx	0	1	1	1	1	1	rD	rA	rB	1	0	0	1	0	0	0	0	1	1	/	X	**DS**
lwepx	0	1	1	1	1	1	rD	rA	rB	0	0	0	0	0	1	1	1	1	1	/	X	**E.PD**
lwz	1	0	0	0	0	0	rD	rA	D												D	
lwzu	1	0	0	0	0	1	rD	rA	D												D	
lwzux	0	1	1	1	1	1	rD	rA	rB	0	0	0	0	1	1	0	1	1	1	/	X	
lwzx	0	1	1	1	1	1	rD	rA	rB	0	0	0	0	0	1	0	1	1	1	/	X	
mbar	0	1	1	1	1	1	MO	///		1	1	0	1	0	1	0	1	1	0	/	X	**Embedded**
mcrf	0	1	0	0	1	1	crD //	crfS	///	0	0	0	0	0	0	0	0	0	0	/	XL	
mcrfs	1	1	1	1	1	1	crD //	crfS	///	0	0	0	1	0	0	0	0	0	0	/	X	**FP**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
mcrxr	0	1	1	1	1	1	crD			///												1	0	0	0	0	0	0	0	0	0	/	X	
mfcr	0	1	1	1	1	1	rD					0	///									0	0	0	0	0	1	0	0	1	1	/	X	
mfdcr	0	1	1	1	1	1	rD					DCRN5–9					DCRN0–4					0	1	0	1	0	0	0	0	1	1	/	XFX	**E.DC**
mffs	1	1	1	1	1	1	frD					///										1	0	0	1	0	0	0	1	1	1	0	X	**FP**
mffs.	1	1	1	1	1	1	frD					///										1	0	0	1	0	0	0	1	1	1	1	X	**FP.R**
mfmsr	0	1	1	1	1	1	rD					///										0	0	0	1	0	1	0	0	1	1	/	X	
mfocrf	0	1	1	1	1	1	rD					1	CRM								/	0	0	0	0	0	1	0	0	1	1	/	X	
mfpmr	0	1	1	1	1	1	rD					PMRN5–9					PMRN0–4					0	1	0	1	0	0	1	1	1	0	/	XFX	**E.PM**
mfspr	0	1	1	1	1	1	rD					SPRN[5–9]					SPRN[0–4]					0	1	0	1	0	1	0	0	1	1	/	XFX	
mftb	0	1	1	1	1	1	rD					TBRN[5–9]					TBRN[0–4]					0	1	0	1	1	1	0	0	1	1	0	XFX	
mfvscr	0	0	0	1	0	0	vD					///					///					1	1	0	0	0	0	0	1	0	0	0	VX	**V**
msgclr	0	1	1	1	1	1	///					///					rB					0	0	1	1	1	0	1	1	1	0	/	X	**E.PC**
msgsnd	0	1	1	1	1	1	///					///					rB					0	0	1	1	0	0	1	1	1	0	/	X	**E.PC**
mtcrf	0	1	1	1	1	1	rS					0	CRM								/	0	0	1	0	0	1	0	0	0	0	/	XFX	
mtdcr	0	1	1	1	1	1	rS					DCRN5–9					DCRN0–4					0	1	1	1	0	0	0	0	1	1	/	XFX	**E.DC**
mtfsb0	1	1	1	1	1	1	crbD					///										0	0	0	1	0	0	0	1	1	0	0	X	**FP**
mtfsb0.	1	1	1	1	1	1	crbD					///										0	0	0	1	0	0	0	1	1	0	1	X	**FP.R**
mtfsb1	1	1	1	1	1	1	crbD					///										0	0	0	0	1	0	0	1	1	0	0	X	**FP**
mtfsb1.	1	1	1	1	1	1	crbD					///										0	0	0	0	1	0	0	1	1	0	1	X	**FP.R**
mtfsf	1	1	1	1	1	1	L	FM								W	frB					1	0	1	1	0	0	0	1	1	1	0	XFX	**FP**
mtfsf.	1	1	1	1	1	1	L	FM								W	frB					1	0	1	1	0	0	0	1	1	1	1	XFX	**FP.R**
mtfsfi	1	1	1	1	1	1	crD			///		///				W	IMM				/	0	0	1	0	0	0	0	1	1	0	0	X	**FP**
mtfsfi.	1	1	1	1	1	1	crD			///		///				W	IMM				/	0	0	1	0	0	0	0	1	1	0	1	X	**FP.R**
mtmsr	0	1	1	1	1	1	rS					///										0	0	1	0	0	1	0	0	1	0	/	X	
mtocrf	0	1	1	1	1	1	rS					1	CRM								/	0	0	1	0	0	1	0	0	0	0	/	XFX	
mtpmr	0	1	1	1	1	1	rS					PMRN5–9					PMRN0–4					0	1	1	1	0	0	1	1	1	0	/	XFX	**E.PM**
mtspr	0	1	1	1	1	1	rS					SPRN[5–9]					SPRN[0–4]					0	1	1	1	0	1	0	0	1	1	/	XFX	
mtvscr	0	1	1	1	1	1	///					///					vB					1	1	0	0	1	0	0	0	1	0	0	VX	**V**
mulhd	0	1	1	1	1	1	rD					rA					rB					/	0	0	1	0	0	1	0	0	1	0	X	**64**
mulhd.	0	1	1	1	1	1	rD					rA					rB					/	0	0	1	0	0	1	0	0	1	1	X	**64**
mulhdu	0	1	1	1	1	1	rD					rA					rB					/	0	0	0	0	0	1	0	0	1	0	X	**64**
mulhdu.	0	1	1	1	1	1	rD					rA					rB					/	0	0	0	0	0	1	0	0	1	1	X	**64**
mulhw	0	1	1	1	1	1	rD					rA					rB					/	0	0	1	0	0	1	0	1	1	0	X	
mulhw.	0	1	1	1	1	1	rD					rA					rB					/	0	0	1	0	0	1	0	1	1	1	X	
mulhwu	0	1	1	1	1	1	rD					rA					rB					/	0	0	0	0	0	1	0	1	1	0	X	

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
mulhwu.	0	1	1	1	1	1	rD					rA					rB					/	0	0	0	0	0	1	0	1	1	1	X	
mulld	0	1	1	1	1	1	rD					rA					rB					0	0	1	1	1	0	1	0	0	1	0	X	**64**
mulld.	0	1	1	1	1	1	rD					rA					rB					0	0	1	1	1	0	1	0	0	1	1	X	**64**
mulldo	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	0	1	0	X	**64**
mulldo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	0	1	1	X	**64**
mulli	0	0	0	1	1	1	rD					rA					SIMM																D	
mullw	0	1	1	1	1	1	rD					rA					rB					0	0	1	1	1	0	1	0	1	1	0	X	
mullw.	0	1	1	1	1	1	rD					rA					rB					0	0	1	1	1	0	1	0	1	1	1	X	
mullwo	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	1	1	0	X	
mullwo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	1	1	1	X	
nand	0	1	1	1	1	1	rS					rA					rB					0	1	1	1	0	1	1	1	0	0	0	X	
nand.	0	1	1	1	1	1	rS					rA					rB					0	1	1	1	0	1	1	1	0	0	1	X	
neg	0	1	1	1	1	1	rD					rA					///					0	0	0	1	1	0	1	0	0	0	0	X	
neg.	0	1	1	1	1	1	rD					rA					///					0	0	0	1	1	0	1	0	0	0	1	X	
nego	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	1	0	0	0	0	X	
nego.	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	1	0	0	0	1	X	
nor	0	1	1	1	1	1	rS					rA					rB					0	0	0	1	1	1	1	1	0	0	0	X	
nor.	0	1	1	1	1	1	rS					rA					rB					0	0	0	1	1	1	1	1	0	0	1	X	
or	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	1	1	1	1	0	0	0	X	
or.	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	1	1	1	1	0	0	1	X	
orc	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	1	0	0	0	X	
orc.	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	1	0	0	1	X	
ori	0	1	1	0	0	0	rS					rA					UIMM																D	
oris	0	1	1	0	0	1	rS					rA					UIMM																D	
popcntb	0	1	1	1	1	1	rS					rA					///					0	0	0	1	1	1	1	0	1	0	/	X	
popcntd	0	1	1	1	1	1	rS					rA					///					0	1	1	1	1	1	1	0	1	0	/	X	**64**
popcntw	0	1	1	1	1	1	rS					rA					///					0	1	0	1	1	1	1	0	1	0	/	X	
prtyd	0	1	1	1	1	1	rS					rA					///					0	0	1	0	1	1	1	0	1	0	/	X	**64**
prtyw	0	1	1	1	1	1	rS					rA					///					0	0	1	0	0	1	1	0	1	0	/	X	
rfci	0	1	0	0	1	1	///															0	0	0	0	1	1	0	0	1	1	/	XL	**Embedded**
rfdi	0	1	0	0	1	1	///															0	0	0	0	1	0	0	1	1	1	/	X	**E.ED**
rfgi	0	1	0	0	1	1	///															0	0	0	0	1	1	0	0	1	0	/	X	**E.HV**
rfi	0	1	0	0	1	1	///															0	0	0	0	1	1	0	0	1	0	/	XL	**Embedded**
rfmci	0	1	0	0	1	1	///															0	0	0	0	1	0	0	1	1	0	/	XL	**Embedded**
rldcl	0	1	1	1	1	0	rS					rA					rB					mb1–5					mb0	1	0	0	0	0	MDS	**64**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
rldcl.	0	1	1	1	1	0	rS					rA					rB					mb1–5					mb0	1	0	0	0	1	MDS	**64**
rldcr	0	1	1	1	1	0	rS					rA					rB					me1–5					me0	1	0	0	1	0	MDS	**64**
rldcr.	0	1	1	1	1	0	rS					rA					rB					me1–5					me0	1	0	0	1	1	MDS	**64**
rldic	0	1	1	1	1	0	rS					rA					sh1–5					mb1–5					mb0	0	1	0	sh0	0	MD	**64**
rldic.	0	1	1	1	1	0	rS					rA					sh1–5					mb1–5					mb0	0	1	0	sh0	1	MD	**64**
rldicl	0	1	1	1	1	0	rS					rA					sh1–5					mb1–5					mb0	0	0	0	sh0	0	MD	**64**
rldicl.	0	1	1	1	1	0	rS					rA					sh1–5					mb1–5					mb0	0	0	0	sh0	1	MD	**64**
rldicr	0	1	1	1	1	0	rS					rA					sh1–5					me1–5					me0	0	0	1	sh0	0	MD	**64**
rldicr.	0	1	1	1	1	0	rS					rA					sh1–5					me1–5					me0	0	0	1	sh0	1	MD	**64**
rldimi	0	1	1	1	1	0	rS					rA					sh1–5					mb1–5					mb0	0	1	1	sh0	0	MD	**64**
rldimi.	0	1	1	1	1	0	rS					rA					sh1–5					mb1–5					mb0	0	1	1	sh0	1	MD	**64**
rlwimi	0	1	0	1	0	0	rS					rA					SH					MB					ME					0	M	
rlwimi.	0	1	0	1	0	0	rS					rA					SH					MB					ME					1	M	
rlwinm	0	1	0	1	0	1	rS					rA					SH					MB					ME					0	M	
rlwinm.	0	1	0	1	0	1	rS					rA					SH					MB					ME					1	M	
rlwnm	0	1	0	1	1	1	rS					rA					rB					MB					ME					0	M	
rlwnm.	0	1	0	1	1	1	rS					rA					rB					MB					ME					1	M	
sc	0	1	0	0	0	1	///														LEV							///			1	/	SC	
sld	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	0	1	1	0	X	**64**
sld.	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	0	1	1	1	X	**64**
slw	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	0	0	0	0	X	
slw.	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	0	0	0	1	X	
srad	0	1	1	1	1	1	rS					rA					rB					1	1	0	0	0	1	1	0	1	0	0	X	**64**
srad.	0	1	1	1	1	1	rS					rA					rB					1	1	0	0	0	1	1	0	1	0	1	X	**64**
sradi	0	1	1	1	1	1	rS					rA					sh1–5					1	1	0	0	1	1	1	0	1	sh0	0	XS	**64**
sradi.	0	1	1	1	1	1	rS					rA					sh1–5					1	1	0	0	1	1	1	0	1	sh0	1	XS	**64**
sraw	0	1	1	1	1	1	rS					rA					rB					1	1	0	0	0	1	1	0	0	0	0	X	
sraw.	0	1	1	1	1	1	rS					rA					rB					1	1	0	0	0	1	1	0	0	0	1	X	
srawi	0	1	1	1	1	1	rS					rA					SH					1	1	0	0	1	1	1	0	0	0	0	X	
srawi.	0	1	1	1	1	1	rS					rA					SH					1	1	0	0	1	1	1	0	0	0	1	X	
srd	0	1	1	1	1	1	rS					rA					rB					1	0	0	0	0	1	1	0	1	1	0	X	**64**
srd.	0	1	1	1	1	1	rS					rA					rB					1	0	0	0	0	1	1	0	1	1	1	X	**64**
srw	0	1	1	1	1	1	rS					rA					rB					1	0	0	0	0	1	1	0	0	0	0	X	
srw.	0	1	1	1	1	1	rS					rA					rB					1	0	0	0	0	1	1	0	0	0	1	X	

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
stb	1	0	0	1	1	0	rS	rA	D												D	
stbcx.	0	1	1	1	1	1	rS	rA	rB	1	0	1	0	1	1	0	1	1	0	1	X	**ER**
stbdx	0	1	1	1	1	1	rS	rA	rB	1	0	1	0	0	0	0	0	1	1	/	X	**DS**
stbepx	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	0	1	1	1	1	1	/	X	**E.PD**
stbu	1	0	0	1	1	1	rS	rA	D												D	
stbux	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	1	1	0	1	1	1	/	X	
stbx	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	0	1	0	1	1	1	/	X	
std	1	1	1	1	1	0	rS	rA	DS										0	0	DS	**64**
stdbrx	0	1	1	1	1	1	rS	rA	rB	1	0	1	0	0	1	0	1	0	0	/	X	**64**
stdcx.	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	0	1	0	1	1	0	1	X	**64**
stddx	0	1	1	1	1	1	rS	rA	rB	1	0	1	1	1	0	0	0	1	1	/	X	**DS, 64**
stdepx	0	1	1	1	1	1	rS	rA	rB	0	0	1	0	0	1	1	1	0	1	/	X	**E.PD, 64**
stdu	1	1	1	1	1	0	rS	rA	DS										0	1	DS	**64**
stdux	0	1	1	1	1	1	rS	rA	rB	0	0	1	0	1	1	0	1	0	1	/	X	**64**
stdx	0	1	1	1	1	1	rS	rA	rB	0	0	1	0	0	1	0	1	0	1	/	X	**64**
stfd	1	1	0	1	1	0	frS	rA	D												D	**FP**
stfddx	0	1	1	1	1	1	frS	rA	rB	1	1	1	0	1	0	0	0	1	1	/	X	**DS, FP**
stfdepx	0	1	1	1	1	1	frS	rA	rB	1	0	1	1	0	1	1	1	1	1	/	X	**E.PD, FP**
stfdu	1	1	0	1	1	1	frS	rA	D												D	**FP**
stfdux	0	1	1	1	1	1	frS	rA	rB	1	0	1	1	1	1	0	1	1	1	/	X	**FP**
stfdx	0	1	1	1	1	1	frS	rA	rB	1	0	1	1	0	1	0	1	1	1	/	X	**FP**
stfiwx	0	1	1	1	1	1	frS	rA	rB	1	1	1	1	0	1	0	1	1	1	/	X	**FP**
stfs	1	1	0	1	0	0	frS	rA	D												D	**FP**
stfsu	1	1	0	1	0	1	frS	rA	D												D	**FP**
stfsux	0	1	1	1	1	1	frS	rA	rB	1	0	1	0	1	1	0	1	1	1	/	X	**FP**
stfsx	0	1	1	1	1	1	frS	rA	rB	1	0	1	0	0	1	0	1	1	1	/	X	**FP**
sth	1	0	1	1	0	0	rS	rA	D												D	
sthbrx	0	1	1	1	1	1	rS	rA	rB	1	1	1	0	0	1	0	1	1	0	/	X	
sthcx.	0	1	1	1	1	1	rS	rA	rB	1	0	1	1	0	1	0	1	1	0	1	X	**ER**
sthdx	0	1	1	1	1	1	rS	rA	rB	1	0	1	0	1	0	0	0	1	1	/	X	**DS**
sthepx	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	0	1	1	1	1	1	/	X	**E.PD**
sthu	1	0	1	1	0	1	rS	rA	D												D	
sthux	0	1	1	1	1	1	rS	rA	rB	0	1	1	0	1	1	0	1	1	1	/	X	
sthx	0	1	1	1	1	1	rS	rA	rB	0	1	1	0	0	1	0	1	1	1	/	X	
stmw	1	0	1	1	1	1	rS	rA	D												D	

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
stvebx	0	1	1	1	1	1	vS					rA					rB					0	0	1	0	0	0	0	1	1	1	/	X	V
stvehx	0	1	1	1	1	1	vS					rA					rB					0	0	1	0	1	0	0	1	1	1	/	X	V
stvepx	0	1	1	1	1	1	vS					rA					rB					1	1	0	0	1	0	0	1	1	1	/	X	E.PD, V
stvepxl	0	1	1	1	1	1	vS					rA					rB					1	1	0	0	0	0	0	1	1	1	/	X	E.PD, V
stvewx	0	1	1	1	1	1	vS					rA					rB					0	0	1	1	0	0	0	1	1	1	/	X	V
stvx	0	1	1	1	1	1	vS					rA					rB					0	0	1	1	1	0	0	1	1	1	/	X	V
stvxl	0	1	1	1	1	1	vS					rA					rB					0	1	1	1	1	0	0	1	1	1	/	X	V
stw	1	0	0	1	0	0	rS					rA					D																D	
stwbrx	0	1	1	1	1	1	rS					rA					rB					1	0	1	0	0	1	0	1	1	0	/	X	
stwcx.	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	0	1	0	1	1	0	1	X	
stwdx	0	1	1	1	1	1	rS					rA					rB					1	0	1	1	0	0	0	0	1	1	/	X	DS
stwepx	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	0	1	1	1	1	1	/	X	E.PD
stwu	1	0	0	1	0	1	rS					rA					D																D	
stwux	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	1	1	0	1	1	1	/	D	
stwx	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	0	1	0	1	1	1	/	D	
subf	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	1	0	1	0	0	0	0	X	
subf.	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	1	0	1	0	0	0	1	X	
subfc	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	0	0	0	X	
subfc.	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	0	0	1	X	
subfco	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	0	0	0	X	
subfco.	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	0	0	1	X	
subfe	0	1	1	1	1	1	rD					rA					rB					0	0	1	0	0	0	1	0	0	0	0	X	
subfe.	0	1	1	1	1	1	rD					rA					rB					0	0	1	0	0	0	1	0	0	0	1	X	
subfeo	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	0	0	0	X	
subfeo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	0	0	1	X	
subfic	0	0	1	0	0	0	rD					rA					SIMM																D	
subfme	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	0	0	0	X	
subfme.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	0	0	1	X	
subfmeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	0	0	0	X	
subfmeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	0	0	1	X	
subfo	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	0	0	0	0	X	
subfo.	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	0	0	0	1	X	
subfze	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	0	0	0	X	
subfze.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	0	0	1	X	
subfzeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	0	0	0	X	

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
subfzeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	0	0	1	X	
sync	0	1	1	1	1	1	///			L		/	////				///					1	0	0	1	0	1	0	1	1	0	/	X	
td	0	1	1	1	1	1	TO					rA					rB					0	0	0	1	0	0	0	1	0	0	/	X	**64**
tdi	0	0	0	0	1	0	TO					rA					SIMM																D	**64**
tlbilx	0	1	1	1	1	1	0	///		T		rA					rB					0	0	0	0	0	1	0	0	1	0	/	X	**Embedded**
tlbivax	0	1	1	1	1	1	0	///				rA					rB					1	1	0	0	0	1	0	0	1	0	/	X	**Embedded**
tlbre	0	1	1	1	1	1	0	///														1	1	1	0	1	1	0	0	1	0	/	X	**Embedded**
tlbsx	0	1	1	1	1	1	0	///				rA					rB					1	1	1	0	0	1	0	0	1	0	/	X	**Embedded**
tlbsync	0	1	1	1	1	1	0	///														1	0	0	0	1	1	0	1	1	0	/	X	**Embedded**
tlbwe	0	1	1	1	1	1	0	///														1	1	1	1	0	1	0	0	1	0	/	X	**Embedded**
tw	0	1	1	1	1	1	TO					rA					rB					0	0	0	0	0	0	0	1	0	0	/	X	
twi	0	0	0	0	1	1	TO					rA					SIMM																D	
vaddcuw	0	0	0	1	0	0	vD					vA					vB					0	0	1	1	0	0	0	0	0	0	0	VX	**V**
vaddfp	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	0	0	0	1	0	1	0	VX	**V**
vaddsbs	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	0	0	0	0	0	0	0	VX	**V**
vaddshs	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	1	0	0	0	0	0	0	VX	**V**
vaddsws	0	0	0	1	0	0	vD					vA					vB					0	1	1	1	0	0	0	0	0	0	0	VX	**V**
vaddubm	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	0	0	0	0	0	0	0	VX	**V**
vaddubs	0	0	0	1	0	0	vD					vA					vB					0	1	0	0	0	0	0	0	0	0	0	VX	**V**
vadduhm	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	1	0	0	0	0	0	0	VX	**V**
vadduhs	0	0	0	1	0	0	vD					vA					vB					0	1	0	0	1	0	0	0	0	0	0	VX	**V**
vadduwm	0	0	0	1	0	0	vD					vA					vB					0	0	0	1	0	0	0	0	0	0	0	VX	**V**
vadduws	0	0	0	1	0	0	vD					vA					vB					0	1	0	1	0	0	0	0	0	0	0	VX	**V**
vand	0	0	0	1	0	0	vD					vA					vB					1	0	0	0	0	0	0	0	1	0	0	VX	**V**
vandc	0	0	0	1	0	0	vD					vA					vB					1	0	0	0	1	0	0	0	1	0	0	VX	**V**
vavgsb	0	0	0	1	0	0	vD					vA					vB					1	0	1	0	0	0	0	0	0	1	0	VX	**V**
vavgsh	0	0	0	1	0	0	vD					vA					vB					1	0	1	0	1	0	0	0	0	1	0	VX	**V**
vavgsw	0	0	0	1	0	0	vD					vA					vB					1	0	1	1	0	0	0	0	0	1	0	VX	**V**
vavgub	0	0	0	1	0	0	vD					vA					vB					1	0	0	0	0	0	0	0	0	1	0	VX	**V**
vavguh	0	0	0	1	0	0	vD					vA					vB					1	0	0	0	1	0	0	0	0	1	0	VX	**V**
vavguw	0	0	0	1	0	0	vD					vA					vB					1	0	0	1	0	0	0	0	0	1	0	VX	**V**
vcfsx	0	0	0	1	0	0	vD					UIMM					vB					0	1	1	0	1	0	0	1	0	1	0	VX	**V**
vcfux	0	0	0	1	0	0	vD					UIMM					vB					0	1	1	0	0	0	0	1	0	1	0	VX	**V**
vcmpbfp	0	0	0	1	0	0	vD					vA					vB					0	1	1	1	1	0	0	0	1	1	0	VC	**V**
vcmpbfp.	0	0	0	1	0	0	vD					vA					vB					1	1	1	1	1	0	0	0	1	1	0	VC	**V**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
vcmpeqfp	0	0	0	1	0	0	vD					vA					vB					0	0	0	1	1	0	0	0	1	1	0	VC	**V**
vcmpeqfp.	0	0	0	1	0	0	vD					vA					vB					1	0	0	1	1	0	0	0	1	1	0	VC	**V**
vcmpequb	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	0	0	0	0	1	1	0	VC	**V**
vcmpequb.	0	0	0	1	0	0	vD					vA					vB					1	0	0	0	0	0	0	0	1	1	0	VC	**V**
vcmpequh	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	1	0	0	0	1	1	0	VC	**V**
vcmpequh.	0	0	0	1	0	0	vD					vA					vB					1	0	0	0	1	0	0	0	1	1	0	VC	**V**
vcmpequw	0	0	0	1	0	0	vD					vA					vB					0	0	0	1	0	0	0	0	1	1	0	VC	**V**
vcmpequw.	0	0	0	1	0	0	vD					vA					vB					1	0	0	1	0	0	0	0	1	1	0	VC	**V**
vcmpgefp	0	0	0	1	0	0	vD					vA					vB					0	0	1	1	1	0	0	0	1	1	0	VC	**V**
vcmpgefp.	0	0	0	1	0	0	vD					vA					vB					1	0	1	1	1	0	0	0	1	1	0	VC	**V**
vcmpgtfp	0	0	0	1	0	0	vD					vA					vB					0	1	0	1	1	0	0	0	1	1	0	VC	**V**
vcmpgtfp.	0	0	0	1	0	0	vD					vA					vB					1	1	0	1	1	0	0	0	1	1	0	VC	**V**
vcmpgtsb	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	0	0	0	0	1	1	0	VC	**V**
vcmpgtsb.	0	0	0	1	0	0	vD					vA					vB					1	1	1	0	0	0	0	0	1	1	0	VC	**V**
vcmpgtsh	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	1	0	0	0	1	1	0	VC	**V**
vcmpgtsh.	0	0	0	1	0	0	vD					vA					vB					1	1	1	0	1	0	0	0	1	1	0	VC	**V**
vcmpgtsw	0	0	0	1	0	0	vD					vA					vB					0	1	1	1	0	0	0	0	1	1	0	VC	**V**
vcmpgtsw.	0	0	0	1	0	0	vD					vA					vB					1	1	1	1	0	0	0	0	1	1	0	VC	**V**
vcmpgtub	0	0	0	1	0	0	vD					vA					vB					0	1	0	0	0	0	0	0	1	1	0	VC	**V**
vcmpgtub.	0	0	0	1	0	0	vD					vA					vB					1	1	0	0	0	0	0	0	1	1	0	VC	**V**
vcmpgtuh	0	0	0	1	0	0	vD					vA					vB					0	1	0	0	1	0	0	0	1	1	0	VC	**V**
vcmpgtuh.	0	0	0	1	0	0	vD					vA					vB					1	1	0	0	1	0	0	0	1	1	0	VC	**V**
vcmpgtuw	0	0	0	1	0	0	vD					vA					vB					0	1	0	1	0	0	0	0	1	1	0	VC	**V**
vcmpgtuw.	0	0	0	1	0	0	vD					vA					vB					1	1	0	1	0	0	0	0	1	1	0	VC	**V**
vctsxs	0	0	0	1	0	0	vD					UIMM					vB					0	1	1	1	1	0	0	1	0	1	0	VX	**V**
vctuxs	0	0	0	1	0	0	vD					UIMM					vB					0	1	1	1	0	0	0	1	0	1	0	VX	**V**
vexptefp	0	0	0	1	0	0	vD					///					vB					0	0	1	1	0	0	0	1	0	1	0	VX	**V**
vlogefp	0	0	0	1	0	0	vD					///					vB					0	0	1	1	1	0	0	1	0	1	0	VX	**V**
vmaddfp	0	0	0	1	0	0	vD					vA					vB					vC					1	0	1	1	1	0	VA	**V**
vmaxfp	0	0	0	1	0	0	vD					vA					vB					1	0	0	0	0	0	0	1	0	1	0	VX	**V**
vmaxsb	0	0	0	1	0	0	vD					vA					vB					0	0	1	0	0	0	0	0	0	1	0	VX	**V**
vmaxsh	0	0	0	1	0	0	vD					vA					vB					0	0	1	0	1	0	0	0	0	1	0	VX	**V**
vmaxsw	0	0	0	1	0	0	vD					vA					vB					0	0	1	1	0	0	0	0	0	1	0	VX	**V**
vmaxub	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	0	0	0	0	0	1	0	VX	**V**
vmaxuh	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	1	0	0	0	0	1	0	VX	**V**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
vmaxuw	0	0	0	1	0	0	vD					vA					vB					0	0	0	1	0	0	0	0	0	1	0	VX	**V**
vmhaddshs	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	0	0	0	VA	**V**
vmhraddshs	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	0	0	1	VA	**V**
vminfp	0	0	0	1	0	0	vD					vA					vB					1	0	0	0	1	0	0	1	0	1	0	VX	**V**
vminsb	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	0	0	0	0	0	1	0	VX	**V**
vminsh	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	1	0	0	0	0	1	0	VX	**V**
vminsw	0	0	0	1	0	0	vD					vA					vB					0	1	1	1	0	0	0	0	0	1	0	VX	**V**
vminub	0	0	0	1	0	0	vD					vA					vB					0	1	0	0	0	0	0	0	0	1	0	VX	**V**
vminuh	0	0	0	1	0	0	vD					vA					vB					0	1	0	0	1	0	0	0	0	1	0	VX	**V**
vminuw	0	0	0	1	0	0	vD					vA					vB					0	1	0	1	0	0	0	0	0	1	0	VX	**V**
vmladduhm	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	0	1	0	VA	**V**
vmrghb	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	0	0	0	1	1	0	0	VX	**V**
vmrghh	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	1	0	0	1	1	0	0	VX	**V**
vmrghw	0	0	0	1	0	0	vD					vA					vB					0	0	0	1	0	0	0	1	1	0	0	VX	**V**
vmrglb	0	0	0	1	0	0	vD					vA					vB					0	0	1	0	0	0	0	1	1	0	0	VX	**V**
vmrglh	0	0	0	1	0	0	vD					vA					vB					0	0	1	0	1	0	0	1	1	0	0	VX	**V**
vmrglw	0	0	0	1	0	0	vD					vA					vB					0	0	1	1	0	0	0	1	1	0	0	VX	**V**
vmsummbm	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	1	0	1	VA	**V**
vmsumshm	0	0	0	1	0	0	vD					vA					vB					vC					1	0	1	0	0	0	VA	**V**
vmsumshs	0	0	0	1	0	0	vD					vA					vB					vC					1	0	1	0	0	1	VA	**V**
vmsumubm	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	1	0	0	VA	**V**
vmsumuhm	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	1	1	0	VA	**V**
vmsumuhs	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	1	1	1	VA	**V**
vmulesb	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	0	0	0	1	0	0	0	VX	**V**
vmulesh	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	1	0	0	1	0	0	0	VX	**V**
vmuleub	0	0	0	1	0	0	vD					vA					vB					0	1	0	0	0	0	0	1	0	0	0	VX	**V**
vmuleuh	0	0	0	1	0	0	vD					vA					vB					0	1	0	0	1	0	0	1	0	0	0	VX	**V**
vmulosb	0	0	0	1	0	0	vD					vA					vB					0	0	1	0	0	0	0	1	0	0	0	VX	**V**
vmulosh	0	0	0	1	0	0	vD					vA					vB					0	0	1	0	1	0	0	1	0	0	0	VX	**V**
vmuloub	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	0	0	0	1	0	0	0	VX	**V**
vmulouh	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	1	0	0	1	0	0	0	VX	**V**
vnmsubfp	0	0	0	1	0	0	vD					vA					vB					vC					1	0	1	1	1	1	VA	**V**
vnor	0	0	0	1	0	0	vD					vA					vB					1	0	1	0	0	0	0	0	1	0	0	VX	**V**
vor	0	0	0	1	0	0	vD					vA					vB					1	0	0	1	0	0	0	0	1	0	0	VX	**V**
vperm	0	0	0	1	0	0	vD					vA					vB					vC					1	0	1	0	1	1	VA	**V**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
vpkpx	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	0	0	0	1	1	1	0	VX	**V**
vpkshss	0	0	0	1	0	0	vD					vA					vB					0	0	1	1	0	0	0	1	1	1	0	VX	**V**
vpkshus	0	0	0	1	0	0	vD					vA					vB					0	0	1	0	0	0	0	1	1	1	0	VX	**V**
vpkswss	0	0	0	1	0	0	vD					vA					vB					0	0	1	1	1	0	0	1	1	1	0	VX	**V**
vpkswus	0	0	0	1	0	0	vD					vA					vB					0	0	1	0	1	0	0	1	1	1	0	VX	**V**
vpkuhum	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	0	0	0	1	1	1	0	VX	**V**
vpkuhus	0	0	0	1	0	0	vD					vA					vB					0	0	0	1	0	0	0	1	1	1	0	VX	**V**
vpkuwum	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	1	0	0	1	1	1	0	VX	**V**
vpkuwus	0	0	0	1	0	0	vD					vA					vB					0	0	0	1	1	0	0	1	1	1	0	VX	**V**
vrefp	0	0	0	1	0	0	vD					///					vB					0	0	1	0	0	0	0	1	0	1	0	VX	**V**
vrfim	0	0	0	1	0	0	vD					///					vB					0	1	0	1	1	0	0	1	0	1	0	VX	**V**
vrfin	0	0	0	1	0	0	vD					///					vB					0	1	0	0	0	0	0	1	0	1	0	VX	**V**
vrfip	0	0	0	1	0	0	vD					///					vB					0	1	0	1	0	0	0	1	0	1	0	VX	**V**
vrfiz	0	0	0	1	0	0	vD					///					vB					0	1	0	0	1	0	0	1	0	1	0	VX	**V**
vrlb	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	0	0	0	0	1	0	0	VX	**V**
vrlh	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	1	0	0	0	1	0	0	VX	**V**
vrlw	0	0	0	1	0	0	vD					vA					vB					0	0	0	1	0	0	0	0	1	0	0	VX	**V**
vrsqrtefp	0	0	0	1	0	0	vD					///					vB					0	0	1	0	1	0	0	1	0	1	0	VX	**V**
vsel	0	0	0	1	0	0	vD					vA					vB					vC					1	0	1	0	1	0	VA	**V**
vsl	0	0	0	1	0	0	vD					vA					vB					0	0	1	1	1	0	0	0	1	0	0	VX	**V**
vslb	0	0	0	1	0	0	vD					vA					vB					0	0	1	0	0	0	0	0	1	0	0	VX	**V**
vsldoi	0	0	0	1	0	0	vD					vA					vB					/	SH				1	0	1	1	0	0	VX	**V**
vslh	0	0	0	1	0	0	vD					vA					vB					0	0	1	0	1	0	0	0	1	0	0	VX	**V**
vslo	0	0	0	1	0	0	vD					vA					vB					1	0	0	0	0	0	0	1	1	0	0	VX	**V**
vslw	0	0	0	1	0	0	vD					vA					vB					0	0	1	1	0	0	0	0	1	0	0	VX	**V**
vspltb	0	0	0	1	0	0	vD					UIMM					vB					0	1	0	0	0	0	0	1	1	0	0	VX	**V**
vsplth	0	0	0	1	0	0	vD					UIMM					vB					0	1	0	0	1	0	0	1	1	0	0	VX	**V**
vspltisb	0	0	0	1	0	0	vD					SIMM					///					0	1	1	0	0	0	0	1	1	0	0	VX	**V**
vspltish	0	0	0	1	0	0	vD					SIMM					///					0	1	1	0	1	0	0	1	1	0	0	VX	**V**
vspltisw	0	0	0	1	0	0	vD					SIMM					///					0	1	1	1	0	0	0	1	1	0	0	VX	**V**
vspltw	0	0	0	1	0	0	vD					UIMM					vB					0	1	0	1	0	0	0	1	1	0	0	VX	**V**
vsr	0	0	0	1	0	0	vD					vA					vB					0	1	0	1	1	0	0	0	1	0	0	VX	**V**
vsrab	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	0	0	0	0	1	0	0	VX	**V**
vsrah	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	1	0	0	0	1	0	0	VX	**V**
vsraw	0	0	0	1	0	0	vD					vA					vB					0	1	1	1	0	0	0	0	1	0	0	VX	**V**

Table B-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
vsrb	0	0	0	1	0	0	vD					vA					vB					0	1	0	0	0	0	0	0	1	0	0	VX	**V**
vsrh	0	0	0	1	0	0	vD					vA					vB					0	1	0	0	1	0	0	0	1	0	0	VX	**V**
vsro	0	0	0	1	0	0	vD					vA					vB					1	0	0	0	1	0	0	1	1	0	0	VX	**V**
vsrw	0	0	0	1	0	0	vD					vA					vB					0	1	0	1	0	0	0	0	1	0	0	VX	**V**
vsubcuw	0	0	0	1	0	0	vD					vA					vB					1	0	1	1	0	0	0	0	0	0	0	VX	**V**
vsubfp	0	0	0	1	0	0	vD					vA					vB					0	0	0	0	1	0	0	1	0	1	0	VX	**V**
vsubsbs	0	0	0	1	0	0	vD					vA					vB					1	1	1	0	0	0	0	0	0	0	0	VX	**V**
vsubshs	0	0	0	1	0	0	vD					vA					vB					1	1	1	0	1	0	0	0	0	0	0	VX	**V**
vsubsws	0	0	0	1	0	0	vD					vA					vB					1	1	1	1	0	0	0	0	0	0	0	VX	**V**
vsububm	0	0	0	1	0	0	vD					vA					vB					1	0	0	0	0	0	0	0	0	0	0	VX	**V**
vsububs	0	0	0	1	0	0	vD					vA					vB					1	1	0	0	0	0	0	0	0	0	0	VX	**V**
vsubuhm	0	0	0	1	0	0	vD					vA					vB					1	0	0	0	1	0	0	0	0	0	0	VX	**V**
vsubuhs	0	0	0	1	0	0	vD					vA					vB					1	1	0	0	1	0	0	0	0	0	0	VX	**V**
vsubuwm	0	0	0	1	0	0	vD					vA					vB					1	0	0	1	0	0	0	0	0	0	0	VX	**V**
vsubuws	0	0	0	1	0	0	vD					vA					vB					1	1	0	1	0	0	0	0	0	0	0	VX	**V**
vsum2sws	0	0	0	1	0	0	vD					vA					vB					1	1	0	1	0	0	0	1	0	0	0	VX	**V**
vsum4sbs	0	0	0	1	0	0	vD					vA					vB					1	1	1	0	0	0	0	1	0	0	0	VX	**V**
vsum4shs	0	0	0	1	0	0	vD					vA					vB					1	1	0	0	1	0	0	1	0	0	0	VX	**V**
vsum4ubs	0	0	0	1	0	0	vD					vA					vB					1	1	0	0	0	0	0	1	0	0	0	VX	**V**
vsumsws	0	0	0	1	0	0	vD					vA					vB					1	1	1	1	0	0	0	1	0	0	0	VX	**V**
vupkhpx	0	0	0	1	0	0	vD					///					vB					0	1	1	0	1	0	0	1	1	1	0	VX	**V**
vupkhsb	0	0	0	1	0	0	vD					///					vB					0	1	0	0	0	0	0	1	1	1	0	VX	**V**
vupkhsh	0	0	0	1	0	0	vD					///					vB					0	1	0	0	1	0	0	1	1	1	0	VX	**V**
vupklpx	0	0	0	1	0	0	vD					///					vB					0	1	1	1	1	0	0	1	1	1	0	VX	**V**
vupklsb	0	0	0	1	0	0	vD					///					vB					0	1	0	1	0	0	0	1	1	1	0	VX	**V**
vupklsh	0	0	0	1	0	0	vD					///					vB					0	1	0	1	1	0	0	1	1	1	0	VX	**V**
vxor	0	0	0	1	0	0	vD					vA					vB					1	0	0	1	1	0	0	0	1	0	0	VX	**V**
wait	0	1	1	1	1	1	///			WC		WH	///									0	0	0	0	1	1	1	1	1	0	/	X	**WT**
wrtee	0	1	1	1	1	1	rS					///										0	0	1	0	0	0	0	0	1	1	/	X	**Embedded**
wrteei	0	1	1	1	1	1	///										E	///				0	0	1	0	1	0	0	0	1	1	/	X	**Embedded**
xor	0	1	1	1	1	1	rS					rA					rB					0	1	0	0	1	1	1	1	0	0	0	X	
xor.	0	1	1	1	1	1	rS					rA					rB					0	1	0	0	1	1	1	1	0	0	1	X	
xori	0	1	1	0	1	0	rS					rA					UIMM																D	
xoris	0	1	1	0	1	1	rS					rA					UIMM																D	

[1] d = UIMM * 8

[2] d = UIMM * 2

[3] d = UIMM * 4

B.4 Instructions Sorted by Opcode (Binary)

The following table lists instructions (non-VLE) by opcode, shown in binary.

Table B-4. Instructions Sorted by Opcode (Binary)

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
tdi	0	0	0	0	1	0	TO	rA	SIMM												D	**64**
twi	0	0	0	0	1	1	TO	rA	SIMM												D	
vaddubm	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	0	0	0	0	0	0	0	VX	**V**
vmaxub	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	0	0	0	0	0	1	0	VX	**V**
vrlb	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	0	0	0	0	1	0	0	VX	**V**
vcmpequb	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	0	0	0	0	1	1	0	VC	**V**
vmuloub	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	0	0	0	1	0	0	0	VX	**V**
vaddfp	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	0	0	0	1	0	1	0	VX	**V**
vmrghb	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	0	0	0	1	1	0	0	VX	**V**
vpkuhum	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	0	0	0	1	1	1	0	VX	**V**
vadduhm	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	1	0	0	0	0	0	0	VX	**V**
vmaxuh	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	1	0	0	0	0	1	0	VX	**V**
vrlh	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	1	0	0	0	1	0	0	VX	**V**
vcmpequh	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	1	0	0	0	1	1	0	VC	**V**
vmulouh	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	1	0	0	1	0	0	0	VX	**V**
vsubfp	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	1	0	0	1	0	1	0	VX	**V**
vmrghh	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	1	0	0	1	1	0	0	VX	**V**
vpkuwum	0	0	0	1	0	0	vD	vA	vB	0	0	0	0	1	0	0	1	1	1	0	VX	**V**
vadduwm	0	0	0	1	0	0	vD	vA	vB	0	0	0	1	0	0	0	0	0	0	0	VX	**V**
vmaxuw	0	0	0	1	0	0	vD	vA	vB	0	0	0	1	0	0	0	0	0	1	0	VX	**V**
vrlw	0	0	0	1	0	0	vD	vA	vB	0	0	0	1	0	0	0	0	1	0	0	VX	**V**
vcmpequw	0	0	0	1	0	0	vD	vA	vB	0	0	0	1	0	0	0	0	1	1	0	VC	**V**
vmrghw	0	0	0	1	0	0	vD	vA	vB	0	0	0	1	0	0	0	1	1	0	0	VX	**V**
vpkuhus	0	0	0	1	0	0	vD	vA	vB	0	0	0	1	0	0	0	1	1	1	0	VX	**V**
vcmpeqfp	0	0	0	1	0	0	vD	vA	vB	0	0	0	1	1	0	0	0	1	1	0	VC	**V**
vpkuwus	0	0	0	1	0	0	vD	vA	vB	0	0	0	1	1	0	0	1	1	1	0	VX	**V**
vmaxsb	0	0	0	1	0	0	vD	vA	vB	0	0	1	0	0	0	0	0	0	1	0	VX	**V**
vslb	0	0	0	1	0	0	vD	vA	vB	0	0	1	0	0	0	0	0	1	0	0	VX	**V**
vmulosb	0	0	0	1	0	0	vD	vA	vB	0	0	1	0	0	0	0	1	0	0	0	VX	**V**
vrefp	0	0	0	1	0	0	vD	///	vB	0	0	1	0	0	0	0	1	0	1	0	VX	**V**
vmrglb	0	0	0	1	0	0	vD	vA	vB	0	0	1	0	0	0	0	1	1	0	0	VX	**V**
vpkshus	0	0	0	1	0	0	vD	vA	vB	0	0	1	0	0	0	0	1	1	1	0	VX	**V**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
vmaxsh	0	0	0	1	0	0	vD	vA	vB	0	0	1	0	1	0	0	0	0	1	0	VX	**V**
vslh	0	0	0	1	0	0	vD	vA	vB	0	0	1	0	1	0	0	0	1	0	0	VX	**V**
vmulosh	0	0	0	1	0	0	vD	vA	vB	0	0	1	0	1	0	0	1	0	0	0	VX	**V**
vrsqrtefp	0	0	0	1	0	0	vD	///	vB	0	0	1	0	1	0	0	1	0	1	0	VX	**V**
vmrglh	0	0	0	1	0	0	vD	vA	vB	0	0	1	0	1	0	0	1	1	0	0	VX	**V**
vpkswus	0	0	0	1	0	0	vD	vA	vB	0	0	1	0	1	0	0	1	1	1	0	VX	**V**
vaddcuw	0	0	0	1	0	0	vD	vA	vB	0	0	1	1	0	0	0	0	0	0	0	VX	**V**
vmaxsw	0	0	0	1	0	0	vD	vA	vB	0	0	1	1	0	0	0	0	0	1	0	VX	**V**
vslw	0	0	0	1	0	0	vD	vA	vB	0	0	1	1	0	0	0	0	1	0	0	VX	**V**
vexptefp	0	0	0	1	0	0	vD	///	vB	0	0	1	1	0	0	0	1	0	1	0	VX	**V**
vmrglw	0	0	0	1	0	0	vD	vA	vB	0	0	1	1	0	0	0	1	1	0	0	VX	**V**
vpkshss	0	0	0	1	0	0	vD	vA	vB	0	0	1	1	0	0	0	1	1	1	0	VX	**V**
vsl	0	0	0	1	0	0	vD	vA	vB	0	0	1	1	1	0	0	0	1	0	0	VX	**V**
vcmpgefp	0	0	0	1	0	0	vD	vA	vB	0	0	1	1	1	0	0	0	1	1	0	VC	**V**
vlogefp	0	0	0	1	0	0	vD	///	vB	0	0	1	1	1	0	0	1	0	1	0	VX	**V**
vpkswss	0	0	0	1	0	0	vD	vA	vB	0	0	1	1	1	0	0	1	1	1	0	VX	**V**
evaddw	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	0	0	0	0	0	0	EVX	**SP**
vaddubs	0	0	0	1	0	0	vD	vA	vB	0	1	0	0	0	0	0	0	0	0	0	VX	**V**
evaddiw	0	0	0	1	0	0	rD	UIMM	rB	0	1	0	0	0	0	0	0	0	1	0	EVX	**SP**
vminub	0	0	0	1	0	0	vD	vA	vB	0	1	0	0	0	0	0	0	0	1	0	VX	**V**
evsubfw	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	0	0	0	1	0	0	EVX	**SP**
vsrb	0	0	0	1	0	0	vD	vA	vB	0	1	0	0	0	0	0	0	1	0	0	VX	**V**
evsubifw	0	0	0	1	0	0	rD	UIMM	rB	0	1	0	0	0	0	0	0	1	1	0	EVX	**SP**
vcmpgtub	0	0	0	1	0	0	vD	vA	vB	0	1	0	0	0	0	0	0	1	1	0	VC	**V**
evabs	0	0	0	1	0	0	rD	rA	///	0	1	0	0	0	0	0	1	0	0	0	EVX	**SP**
vmuleub	0	0	0	1	0	0	vD	vA	vB	0	1	0	0	0	0	0	1	0	0	0	VX	**V**
evneg	0	0	0	1	0	0	rD	rA	///	0	1	0	0	0	0	0	1	0	0	1	EVX	**SP**
evextsb	0	0	0	1	0	0	rD	rA	///	0	1	0	0	0	0	0	1	0	1	0	EVX	**SP**
vrfin	0	0	0	1	0	0	vD	///	vB	0	1	0	0	0	0	0	1	0	1	0	VX	**V**
evextsh	0	0	0	1	0	0	rD	rA	///	0	1	0	0	0	0	0	1	0	1	1	EVX	**SP**
evrndw	0	0	0	1	0	0	rD	rA	UIMM	0	1	0	0	0	0	0	1	1	0	0	EVX	**SP**
vspltb	0	0	0	1	0	0	vD	UIMM	vB	0	1	0	0	0	0	0	1	1	0	0	VX	**V**
evcntlzw	0	0	0	1	0	0	rD	rA	///	0	1	0	0	0	0	0	1	1	0	1	EVX	**SP**
evcntlsw	0	0	0	1	0	0	rD	rA	///	0	1	0	0	0	0	0	1	1	1	0	EVX	**SP**
vupkhsb	0	0	0	1	0	0	vD	///	vB	0	1	0	0	0	0	0	1	1	1	0	VX	**V**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
brinc	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	0	0	1	1	1	1	EVX	**SP**
evand	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	0	1	0	0	0	1	EVX	**SP**
evandc	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	0	1	0	0	1	0	EVX	**SP**
evxor	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	0	1	0	1	1	0	EVX	**SP**
evor	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	0	1	0	1	1	1	EVX	**SP**
evnor	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	0	1	1	0	0	0	EVX	**SP**
eveqv	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	0	1	1	0	0	1	EVX	**SP**
evorc	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	0	1	1	0	1	1	EVX	**SP**
evnand	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	0	1	1	1	1	0	EVX	**SP**
evsrwu	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	1	0	0	0	0	0	EVX	**SP**
evsrws	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	1	0	0	0	0	1	EVX	**SP**
evsrwiu	0	0	0	1	0	0	rD		rA	UIMM	0	1	0	0	0	1	0	0	0	1	0	EVX	**SP**
evsrwis	0	0	0	1	0	0	rD		rA	UIMM	0	1	0	0	0	1	0	0	0	1	1	EVX	**SP**
evslw	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	1	0	0	1	0	0	EVX	**SP**
evslwi	0	0	0	1	0	0	rD		rA	UIMM	0	1	0	0	0	1	0	0	1	1	0	EVX	**SP**
evrlw	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	1	0	1	0	0	0	EVX	**SP**
evsplati	0	0	0	1	0	0	rD		SIMM	///	0	1	0	0	0	1	0	1	0	0	1	EVX	**SP**
evrlwi	0	0	0	1	0	0	rD		rA	UIMM	0	1	0	0	0	1	0	1	0	1	0	EVX	**SP**
evsplatfi	0	0	0	1	0	0	rD		SIMM	///	0	1	0	0	0	1	0	1	0	1	1	EVX	**SP**
evmergehi	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	1	0	1	1	0	0	EVX	**SP**
evmergelo	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	1	0	1	1	0	1	EVX	**SP**
evmergehilo	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	1	0	1	1	1	0	EVX	**SP**
evmergelohi	0	0	0	1	0	0	rD		rA	rB	0	1	0	0	0	1	0	1	1	1	1	EVX	**SP**
evcmpgtu	0	0	0	1	0	0	crD	/ /	rA	rB	0	1	0	0	0	1	1	0	0	0	0	EVX	**SP**
evcmpgts	0	0	0	1	0	0	crD	/ /	rA	rB	0	1	0	0	0	1	1	0	0	0	1	EVX	**SP**
evcmpltu	0	0	0	1	0	0	crD	/ /	rA	rB	0	1	0	0	0	1	1	0	0	1	0	EVX	**SP**
evcmplts	0	0	0	1	0	0	crD	/ /	rA	rB	0	1	0	0	0	1	1	0	0	1	1	EVX	**SP**
evcmpeq	0	0	0	1	0	0	crD	/ /	rA	rB	0	1	0	0	0	1	1	0	1	0	0	EVX	**SP**
vadduhs	0	0	0	1	0	0	vD		vA	vB	0	1	0	0	1	0	0	0	0	0	0	VX	**V**
vminuh	0	0	0	1	0	0	vD		vA	vB	0	1	0	0	1	0	0	0	0	1	0	VX	**V**
vsrh	0	0	0	1	0	0	vD		vA	vB	0	1	0	0	1	0	0	0	1	0	0	VX	**V**
vcmpgtuh	0	0	0	1	0	0	vD		vA	vB	0	1	0	0	1	0	0	0	1	1	0	VC	**V**
vmuleuh	0	0	0	1	0	0	vD		vA	vB	0	1	0	0	1	0	0	1	0	0	0	VX	**V**
vrfiz	0	0	0	1	0	0	vD		///	vB	0	1	0	0	1	0	0	1	0	1	0	VX	**V**
vsplth	0	0	0	1	0	0	vD		UIMM	vB	0	1	0	0	1	0	0	1	1	0	0	VX	**V**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
vupkhsh	0	0	0	1	0	0	vD	///	vB	0	1	0	0	1	0	0	1	1	1	0	VX	**V**
evsel	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	1	1	1	1	crfS			EVX	**SP**
evfsadd	0	0	0	1	0	0	rD	rA	rB	0	1	0	1	0	0	0	0	0	0	0	EVX	**SP.FV**
vadduws	0	0	0	1	0	0	vD	vA	vB	0	1	0	1	0	0	0	0	0	0	0	VX	**V**
evfssub	0	0	0	1	0	0	rD	rA	rB	0	1	0	1	0	0	0	0	0	0	1	EVX	**SP.FV**
vminuw	0	0	0	1	0	0	vD	vA	vB	0	1	0	1	0	0	0	0	0	1	0	VX	**V**
evfsabs	0	0	0	1	0	0	rD	rA	///	0	1	0	1	0	0	0	0	1	0	0	EVX	**SP.FV**
vsrw	0	0	0	1	0	0	vD	vA	vB	0	1	0	1	0	0	0	0	1	0	0	VX	**V**
evfsnabs	0	0	0	1	0	0	rD	rA	///	0	1	0	1	0	0	0	0	1	0	1	EVX	**SP.FV**
evfsneg	0	0	0	1	0	0	rD	rA	///	0	1	0	1	0	0	0	0	1	1	0	EVX	**SP.FV**
vcmpgtuw	0	0	0	1	0	0	vD	vA	vB	0	1	0	1	0	0	0	0	1	1	0	VC	**V**
evfsmul	0	0	0	1	0	0	rD	rA	rB	0	1	0	1	0	0	0	1	0	0	0	EVX	**SP.FV**
evfsdiv	0	0	0	1	0	0	rD	rA	rB	0	1	0	1	0	0	0	1	0	0	1	EVX	**SP.FV**
vrfip	0	0	0	1	0	0	vD	///	vB	0	1	0	1	0	0	0	1	0	1	0	VX	**V**
evfscmpgt	0	0	0	1	0	0	crD / /	rA	rB	0	1	0	1	0	0	0	1	1	0	0	EVX	**SP.FV**
vspltw	0	0	0	1	0	0	vD	UIMM	vB	0	1	0	1	0	0	0	1	1	0	0	VX	**V**
evfscmplt	0	0	0	1	0	0	crD / /	rA	rB	0	1	0	1	0	0	0	1	1	0	1	EVX	**SP.FV**
evfscmpeq	0	0	0	1	0	0	crD / /	rA	rB	0	1	0	1	0	0	0	1	1	1	0	EVX	**SP.FV**
vupklsb	0	0	0	1	0	0	vD	///	vB	0	1	0	1	0	0	0	1	1	1	0	VX	**V**
evfscfui	0	0	0	1	0	0	rD	///	rB	0	1	0	1	0	0	1	0	0	0	0	EVX	**SP.FV**
evfscfsi	0	0	0	1	0	0	rD	///	rB	0	1	0	1	0	0	1	0	0	0	1	EVX	**SP.FV**
evfscfuf	0	0	0	1	0	0	rD	///	rB	0	1	0	1	0	0	1	0	0	1	0	EVX	**SP.FV**
evfscfsf	0	0	0	1	0	0	rD	///	rB	0	1	0	1	0	0	1	0	0	1	1	EVX	**SP.FV**
evfsctui	0	0	0	1	0	0	rD	///	rB	0	1	0	1	0	0	1	0	1	0	0	EVX	**SP.FV**
evfsctsi	0	0	0	1	0	0	rD	///	rB	0	1	0	1	0	0	1	0	1	0	1	EVX	**SP.FV**
evfsctuf	0	0	0	1	0	0	rD	///	rB	0	1	0	1	0	0	1	0	1	1	0	EVX	**SP.FV**
evfsctsf	0	0	0	1	0	0	rD	///	rB	0	1	0	1	0	0	1	0	1	1	1	EVX	**SP.FV**
evfsctuiz	0	0	0	1	0	0	rD	///	rB	0	1	0	1	0	0	1	1	0	0	0	EVX	**SP.FV**
evfsctsiz	0	0	0	1	0	0	rD	///	rB	0	1	0	1	0	0	1	1	0	1	0	EVX	**SP.FV**
evfststgt	0	0	0	1	0	0	crD / /	rA	rB	0	1	0	1	0	0	1	1	1	0	0	EVX	**SP.FV**
evfststlt	0	0	0	1	0	0	crD / /	rA	rB	0	1	0	1	0	0	1	1	1	0	1	EVX	**SP.FV**
evfststeq	0	0	0	1	0	0	crD / /	rA	rB	0	1	0	1	0	0	1	1	1	1	0	EVX	**SP.FV**
efsadd	0	0	0	1	0	0	rD	rA	rB	0	1	0	1	1	0	0	0	0	0	0	EVX	**SP.FS**
efssub	0	0	0	1	0	0	rD	rA	rB	0	1	0	1	1	0	0	0	0	0	1	EVX	**SP.FS**
efsabs	0	0	0	1	0	0	rD	rA	///	0	1	0	1	1	0	0	0	1	0	0	EVX	**SP.FS**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
vsr	0	0	0	1	0	0	vD					vA					vB					0	1	0	1	1	0	0	0	1	0	0	VX	**V**
efsnabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	0	0	0	1	0	1	EVX	**SP.FS**
efsneg	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	0	0	0	1	1	0	EVX	**SP.FS**
vcmpgtfp	0	0	0	1	0	0	vD					vA					vB					0	1	0	1	1	0	0	0	1	1	0	VC	**V**
efsmul	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	1	0	0	0	EVX	**SP.FS**
efsdiv	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	0	0	1	0	0	1	EVX	**SP.FS**
vrfim	0	0	0	1	0	0	vD					///					vB					0	1	0	1	1	0	0	1	0	1	0	VX	**V**
efscmpgt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	0	0	EVX	**SP.FS**
efscmplt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	0	1	EVX	**SP.FS**
efscmpeq	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	0	0	1	1	1	0	EVX	**SP.FS**
vupklsh	0	0	0	1	0	0	vD					///					vB					0	1	0	1	1	0	0	1	1	1	0	VX	**V**
efscfd	0	0	0	1	0	0	rD					0	0	0	0	0	rB					0	1	0	1	1	0	0	1	1	1	1	EVX	**SP.FS**
efscfui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	0	0	EVX	**SP.FS**
efscfsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	0	1	EVX	**SP.FS**
efscfuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	1	0	EVX	**SP.FS**
efscfsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	0	1	1	EVX	**SP.FS**
efsctui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	0	0	EVX	**SP.FS**
efsctsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	0	1	EVX	**SP.FS**
efsctuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	1	0	EVX	**SP.FS**
efsctsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	0	1	1	1	EVX	**SP.FS**
efsctuiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	1	0	0	0	EVX	**SP.FS**
efsctsiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	0	1	1	0	1	0	EVX	**SP.FS**
efststgt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	0	0	EVX	**SP.FS**
efststlt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	0	1	EVX	**SP.FS**
efststeq	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	0	1	1	1	1	0	EVX	**SP.FS**
efdadd	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	0	0	0	0	EVX	**SP.FD**
efdsub	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	0	0	0	1	EVX	**SP.FD**
efdabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	0	0	EVX	**SP.FD**
efdnabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	0	1	EVX	**SP.FD**
efdneg	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	1	0	EVX	**SP.FD**
efdmul	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	1	0	0	0	EVX	**SP.FD**
efddiv	0	0	0	1	0	0	rD					rA					rB					0	1	0	1	1	1	0	1	0	0	1	EVX	**SP.FD**
efdcmpgt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	1	0	1	1	0	0	EVX	**SP.FD**
efdcmplt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	1	0	1	1	0	1	EVX	**SP.FD**
efdcmpeq	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	1	0	1	1	1	0	EVX	**SP.FD**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
efdcfs	0	0	0	1	0	0	rD					0	0	0	0	0	rB					0	1	0	1	1	1	0	1	1	1	1	EVX	**SP.FD**
efdcfui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	0	0	EVX	**SP.FD**
efdcfsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	0	1	EVX	**SP.FD**
efdcfuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	1	0	EVX	**SP.FD**
efdcfsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	0	1	1	EVX	**SP.FD**
efdctui	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	0	0	EVX	**SP.FD**
efdctsi	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	0	1	EVX	**SP.FD**
efdctuf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	1	0	EVX	**SP.FD**
efdctsf	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	0	1	1	1	EVX	**SP.FD**
efdctuiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	1	0	0	0	EVX	**SP.FD**
efdctsiz	0	0	0	1	0	0	rD					///					rB					0	1	0	1	1	1	1	1	0	1	0	EVX	**SP.FD**
efdtstgt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	0	0	EVX	**SP.FD**
efdtstlt	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	0	1	EVX	**SP.FD**
efdtsteq	0	0	0	1	0	0	crD			/	/	rA					rB					0	1	0	1	1	1	1	1	1	1	0	EVX	**SP.FD**
evlddx	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	0	0	0	0	0	EVX	**SP**
vaddsbs	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	0	0	0	0	0	0	0	VX	**V**
evldd	0	0	0	1	0	0	rD					rA					UIMM[1]					0	1	1	0	0	0	0	0	0	0	1	EVX	**SP**
evldwx	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	0	0	0	1	0	EVX	**SP**
vminsb	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	0	0	0	0	0	1	0	VX	**V**
evldw	0	0	0	1	0	0	rD					rA					UIMM[2]					0	1	1	0	0	0	0	0	0	1	1	EVX	**SP**
evldhx	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	0	0	1	0	0	EVX	**SP**
vsrab	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	0	0	0	0	1	0	0	VX	**V**
evldh	0	0	0	1	0	0	rD					rA					UIMM[3]					0	1	1	0	0	0	0	0	1	0	1	EVX	**SP**
vcmpgtsb	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	0	0	0	0	1	1	0	VC	**V**
evlhhesplatx	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	0	1	0	0	0	EVX	**SP**
vmulesb	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	0	0	0	1	0	0	0	VX	**V**
evlhhesplat	0	0	0	1	0	0	rD					rA					UIMM[2]					0	1	1	0	0	0	0	1	0	0	1	EVX	**SP**
vcfux	0	0	0	1	0	0	vD					UIMM					vB					0	1	1	0	0	0	0	1	0	1	0	VX	**V**
evlhhousplatx	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	0	1	1	0	0	EVX	**SP**
vspltisb	0	0	0	1	0	0	vD					SIMM					///					0	1	1	0	0	0	0	1	1	0	0	VX	**V**
evlhhousplat	0	0	0	1	0	0	rD					rA					UIMM[2]					0	1	1	0	0	0	0	1	1	0	1	EVX	**SP**
evlhhossplatx	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	0	1	1	1	0	EVX	**SP**
vpkpx	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	0	0	0	1	1	1	0	VX	**V**
evlhhossplat	0	0	0	1	0	0	rD					rA					UIMM[2]					0	1	1	0	0	0	0	1	1	1	1	EVX	**SP**
evlwhex	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	1	0	0	0	0	EVX	**SP**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
evlwhe	0	0	0	1	0	0	rD					rA					UIMM[2]					0	1	1	0	0	0	1	0	0	0	1	EVX	**SP**
evlwhoux	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	1	0	1	0	0	EVX	**SP**
evlwhou	0	0	0	1	0	0	rD					rA					UIMM[2]					0	1	1	0	0	0	1	0	1	0	1	EVX	**SP**
evlwhosx	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	1	0	1	1	0	EVX	**SP**
evlwhos	0	0	0	1	0	0	rD					rA					UIMM[2]					0	1	1	0	0	0	1	0	1	1	1	EVX	**SP**
evlwwsplatx	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	1	1	0	0	0	EVX	**SP**
evlwwsplat	0	0	0	1	0	0	rD					rA					UIMM[3]					0	1	1	0	0	0	1	1	0	0	1	EVX	**SP**
evlwhsplatx	0	0	0	1	0	0	rD					rA					rB					0	1	1	0	0	0	1	1	1	0	0	EVX	**SP**
evlwhsplat	0	0	0	1	0	0	rD					rA					UIMM[2]					0	1	1	0	0	0	1	1	1	0	1	EVX	**SP**
evstddx	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	0	0	0	0	0	EVX	**SP**
evstdd	0	0	0	1	0	0	rD					rA					UIMM[1]					0	1	1	0	0	1	0	0	0	0	1	EVX	**SP**
evstdwx	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	0	0	0	1	0	EVX	**SP**
evstdw	0	0	0	1	0	0	rS					rA					UIMM[3]					0	1	1	0	0	1	0	0	0	1	1	EVX	**SP**
evstdhx	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	0	0	1	0	0	EVX	**SP**
evstdh	0	0	0	1	0	0	rS					rA					UIMM[2]					0	1	1	0	0	1	0	0	1	0	1	EVX	**SP**
evstwhex	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	1	0	0	0	0	EVX	**SP**
evstwhe	0	0	0	1	0	0	rS					rA					UIMM[2]					0	1	1	0	0	1	1	0	0	0	1	EVX	**SP**
evstwhox	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	1	0	1	0	0	EVX	**SP**
evstwho	0	0	0	1	0	0	rS					rA					UIMM[2]					0	1	1	0	0	1	1	0	1	0	1	EVX	**SP**
evstwwex	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	1	1	0	0	0	EVX	**SP**
evstwwe	0	0	0	1	0	0	rS					rA					UIMM[3]					0	1	1	0	0	1	1	1	0	0	1	EVX	**SP**
evstwwox	0	0	0	1	0	0	rS					rA					rB					0	1	1	0	0	1	1	1	1	0	0	EVX	**SP**
evstwwo	0	0	0	1	0	0	rS					rA					UIMM[3]					0	1	1	0	0	1	1	1	1	0	1	EVX	**SP**
vaddshs	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	1	0	0	0	0	0	0	VX	**V**
vminsh	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	1	0	0	0	0	1	0	VX	**V**
vsrah	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	1	0	0	0	1	0	0	VX	**V**
vcmpgtsh	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	1	0	0	0	1	1	0	VC	**V**
vmulesh	0	0	0	1	0	0	vD					vA					vB					0	1	1	0	1	0	0	1	0	0	0	VX	**V**
vcfsx	0	0	0	1	0	0	vD					UIMM					vB					0	1	1	0	1	0	0	1	0	1	0	VX	**V**
vspltish	0	0	0	1	0	0	vD					SIMM					///					0	1	1	0	1	0	0	1	1	0	0	VX	**V**
vupkhpx	0	0	0	1	0	0	vD					///					vB					0	1	1	0	1	0	0	1	1	1	0	VX	**V**
vaddsws	0	0	0	1	0	0	vD					vA					vB					0	1	1	1	0	0	0	0	0	0	0	VX	**V**
vminsw	0	0	0	1	0	0	vD					vA					vB					0	1	1	1	0	0	0	0	0	1	0	VX	**V**
vsraw	0	0	0	1	0	0	vD					vA					vB					0	1	1	1	0	0	0	0	1	0	0	VX	**V**
vcmpgtsw	0	0	0	1	0	0	vD					vA					vB					0	1	1	1	0	0	0	0	1	1	0	VC	**V**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
vctuxs	0	0	0	1	0	0	vD	UIMM	vB	0	1	1	1	0	0	0	1	0	1	0	VX	**V**
vspltisw	0	0	0	1	0	0	vD	SIMM	///	0	1	1	1	0	0	0	1	1	0	0	VX	**V**
vcmpbfp	0	0	0	1	0	0	vD	vA	vB	0	1	1	1	1	0	0	0	1	1	0	VC	**V**
vctsxs	0	0	0	1	0	0	vD	UIMM	vB	0	1	1	1	1	0	0	1	0	1	0	VX	**V**
vupklpx	0	0	0	1	0	0	vD	///	vB	0	1	1	1	1	0	0	1	1	1	0	VX	**V**
vsububm	0	0	0	1	0	0	vD	vA	vB	1	0	0	0	0	0	0	0	0	0	0	VX	**V**
vavgub	0	0	0	1	0	0	vD	vA	vB	1	0	0	0	0	0	0	0	0	1	0	VX	**V**
evmhessf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	0	0	1	1	EVX	**SP**
vand	0	0	0	1	0	0	vD	vA	vB	1	0	0	0	0	0	0	0	1	0	0	VX	**V**
vcmpequb.	0	0	0	1	0	0	vD	vA	vB	1	0	0	0	0	0	0	0	1	1	0	VC	**V**
evmhossf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	0	1	1	1	EVX	**SP**
evmheumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	0	0	0	EVX	**SP**
evmhesmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	0	0	1	EVX	**SP**
vmaxfp	0	0	0	1	0	0	vD	vA	vB	1	0	0	0	0	0	0	1	0	1	0	VX	**V**
evmhesmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	0	1	1	EVX	**SP**
evmhoumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	1	0	0	EVX	**SP**
vslo	0	0	0	1	0	0	vD	vA	vB	1	0	0	0	0	0	0	1	1	0	0	VX	**V**
evmhosmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	1	0	1	EVX	**SP**
evmhosmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	1	1	1	EVX	**SP**
evmhessfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	0	0	1	1	EVX	**SP**
evmhossfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	0	1	1	1	EVX	**SP**
evmheumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	0	0	0	EVX	**SP**
evmhesmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	0	0	1	EVX	**SP**
evmhesmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	0	1	1	EVX	**SP**
evmhoumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	1	0	0	EVX	**SP**
evmhosmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	1	0	1	EVX	**SP**
evmhosmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	1	1	1	EVX	**SP**
vsubuhm	0	0	0	1	0	0	vD	vA	vB	1	0	0	0	1	0	0	0	0	0	0	VX	**V**
vavguh	0	0	0	1	0	0	vD	vA	vB	1	0	0	0	1	0	0	0	0	1	0	VX	**V**
vandc	0	0	0	1	0	0	vD	vA	vB	1	0	0	0	1	0	0	0	1	0	0	VX	**V**
vcmpequh.	0	0	0	1	0	0	vD	vA	vB	1	0	0	0	1	0	0	0	1	1	0	VC	**V**
evmwhssf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	0	1	1	1	EVX	**SP**
evmwlumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	0	0	0	EVX	**SP**
vminfp	0	0	0	1	0	0	vD	vA	vB	1	0	0	0	1	0	0	1	0	1	0	VX	**V**
evmwhumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	1	0	0	EVX	**SP**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
vsro	0	0	0	1	0	0	vD	vA	vB	1	0	0	0	1	0	0	1	1	0	0	VX	**V**
evmwhsmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	1	0	1	EVX	**SP**
evmwhsmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	1	1	1	EVX	**SP**
evmwssf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	0	0	1	1	EVX	**SP**
evmwumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	1	0	0	0	EVX	**SP**
evmwsmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	1	0	0	1	EVX	**SP**
evmwsmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	1	0	1	1	EVX	**SP**
evmwhssfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	0	1	1	1	EVX	**SP**
evmwlumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	0	0	0	EVX	**SP**
evmwhumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	1	0	0	EVX	**SP**
evmwhsmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	1	0	1	EVX	**SP**
evmwhsmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	1	1	1	EVX	**SP**
evmwssfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	0	0	1	1	EVX	**SP**
evmwumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	1	0	0	0	EVX	**SP**
evmwsmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	1	0	0	1	EVX	**SP**
evmwsmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	1	0	1	1	EVX	**SP**
vsubuwm	0	0	0	1	0	0	vD	vA	vB	1	0	0	1	0	0	0	0	0	0	0	VX	**V**
vavguw	0	0	0	1	0	0	vD	vA	vB	1	0	0	1	0	0	0	0	0	1	0	VX	**V**
vor	0	0	0	1	0	0	vD	vA	vB	1	0	0	1	0	0	0	0	1	0	0	VX	**V**
vcmpequw.	0	0	0	1	0	0	vD	vA	vB	1	0	0	1	0	0	0	0	1	1	0	VC	**V**
evaddusiaaw	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	0	0	0	0	EVX	**SP**
evaddssiaaw	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	0	0	0	1	EVX	**SP**
evsubfusiaaw	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	0	0	1	0	EVX	**SP**
evsubfssiaaw	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	0	0	1	1	EVX	**SP**
evmra	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	0	1	0	0	EVX	**SP**
vxor	0	0	0	1	0	0	vD	vA	vB	1	0	0	1	1	0	0	0	1	0	0	VX	**V**
evdivws	0	0	0	1	0	0	rD	rA	rB	1	0	0	1	1	0	0	0	1	1	0	EVX	**SP**
vcmpeqfp.	0	0	0	1	0	0	vD	vA	vB	1	0	0	1	1	0	0	0	1	1	0	VC	**V**
evdivwu	0	0	0	1	0	0	rD	rA	rB	1	0	0	1	1	0	0	0	1	1	1	EVX	**SP**
evaddumiaaw	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	1	0	0	0	EVX	**SP**
evaddsmiaaw	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	1	0	0	1	EVX	**SP**
evsubfumiaaw	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	1	0	1	0	EVX	**SP**
evsubfsmiaaw	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	1	0	1	1	EVX	**SP**
evmheusiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	0	0	0	0	EVX	**SP**
evmhessiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	0	0	0	1	EVX	**SP**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
vavgsb	0	0	0	1	0	0	vD					vA					vB					1	0	1	0	0	0	0	0	0	1	0	VX	**V**
evmhessfaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	0	0	1	1	EVX	**SP**
evmhousiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	0	1	0	0	EVX	**SP**
vnor	0	0	0	1	0	0	vD					vA					vB					1	0	1	0	0	0	0	0	1	0	0	VX	**V**
evmhossiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	0	1	0	1	EVX	**SP**
evmhossfaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	0	1	1	1	EVX	**SP**
evmheumiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	1	0	0	0	EVX	**SP**
evmhesmiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	1	0	0	1	EVX	**SP**
evmhesmfaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	1	0	1	1	EVX	**SP**
evmhoumiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	1	1	0	0	EVX	**SP**
evmhosmiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	1	1	0	1	EVX	**SP**
evmhosmfaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	0	0	1	1	1	1	EVX	**SP**
evmhegumiaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	1	0	1	0	0	0	EVX	**SP**
evmhegsmiaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	1	0	1	0	0	1	EVX	**SP**
evmhegsmfaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	1	0	1	0	1	1	EVX	**SP**
evmhogumiaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	1	0	1	1	0	0	EVX	**SP**
evmhogsmiaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	1	0	1	1	0	1	EVX	**SP**
evmhogsmfaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	0	1	0	1	1	1	1	EVX	**SP**
evmwlusiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	1	0	0	0	0	0	0	EVX	**SP**
evmwlssiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	1	0	0	0	0	0	1	EVX	**SP**
vavgsh	0	0	0	1	0	0	vD					vA					vB					1	0	1	0	1	0	0	0	0	1	0	VX	**V**
evmwhssmaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	1	0	0	0	1	0	1	EVX	**SP**
evmwlumiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	1	0	0	1	0	0	0	EVX	**SP**
evmwlsmiaaw	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	1	0	0	1	0	0	1	EVX	**SP**
evmwssfaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	1	0	1	0	0	1	1	EVX	**SP**
evmwumiaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	1	0	1	1	0	0	0	EVX	**SP**
evmwsmiaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	1	0	1	1	0	0	1	EVX	**SP**
evmwsmfaa	0	0	0	1	0	0	rD					rA					rB					1	0	1	0	1	0	1	1	0	1	1	EVX	**SP**
evmheusianw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	0	0	0	0	EVX	**SP**
vsubcuw	0	0	0	1	0	0	vD					vA					vB					1	0	1	1	0	0	0	0	0	0	0	VX	**V**
evmhessianw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	0	0	0	1	EVX	**SP**
vavgsw	0	0	0	1	0	0	vD					vA					vB					1	0	1	1	0	0	0	0	0	1	0	VX	**V**
evmhessfanw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	0	0	1	1	EVX	**SP**
evmhousianw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	0	1	0	0	EVX	**SP**
evmhossianw	0	0	0	1	0	0	rD					rA					rB					1	0	1	1	0	0	0	0	1	0	1	EVX	**SP**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
evmhossfanw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	0	1	1	1	EVX	**SP**
evmheumianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	0	0	0	EVX	**SP**
evmhesmianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	0	0	1	EVX	**SP**
evmhesmfanw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	0	1	1	EVX	**SP**
evmhoumianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	1	0	0	EVX	**SP**
evmhosmianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	1	0	1	EVX	**SP**
evmhosmfanw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	1	1	1	EVX	**SP**
evmhegumian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	0	0	0	EVX	**SP**
evmhegsmian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	0	0	1	EVX	**SP**
evmhegsmfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	0	1	1	EVX	**SP**
evmhogumian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	1	0	0	EVX	**SP**
evmhogsmian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	1	0	1	EVX	**SP**
evmhogsmfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	1	1	1	EVX	**SP**
evmwlusianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	0	0	0	0	EVX	**SP**
evmwlssianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	0	0	0	1	EVX	**SP**
vcmpgefp.	0	0	0	1	0	0	vD	vA	vB	1	0	1	1	1	0	0	0	1	1	0	VC	**V**
evmwlumianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	1	0	0	0	EVX	**SP**
evmwlsmianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	1	0	0	1	EVX	**SP**
evmwssfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	0	0	1	1	EVX	**SP**
evmwumian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	1	0	0	0	EVX	**SP**
evmwsmian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	1	0	0	1	EVX	**SP**
evmwsmfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	1	0	1	1	EVX	**SP**
vsububs	0	0	0	1	0	0	vD	vA	vB	1	1	0	0	0	0	0	0	0	0	0	VX	**V**
mfvscr	0	0	0	1	0	0	vD	///	///	1	1	0	0	0	0	0	0	1	0	0	VX	**V**
vcmpgtub.	0	0	0	1	0	0	vD	vA	vB	1	1	0	0	0	0	0	0	1	1	0	VC	**V**
vsum4ubs	0	0	0	1	0	0	vD	vA	vB	1	1	0	0	0	0	0	1	0	0	0	VX	**V**
vsubuhs	0	0	0	1	0	0	vD	vA	vB	1	1	0	0	1	0	0	0	0	0	0	VX	**V**
vcmpgtuh.	0	0	0	1	0	0	vD	vA	vB	1	1	0	0	1	0	0	0	1	1	0	VC	**V**
vsum4shs	0	0	0	1	0	0	vD	vA	vB	1	1	0	0	1	0	0	1	0	0	0	VX	**V**
vsubuws	0	0	0	1	0	0	vD	vA	vB	1	1	0	1	0	0	0	0	0	0	0	VX	**V**
vcmpgtuw.	0	0	0	1	0	0	vD	vA	vB	1	1	0	1	0	0	0	0	1	1	0	VC	**V**
vsum2sws	0	0	0	1	0	0	vD	vA	vB	1	1	0	1	0	0	0	1	0	0	0	VX	**V**
vcmpgtfp.	0	0	0	1	0	0	vD	vA	vB	1	1	0	1	1	0	0	0	1	1	0	VC	**V**
vsubsbs	0	0	0	1	0	0	vD	vA	vB	1	1	1	0	0	0	0	0	0	0	0	VX	**V**
vcmpgtsb.	0	0	0	1	0	0	vD	vA	vB	1	1	1	0	0	0	0	0	1	1	0	VC	**V**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
vsum4sbs	0	0	0	1	0	0	vD					vA					vB					1	1	1	0	0	0	0	1	0	0	0	VX	**V**
vsubshs	0	0	0	1	0	0	vD					vA					vB					1	1	1	0	1	0	0	0	0	0	0	VX	**V**
vcmpgtsh.	0	0	0	1	0	0	vD					vA					vB					1	1	1	0	1	0	0	0	1	1	0	VC	**V**
vsubsws	0	0	0	1	0	0	vD					vA					vB					1	1	1	1	0	0	0	0	0	0	0	VX	**V**
vcmpgtsw.	0	0	0	1	0	0	vD					vA					vB					1	1	1	1	0	0	0	0	1	1	0	VC	**V**
vsumsws	0	0	0	1	0	0	vD					vA					vB					1	1	1	1	0	0	0	1	0	0	0	VX	**V**
vcmpbfp.	0	0	0	1	0	0	vD					vA					vB					1	1	1	1	1	0	0	0	1	1	0	VC	**V**
vmhaddshs	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	0	0	0	VA	**V**
vmhraddshs	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	0	0	1	VA	**V**
vmladduhm	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	0	1	0	VA	**V**
vmsumubm	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	1	0	0	VA	**V**
vmsummbm	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	1	0	1	VA	**V**
vmsumuhm	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	1	1	0	VA	**V**
vmsumuhs	0	0	0	1	0	0	vD					vA					vB					vC					1	0	0	1	1	1	VA	**V**
vmsumshm	0	0	0	1	0	0	vD					vA					vB					vC					1	0	1	0	0	0	VA	**V**
vmsumshs	0	0	0	1	0	0	vD					vA					vB					vC					1	0	1	0	0	1	VA	**V**
vsel	0	0	0	1	0	0	vD					vA					vB					vC					1	0	1	0	1	0	VA	**V**
vperm	0	0	0	1	0	0	vD					vA					vB					vC					1	0	1	0	1	1	VA	**V**
vsldoi	0	0	0	1	0	0	vD					vA					vB					/	SH				1	0	1	1	0	0	VX	**V**
vmaddfp	0	0	0	1	0	0	vD					vA					vB					vC					1	0	1	1	1	0	VA	**V**
vnmsubfp	0	0	0	1	0	0	vD					vA					vB					vC					1	0	1	1	1	1	VA	**V**
mulli	0	0	0	1	1	1	rD					rA					SIMM																D	
subfic	0	0	1	0	0	0	rD					rA					SIMM																D	
cmpli	0	0	1	0	1	0	crD			/	L	rA					UIMM																D	
cmpi	0	0	1	0	1	1	crD			/	L	rA					SIMM																D	
addic	0	0	1	1	0	0	rD					rA					SIMM																D	
addic.	0	0	1	1	0	1	rD					rA					SIMM																D	
addi	0	0	1	1	1	0	rD					rA					SIMM																D	
addis	0	0	1	1	1	1	rD					rA					SIMM																D	
bc	0	1	0	0	0	0	BO					BI					BD														0	0	B	
bcl	0	1	0	0	0	0	BO					BI					BD														0	1	B	
bca	0	1	0	0	0	0	BO					BI					BD														1	0	B	
bcla	0	1	0	0	0	0	BO					BI					BD														1	1	B	
sc	0	1	0	0	0	1	///														LEV							///			1	/	SC	
b	0	1	0	0	1	0	LI																								0	0	I	

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
bl	0	1	0	0	1	0	LI																								0	1	I	
ba	0	1	0	0	1	0	LI																								1	0	I	
bla	0	1	0	0	1	0	LI																								1	1	I	
mcrf	0	1	0	0	1	1	crD			//		crfS			///							0	0	0	0	0	0	0	0	0	0	/	XL	
bclr	0	1	0	0	1	1	BO					BI					///			BH		0	0	0	0	0	1	0	0	0	0	0	XL	
bclrl	0	1	0	0	1	1	BO					BI					///			BH		0	0	0	0	0	1	0	0	0	0	1	XL	
crnor	0	1	0	0	1	1	crbD					crbA					crbB					0	0	0	0	1	0	0	0	0	1	/	XL	
rfmci	0	1	0	0	1	1	///															0	0	0	0	1	0	0	1	1	0	/	XL	**Embedded**
rfdi	0	1	0	0	1	1	///															0	0	0	0	1	0	0	1	1	1	/	X	**E.ED**
rfi	0	1	0	0	1	1	///															0	0	0	0	1	1	0	0	1	0	/	XL	**Embedded**
rfci	0	1	0	0	1	1	///															0	0	0	0	1	1	0	0	1	1	/	XL	**Embedded**
rfgi	0	1	0	0	1	1	///															0	0	0	1	1	0	0	1	1	0	/	X	**E.HV**
crandc	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	0	0	0	0	0	0	1	/	XL	
isync	0	1	0	0	1	1	///															0	0	1	0	0	1	0	1	1	0	/	XL	
crxor	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	1	0	0	0	0	0	1	/	XL	
dnh	0	1	0	0	1	1	DUI					DCTL					0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	/	X	**E.ED**
crnand	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	1	1	0	0	0	0	1	/	XL	
crand	0	1	0	0	1	1	crbD					crbA					crbB					0	1	0	0	0	0	0	0	0	1	/	XL	
creqv	0	1	0	0	1	1	crbD					crbA					crbB					0	1	0	0	1	0	0	0	0	1	/	XL	
crorc	0	1	0	0	1	1	crbD					crbA					crbB					0	1	1	0	1	0	0	0	0	1	/	XL	
cror	0	1	0	0	1	1	crbD					crbA					crbB					0	1	1	1	0	0	0	0	0	1	/	XL	
bcctr	0	1	0	0	1	1	BO					BI					///			BH		1	0	0	0	0	1	0	0	0	0	0	XL	
bcctrl	0	1	0	0	1	1	BO					BI					///			BH		1	0	0	0	0	1	0	0	0	0	1	XL	
rlwimi	0	1	0	1	0	0	rS					rA					SH					MB					ME					0	M	
rlwimi.	0	1	0	1	0	0	rS					rA					SH					MB					ME					1	M	
rlwinm	0	1	0	1	0	1	rS					rA					SH					MB					ME					0	M	
rlwinm.	0	1	0	1	0	1	rS					rA					SH					MB					ME					1	M	
rlwnm	0	1	0	1	1	1	rS					rA					rB					MB					ME					0	M	
rlwnm.	0	1	0	1	1	1	rS					rA					rB					MB					ME					1	M	
ori	0	1	1	0	0	0	rS					rA					UIMM																D	
oris	0	1	1	0	0	1	rS					rA					UIMM																D	
xori	0	1	1	0	1	0	rS					rA					UIMM																D	
xoris	0	1	1	0	1	1	rS					rA					UIMM																D	
andi.	0	1	1	1	0	0	rS					rA					UIMM																D	
andis.	0	1	1	1	0	1	rS					rA					UIMM																D	

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
rldicl	0	1	1	1	1	0	rS					rA					sh1–5					mb1–5					mb0	0	0	0	sh0	0	MD	**64**
rldicl.	0	1	1	1	1	0	rS					rA					sh1–5					mb1–5					mb0	0	0	0	sh0	1	MD	**64**
rldicr	0	1	1	1	1	0	rS					rA					sh1–5					me1–5					me0	0	0	1	sh0	0	MD	**64**
rldicr.	0	1	1	1	1	0	rS					rA					sh1–5					me1–5					me0	0	0	1	sh0	1	MD	**64**
rldic	0	1	1	1	1	0	rS					rA					sh1–5					mb1–5					mb0	0	1	0	sh0	0	MD	**64**
rldic.	0	1	1	1	1	0	rS					rA					sh1–5					mb1–5					mb0	0	1	0	sh0	1	MD	**64**
rldimi	0	1	1	1	1	0	rS					rA					sh1–5					mb1–5					mb0	0	1	1	sh0	0	MD	**64**
rldimi.	0	1	1	1	1	0	rS					rA					sh1–5					mb1–5					mb0	0	1	1	sh0	1	MD	**64**
rldcl	0	1	1	1	1	0	rS					rA					rB					mb1–5					mb0	1	0	0	0	0	MDS	**64**
rldcl.	0	1	1	1	1	0	rS					rA					rB					mb1–5					mb0	1	0	0	0	1	MDS	**64**
rldcr	0	1	1	1	1	0	rS					rA					rB					me1–5					me0	1	0	0	1	0	MDS	**64**
rldcr.	0	1	1	1	1	0	rS					rA					rB					me1–5					me0	1	0	0	1	1	MDS	**64**
cmp	0	1	1	1	1	1	crD			/	L	rA					rB					0	0	0	0	0	0	0	0	0	0	/	X	
tw	0	1	1	1	1	1	TO					rA					rB					0	0	0	0	0	0	0	1	0	0	/	X	
lvsl	0	1	1	1	1	1	vD					rA					rB					0	0	0	0	0	0	0	1	1	0	/	X	**V**
lvebx	0	1	1	1	1	1	vD					rA					rB					0	0	0	0	0	0	0	1	1	1	/	X	**V**
subfc	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	0	0	0	X	
subfc.	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	0	0	1	X	
mulhdu	0	1	1	1	1	1	rD					rA					rB					/	0	0	0	0	0	1	0	0	1	0	X	**64**
mulhdu.	0	1	1	1	1	1	rD					rA					rB					/	0	0	0	0	0	1	0	0	1	1	X	**64**
addc	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	1	0	0	X	
addc.	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	1	0	1	X	
mulhwu	0	1	1	1	1	1	rD					rA					rB					/	0	0	0	0	0	1	0	1	1	0	X	
mulhwu.	0	1	1	1	1	1	rD					rA					rB					/	0	0	0	0	0	1	0	1	1	1	X	
isel	0	1	1	1	1	1	rD					rA					rB					crb					0	1	1	1	1	0	A	
mfcr	0	1	1	1	1	1	rD					0	///									0	0	0	0	0	1	0	0	1	1	/	X	
mfocrf	0	1	1	1	1	1	rD					1	CRM								/	0	0	0	0	0	1	0	0	1	1	/	X	
tlbilx	0	1	1	1	1	1	0	///		T		rA					rB					0	0	0	0	0	1	0	0	1	0	/	X	**Embedded**
lwarx	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	1	0	1	0	0	/	X	
ldx	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	1	0	1	0	1	/	X	**64**
icbt	0	1	1	1	1	1	CT					rA					rB					0	0	0	0	0	1	0	1	1	0	/	X	**Embedded**
lwzx	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	1	0	1	1	1	/	X	
slw	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	0	0	0	0	X	
slw.	0	1	1	1	1	1	rS					rA					rB					0	0	0	0	0	1	1	0	0	0	1	X	
cntlzw	0	1	1	1	1	1	rS					rA					///					0	0	0	0	0	1	1	0	1	0	0	X	

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
cntlzw.	0	1	1	1	1	1			rS					rA					///			0	0	0	0	0	1	1	0	1	0	1	X	
sld	0	1	1	1	1	1			rS					rA					rB			0	0	0	0	0	1	1	0	1	1	0	X	**64**
sld.	0	1	1	1	1	1			rS					rA					rB			0	0	0	0	0	1	1	0	1	1	1	X	**64**
and	0	1	1	1	1	1			rS					rA					rB			0	0	0	0	0	1	1	1	0	0	0	X	
and.	0	1	1	1	1	1			rS					rA					rB			0	0	0	0	0	1	1	1	0	0	1	X	
ldepx	0	1	1	1	1	1			rD					rA					rB			0	0	0	0	0	1	1	1	0	1	/	X	**E.PD, 64**
lwepx	0	1	1	1	1	1			rD					rA					rB			0	0	0	0	0	1	1	1	1	1	/	X	**E.PD**
cmpl	0	1	1	1	1	1	/	L			rA					rB				///		0	0	0	0	1	0	0	0	0	0	/	X	
lvsr	0	1	1	1	1	1			vD					rA					rB			0	0	0	0	1	0	0	1	1	0	/	X	**V**
lvehx	0	1	1	1	1	1			vD					rA					rB			0	0	0	0	1	0	0	1	1	1	/	X	**V**
subf	0	1	1	1	1	1			rD					rA					rB			0	0	0	0	1	0	1	0	0	0	0	X	
subf.	0	1	1	1	1	1			rD					rA					rB			0	0	0	0	1	0	1	0	0	0	1	X	
lbarx	0	1	1	1	1	1			rD					rA					rB			0	0	0	0	1	1	0	1	0	0	/	X	**ER**
ldux	0	1	1	1	1	1			rD					rA					rB			0	0	0	0	1	1	0	1	0	1	/	X	**64**
dcbst	0	1	1	1	1	1			///					rA					rB			0	0	0	0	1	1	0	1	1	0	/	X	
lwzux	0	1	1	1	1	1			rD					rA					rB			0	0	0	0	1	1	0	1	1	1	/	X	
cntlzd	0	1	1	1	1	1			rS					rA					///			0	0	0	0	1	1	1	0	1	0	0	**X**	**64**
cntlzd.	0	1	1	1	1	1			rS					rA					///			0	0	0	0	1	1	1	0	1	0	1	**X**	**64**
andc	0	1	1	1	1	1			rS					rA					rB			0	0	0	0	1	1	1	1	0	0	0	X	
andc.	0	1	1	1	1	1			rS					rA					rB			0	0	0	0	1	1	1	1	0	0	1	X	
wait	0	1	1	1	1	1		///		WC		WH					///					0	0	0	0	1	1	1	1	1	0	/	X	**WT**
dcbstep	0	1	1	1	1	1			///					rA					rB			0	0	0	0	1	1	1	1	1	1	/	X	**E.PD**
td	0	1	1	1	1	1			TO					rA					rB			0	0	0	1	0	0	0	1	0	0	/	X	**64**
lvewx	0	1	1	1	1	1			vD					rA					rB			0	0	0	1	0	0	0	1	1	1	/	X	**V**
mulhd	0	1	1	1	1	1			rD					rA					rB			/	0	0	1	0	0	1	0	0	1	0	X	**64**
mulhd.	0	1	1	1	1	1			rD					rA					rB			/	0	0	1	0	0	1	0	0	1	1	X	**64**
mulhw	0	1	1	1	1	1			rD					rA					rB			/	0	0	1	0	0	1	0	1	1	0	X	
mulhw.	0	1	1	1	1	1			rD					rA					rB			/	0	0	1	0	0	1	0	1	1	1	X	
mfmsr	0	1	1	1	1	1			rD							///						0	0	0	1	0	1	0	0	1	1	/	X	
ldarx	0	1	1	1	1	1			rD					rA					rB			0	0	0	1	0	1	0	1	0	0	/	X	**64**
dcbf	0	1	1	1	1	1			///					rA					rB			0	0	0	1	0	1	0	1	1	0	/	X	
lbzx	0	1	1	1	1	1			rD					rA					rB			0	0	0	1	0	1	0	1	1	1	/	X	
lbepx	0	1	1	1	1	1			rD					rA					rB			0	0	0	1	0	1	1	1	1	1	/	X	**E.PD**
dni	0	1	1	1	1	1			DUI					DCTL			0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	X	**E.ED**
lvx	0	1	1	1	1	1			vD					rA					rB			0	0	0	1	1	0	0	1	1	1	/	X	**V**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
neg	0	1	1	1	1	1			rD					rA					///			0	0	0	1	1	0	1	0	0	0	0	X	
neg.	0	1	1	1	1	1			rD					rA					///			0	0	0	1	1	0	1	0	0	0	1	X	
lharx	0	1	1	1	1	1			rD					rA					rB			0	0	0	1	1	1	0	1	0	0	/	X	**ER**
lbzux	0	1	1	1	1	1			rD					rA					rB			0	0	0	1	1	1	0	1	1	1	/	X	
popcntb	0	1	1	1	1	1			rS					rA					///			0	0	0	1	1	1	1	0	1	0	/	X	
nor	0	1	1	1	1	1			rS					rA					rB			0	0	0	1	1	1	1	1	0	0	0	X	
nor.	0	1	1	1	1	1			rS					rA					rB			0	0	0	1	1	1	1	1	0	0	1	X	
dcbfep	0	1	1	1	1	1			///					rA					rB			0	0	0	1	1	1	1	1	1	1	/	X	**E.PD**
wrtee	0	1	1	1	1	1			rS							///						0	0	1	0	0	0	0	0	1	1	/	X	**Embedded**
dcbtstls	0	1	1	1	1	1			CT					rA					rB			0	0	1	0	0	0	0	1	1	0	/	X	**E.CL**
stvebx	0	1	1	1	1	1			vS					rA					rB			0	0	1	0	0	0	0	1	1	1	/	X	**V**
subfe	0	1	1	1	1	1			rD					rA					rB			0	0	1	0	0	0	1	0	0	0	0	X	
subfe.	0	1	1	1	1	1			rD					rA					rB			0	0	1	0	0	0	1	0	0	0	1	X	
adde	0	1	1	1	1	1			rD					rA					rB			0	0	1	0	0	0	1	0	1	0	0	X	
adde.	0	1	1	1	1	1			rD					rA					rB			0	0	1	0	0	0	1	0	1	0	1	X	
mtcrf	0	1	1	1	1	1			rS			0				CRM					/	0	0	1	0	0	1	0	0	0	0	/	XFX	
mtocrf	0	1	1	1	1	1			rS			1				CRM					/	0	0	1	0	0	1	0	0	0	0	/	XFX	
mtmsr	0	1	1	1	1	1			rS							///						0	0	1	0	0	1	0	0	1	0	/	X	
stdx	0	1	1	1	1	1			rS					rA					rB			0	0	1	0	0	1	0	1	0	1	/	X	**64**
stwcx.	0	1	1	1	1	1			rS					rA					rB			0	0	1	0	0	1	0	1	1	0	1	X	
stwx	0	1	1	1	1	1			rS					rA					rB			0	0	1	0	0	1	0	1	1	1	/	D	
prtyw	0	1	1	1	1	1			rS					rA					///			0	0	1	0	0	1	1	0	1	0	/	X	
stdepx	0	1	1	1	1	1			rS					rA					rB			0	0	1	0	0	1	1	1	0	1	/	X	**E.PD, 64**
stwepx	0	1	1	1	1	1			rS					rA					rB			0	0	1	0	0	1	1	1	1	1	/	X	**E.PD**
wrteei	0	1	1	1	1	1						///					E		///			0	0	1	0	1	0	0	0	1	1	/	X	**Embedded**
dcbtls	0	1	1	1	1	1			CT					rA					rB			0	0	1	0	1	0	0	1	1	0	/	X	**E.CL**
stvehx	0	1	1	1	1	1			vS					rA					rB			0	0	1	0	1	0	0	1	1	1	/	X	**V**
stdux	0	1	1	1	1	1			rS					rA					rB			0	0	1	0	1	1	0	1	0	1	/	X	**64**
stwux	0	1	1	1	1	1			rS					rA					rB			0	0	1	0	1	1	0	1	1	1	/	D	
prtyd	0	1	1	1	1	1			rS					rA					///			0	0	1	0	1	1	1	0	1	0	/	X	**64**
stvewx	0	1	1	1	1	1			vS					rA					rB			0	0	1	1	0	0	0	1	1	1	/	X	**V**
subfze	0	1	1	1	1	1			rD					rA					///			0	0	1	1	0	0	1	0	0	0	0	X	
subfze.	0	1	1	1	1	1			rD					rA					///			0	0	1	1	0	0	1	0	0	0	1	X	
addze	0	1	1	1	1	1			rD					rA					///			0	0	1	1	0	0	1	0	1	0	0	X	
addze.	0	1	1	1	1	1			rD					rA					///			0	0	1	1	0	0	1	0	1	0	1	X	

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
msgsnd	0	1	1	1	1	1	///	///	rB	0	0	1	1	0	0	1	1	1	0	/	X	**E.PC**
stdcx.	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	0	1	0	1	1	0	1	X	**64**
stbx	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	0	1	0	1	1	1	/	X	
stbepx	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	0	1	1	1	1	1	/	X	**E.PD**
sthepx	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	0	1	1	1	1	1	/	X	**E.PD**
icblc	0	1	1	1	1	1	CT	rA	rB	0	0	1	1	1	0	0	1	1	0	/	X	**E.CL**
stvx	0	1	1	1	1	1	vS	rA	rB	0	0	1	1	1	0	0	1	1	1	/	X	**V**
mulld	0	1	1	1	1	1	rD	rA	rB	0	0	1	1	1	0	1	0	0	1	0	X	**64**
mulld.	0	1	1	1	1	1	rD	rA	rB	0	0	1	1	1	0	1	0	0	1	1	X	**64**
subfme	0	1	1	1	1	1	rD	rA	///	0	0	1	1	1	0	1	0	0	0	0	X	
subfme.	0	1	1	1	1	1	rD	rA	///	0	0	1	1	1	0	1	0	0	0	1	X	
addme	0	1	1	1	1	1	rD	rA	///	0	0	1	1	1	0	1	0	1	0	0	X	
addme.	0	1	1	1	1	1	rD	rA	///	0	0	1	1	1	0	1	0	1	0	1	X	
mullw	0	1	1	1	1	1	rD	rA	rB	0	0	1	1	1	0	1	0	1	1	0	X	
mullw.	0	1	1	1	1	1	rD	rA	rB	0	0	1	1	1	0	1	0	1	1	1	X	
msgclr	0	1	1	1	1	1	///	///	rB	0	0	1	1	1	0	1	1	1	0	/	X	**E.PC**
dcbtst	0	1	1	1	1	1	TH	rA	rB	0	0	1	1	1	1	0	1	1	0	/	X	
stbux	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	1	1	0	1	1	1	/	X	
bpermd	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	1	1	1	1	0	0	/	X	**64**
dcbtstep	0	1	1	1	1	1	TH	rA	rB	0	0	1	1	1	1	1	1	1	1	/	X	**E.PD**
lvepxl	0	1	1	1	1	1	vD	rA	rB	0	1	0	0	0	0	0	1	1	1	/	X	**E.PD, V**
add	0	1	1	1	1	1	rD	rA	rB	0	1	0	0	0	0	1	0	1	0	0	X	
add.	0	1	1	1	1	1	rD	rA	rB	0	1	0	0	0	0	1	0	1	0	1	X	
ehpriv	0	1	1	1	1	1	OC			0	1	0	0	0	0	1	1	1	0	/	XL	**E.HV**
dcbt	0	1	1	1	1	1	TH	rA	rB	0	1	0	0	0	1	0	1	1	0	/	X	
lhzx	0	1	1	1	1	1	rD	rA	rB	0	1	0	0	0	1	0	1	1	1	/		
eqv	0	1	1	1	1	1	rD	rA	rB	0	1	0	0	0	1	1	1	0	0	0	X	
eqv.	0	1	1	1	1	1	rD	rA	rB	0	1	0	0	0	1	1	1	0	0	1	X	
lhepx	0	1	1	1	1	1	rD	rA	rB	0	1	0	0	0	1	1	1	1	1	/	X	**E.PD**
lvepx	0	1	1	1	1	1	vD	rA	rB	0	1	0	0	1	0	0	1	1	1	/	X	**E.PD, V**
lhzux	0	1	1	1	1	1	rD	rA	rB	0	1	0	0	1	1	0	1	1	1	/	X	
xor	0	1	1	1	1	1	rS	rA	rB	0	1	0	0	1	1	1	1	0	0	0	X	
xor.	0	1	1	1	1	1	rS	rA	rB	0	1	0	0	1	1	1	1	0	0	1	X	
dcbtep	0	1	1	1	1	1	TH	rA	rB	0	1	0	0	1	1	1	1	1	1	/	X	**E.PD**
mfdcr	0	1	1	1	1	1	rD	DCRN5–9	DCRN0–4	0	1	0	1	0	0	0	0	1	1	/	XFX	**E.DC**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
mfpmr	0	1	1	1	1	1	rD					PMRN5–9					PMRN0–4					0	1	0	1	0	0	1	1	1	0	/	XFX	**E.PM**
mfspr	0	1	1	1	1	1	rD					SPRN[5–9]					SPRN[0–4]					0	1	0	1	0	1	0	0	1	1	/	XFX	
lwax	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	1	0	1	0	1	/	X	**64**
dst	0	1	1	1	1	1	0	//		STRM		rA					rB					0	1	0	1	0	1	0	1	1	0	/	X	**V**
dstt	0	1	1	1	1	1	1	//		STRM		rA					rB					0	1	0	1	0	1	0	1	1	0	/	X	**V**
lhax	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	1	0	1	1	1	/	X	
lvxl	0	1	1	1	1	1	vD					rA					rB					0	1	0	1	1	0	0	1	1	1	/	X	**V**
mftb	0	1	1	1	1	1	rD					TBRN[5–9]					TBRN[0–4]					0	1	0	1	1	1	0	0	1	1	0	XFX	
lwaux	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	1	1	0	1	0	1	/	X	**64**
dstst	0	1	1	1	1	1	0	//		STRM		rA					rB					0	1	0	1	1	1	0	1	1	0	/	X	**V**
dststt	0	1	1	1	1	1	1	//		STRM		rA					rB					0	1	0	1	1	1	0	1	1	0	/	X	**V**
lhaux	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	1	1	0	1	1	1	/	X	
popcntw	0	1	1	1	1	1	rS					rA					///					0	1	0	1	1	1	1	0	1	0	/	X	
dcblc	0	1	1	1	1	1	CT					rA					rB					0	1	1	0	0	0	0	1	1	0	/	X	**E.CL**
sthx	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	0	1	1	1	/	X	
orc	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	1	0	0	0	X	
orc.	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	0	1	1	1	0	0	1	X	
sthux	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	1	1	0	1	1	1	/	X	
or	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	1	1	1	1	0	0	0	X	
or.	0	1	1	1	1	1	rS					rA					rB					0	1	1	0	1	1	1	1	0	0	1	X	
mtdcr	0	1	1	1	1	1	rS					DCRN5–9					DCRN0–4					0	1	1	1	0	0	0	0	1	1	/	XFX	**E.DC**
divdu	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	0	0	1	0	0	1	0	**X**	**64**
divdu.	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	0	0	1	0	0	1	1	**X**	**64**
divwu	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	0	0	1	0	1	1	0	X	
divwu.	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	0	0	1	0	1	1	1	X	
mtpmr	0	1	1	1	1	1	rS					PMRN5–9					PMRN0–4					0	1	1	1	0	0	1	1	1	0	/	XFX	**E.PM**
mtspr	0	1	1	1	1	1	rS					SPRN[5–9]					SPRN[0–4]					0	1	1	1	0	1	0	0	1	1	/	XFX	
dcbi	0	1	1	1	1	1	///					rA					rB					0	1	1	1	0	1	0	1	1	0	/	X	**Embedded**
dsn	0	1	1	1	1	1	///					rA					rB					0	1	1	1	0	1	1	0	0	1	/	X	**DS**
nand	0	1	1	1	1	1	rS					rA					rB					0	1	1	1	0	1	1	1	0	0	0	X	
nand.	0	1	1	1	1	1	rS					rA					rB					0	1	1	1	0	1	1	1	0	0	1	X	
icbtls	0	1	1	1	1	1	CT					rA					rB					0	1	1	1	1	0	0	1	1	0	/	X	**E.CL**
stvxl	0	1	1	1	1	1	vS					rA					rB					0	1	1	1	1	0	0	1	1	1	/	X	**V**
divd	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	1	0	1	0	0	1	0	**X**	**64**
divd.	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	1	0	1	0	0	1	1	**X**	**64**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
divw	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	1	0	1	0	1	1	0	X	
divw.	0	1	1	1	1	1	rD					rA					rB					0	1	1	1	1	0	1	0	1	1	1	X	
popcntd	0	1	1	1	1	1	rS					rA					///					0	1	1	1	1	1	1	0	1	0	/	X	**64**
cmpb	0	1	1	1	1	1	rS					rA					rB					0	1	1	1	1	1	1	1	0	0	/	X	
mcrxr	0	1	1	1	1	1	crD			///												1	0	0	0	0	0	0	0	0	0	/	X	
lbdx	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	0	0	1	1	/	X	**DS**
subfco	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	0	0	0	X	
subfco.	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	0	0	1	X	
addco	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	1	0	0	X	
addco.	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	1	0	1	X	
ldbrx	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	0	0	/	X	**64**
lfsx	0	1	1	1	1	1	frD					rA					rB					1	0	0	0	0	1	0	1	1	1	/	X	
lwbrx	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	1	0	/	X	
srw	0	1	1	1	1	1	rS					rA					rB					1	0	0	0	0	1	1	0	0	0	0	X	
srw.	0	1	1	1	1	1	rS					rA					rB					1	0	0	0	0	1	1	0	0	0	1	X	
srd	0	1	1	1	1	1	rS					rA					rB					1	0	0	0	0	1	1	0	1	1	0	X	**64**
srd.	0	1	1	1	1	1	rS					rA					rB					1	0	0	0	0	1	1	0	1	1	1	X	**64**
lhdx	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	0	0	1	0	/	X	**DS**
subfo	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	0	0	0	0	X	
subfo.	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	0	0	0	1	X	
tlbsync	0	1	1	1	1	1	0	///														1	0	0	0	1	1	0	1	1	0	/	X	**Embedded**
lfsux	0	1	1	1	1	1	frD					rA					rB					1	0	0	0	1	1	0	1	1	1	/	X	**FP**
lwdx	0	1	1	1	1	1	rD					rA					rB					1	0	0	1	0	0	0	0	1	1	/	X	**DS**
sync	0	1	1	1	1	1	///			L		/	E				///					1	0	0	1	0	1	0	1	1	0	/	X	
lfdx	0	1	1	1	1	1	frD					rA					rB					1	0	0	1	0	1	0	1	1	1	/	X	**FP**
lfdepx	0	1	1	1	1	1	frD					rA					rB					1	0	0	1	0	1	1	1	1	1	/	X	**E.PD, FP**
lddx	0	1	1	1	1	1	rD					rA					rB					1	0	0	1	1	0	0	0	1	1	/	X	**DS, 64**
nego	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	1	0	0	0	0	X	
nego.	0	1	1	1	1	1	rD					rA					///					1	0	0	1	1	0	1	0	0	0	1	X	
lfdux	0	1	1	1	1	1	frD					rA					rB					1	0	0	1	1	1	0	1	1	1	/	X	**FP**
stbdx	0	1	1	1	1	1	rS					rA					rB					1	0	1	0	0	0	0	0	1	1	/	X	**DS**
subfeo	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	0	0	0	X	
subfeo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	0	0	1	X	
addeo	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	1	0	0	X	
addeo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	1	0	1	X	

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
stdbrx	0	1	1	1	1	1	rS					rA					rB					1	0	1	0	0	1	0	1	0	0	/	X	64
stwbrx	0	1	1	1	1	1	rS					rA					rB					1	0	1	0	0	1	0	1	1	0	/	X	
stfsx	0	1	1	1	1	1	frS					rA					rB					1	0	1	0	0	1	0	1	1	1	/	X	FP
sthdx	0	1	1	1	1	1	rS					rA					rB					1	0	1	0	1	0	0	0	1	1	/	X	DS
stbcx.	0	1	1	1	1	1	rS					rA					rB					1	0	1	0	1	1	0	1	1	0	1	X	ER
stfsux	0	1	1	1	1	1	frS					rA					rB					1	0	1	0	1	1	0	1	1	1	/	X	FP
stwdx	0	1	1	1	1	1	rS					rA					rB					1	0	1	1	0	0	0	0	1	1	/	X	DS
subfzeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	0	0	0	X	
subfzeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	0	0	1	X	
addzeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	1	0	0	X	
addzeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	1	0	1	X	
sthcx.	0	1	1	1	1	1	rS					rA					rB					1	0	1	1	0	1	0	1	1	0	1	X	ER
stfdx	0	1	1	1	1	1	frS					rA					rB					1	0	1	1	0	1	0	1	1	1	/	X	FP
stfdepx	0	1	1	1	1	1	frS					rA					rB					1	0	1	1	0	1	1	1	1	1	/	X	E.PD, FP
stddx	0	1	1	1	1	1	rS					rA					rB					1	0	1	1	1	0	0	0	1	1	/	X	DS, 64
subfmeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	0	0	0	X	
subfmeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	0	0	1	X	
mulldo	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	0	1	0	X	64
mulldo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	0	1	1	X	64
addmeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	1	0	0	X	
addmeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	1	0	1	X	
mullwo	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	1	1	0	X	
mullwo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	1	1	0	1	0	1	1	1	X	
dcba	0	1	1	1	1	1	///				0	rA					rB					1	0	1	1	1	1	0	1	1	0	/	X	
dcbal	0	1	1	1	1	1	///				1	rA					rB					1	0	1	1	1	1	0	1	1	0	/	X	DEO
stfdux	0	1	1	1	1	1	frS					rA					rB					1	0	1	1	1	1	0	1	1	1	/	X	FP
stvepxl	0	1	1	1	1	1	vS					rA					rB					1	1	0	0	0	0	0	1	1	1	/	X	E.PD, V
addo	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	0	1	0	1	0	0	X	
addo.	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	0	1	0	1	0	1	X	
tlbivax	0	1	1	1	1	1	0	///				rA					rB					1	1	0	0	0	1	0	0	1	0	/	X	Embedded
lhbrx	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	1	0	1	1	0	/	X	
sraw	0	1	1	1	1	1	rS					rA					rB					1	1	0	0	0	1	1	0	0	0	0	X	
sraw.	0	1	1	1	1	1	rS					rA					rB					1	1	0	0	0	1	1	0	0	0	1	X	
srad	0	1	1	1	1	1	rS					rA					rB					1	1	0	0	0	1	1	0	1	0	0	X	64
srad.	0	1	1	1	1	1	rS					rA					rB					1	1	0	0	0	1	1	0	1	0	1	X	64

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
evlddepx	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	1	1	1	1	1	/	X	**E.PD, SP**
mtvscr	0	1	1	1	1	1	///					///					vB					1	1	0	0	1	0	0	0	1	0	0	VX	**V**
lfddx	0	1	1	1	1	1	frD					rA					rB					1	1	0	0	1	0	0	0	1	1	/	X	**DS, FP**
stvepx	0	1	1	1	1	1	vS					rA					rB					1	1	0	0	1	0	0	1	1	1	/	X	**E.PD, V**
dss	0	1	1	1	1	1	0	//		STRM		///					///					1	1	0	0	1	1	0	1	1	0	/	X	**V**
dssall	0	1	1	1	1	1	1	//		STRM		///					///					1	1	0	0	1	1	0	1	1	0	/	X	**V**
srawi	0	1	1	1	1	1	rS					rA					SH					1	1	0	0	1	1	1	0	0	0	0	X	
srawi.	0	1	1	1	1	1	rS					rA					SH					1	1	0	0	1	1	1	0	0	0	1	X	
sradi	0	1	1	1	1	1	rS					rA					sh1–5					1	1	0	0	1	1	1	0	1	sh0	0	XS	**64**
sradi.	0	1	1	1	1	1	rS					rA					sh1–5					1	1	0	0	1	1	1	0	1	sh0	1	XS	**64**
mbar	0	1	1	1	1	1	MO					///										1	1	0	1	0	1	0	1	1	0	/	X	**Embedded**
tlbsx	0	1	1	1	1	1	0	///				rA					rB					1	1	1	0	0	1	0	0	1	0	/	X	**Embedded**
sthbrx	0	1	1	1	1	1	rS					rA					rB					1	1	1	0	0	1	0	1	1	0	/	X	
extsh	0	1	1	1	1	1	rS					rA					///					1	1	1	0	0	1	1	0	1	0	0	X	
extsh.	0	1	1	1	1	1	rS					rA					///					1	1	1	0	0	1	1	0	1	0	1	X	
evstddepx	0	1	1	1	1	1	rS					rA					rB					1	1	1	0	0	1	1	1	1	1	/	X	**E.PD, SP**
stfddx	0	1	1	1	1	1	frS					rA					rB					1	1	1	0	1	0	0	0	1	1	/	X	**DS, FP**
tlbre	0	1	1	1	1	1	0	///														1	1	1	0	1	1	0	0	1	0	/	X	**Embedded**
extsb	0	1	1	1	1	1	rS					rA					///					1	1	1	0	1	1	1	0	1	0	0	X	
extsb.	0	1	1	1	1	1	rS					rA					///					1	1	1	0	1	1	1	0	1	0	1	X	
divduo	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	0	0	1	0	0	1	0	**X**	**64**
divduo.	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	0	0	1	0	0	1	1	**X**	**64**
divwuo	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	0	0	1	0	1	1	0	X	
divwuo.	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	0	0	1	0	1	1	1	X	
tlbwe	0	1	1	1	1	1	0	///														1	1	1	1	0	1	0	0	1	0	/	X	**Embedded**
icbi	0	1	1	1	1	1	///					rA					rB					1	1	1	1	0	1	0	1	1	0	/	X	
stfiwx	0	1	1	1	1	1	frS					rA					rB					1	1	1	1	0	1	0	1	1	1	/	X	**FP**
extsw	0	1	1	1	1	1	rS					rA					///					1	1	1	1	0	1	1	0	1	0	0	X	**64**
extsw.	0	1	1	1	1	1	rS					rA					///					1	1	1	1	0	1	1	0	1	0	1	X	**64**
icbiep	0	1	1	1	1	1	///					rA					rB					1	1	1	1	0	1	1	1	1	1	/	X	**E>PD**
divdo	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	1	0	1	0	0	1	0	**X**	**64**
divdo.	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	1	0	1	0	0	1	1	**X**	**64**
divwo	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	1	0	1	0	1	1	0	X	
divwo.	0	1	1	1	1	1	rD					rA					rB					1	1	1	1	1	0	1	0	1	1	1	X	
dcbz	0	1	1	1	1	1	///				0	rA					rB					1	1	1	1	1	1	0	1	1	0	/	X	

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
dcbzl	0	1	1	1	1	1	///				1	rA					rB					1	1	1	1	1	1	0	1	1	0	/	X	DEO
dcbzep	0	1	1	1	1	1	///				0	rA					rB					1	1	1	1	1	1	1	1	1	1	/	X	E.PD
dcbzlep	0	1	1	1	1	1	///				1	rA					rB					1	1	1	1	1	1	1	1	1	1	/	X	DEO, E.PD
lwz	1	0	0	0	0	0	rD					rA					D																D	
lwzu	1	0	0	0	0	1	rD					rA					D																D	
lbz	1	0	0	0	1	0	rD					rA					D																D	
lbzu	1	0	0	0	1	1	rD					rA					D																D	
stw	1	0	0	1	0	0	rS					rA					D																D	
stwu	1	0	0	1	0	1	rS					rA					D																D	
stb	1	0	0	1	1	0	rS					rA					D																D	
stbu	1	0	0	1	1	1	rS					rA					D																D	
lhz	1	0	1	0	0	0	rD					rA					D																D	
lhzu	1	0	1	0	0	1	rD					rA					D																D	
lha	1	0	1	0	1	0	rD					rA					D																D	
lhau	1	0	1	0	1	1	rD					rA					D																D	
sth	1	0	1	1	0	0	rS					rA					D																D	
sthu	1	0	1	1	0	1	rS					rA					D																D	
lmw	1	0	1	1	1	0	rD					rA					D																D	
stmw	1	0	1	1	1	1	rS					rA					D																D	
lfs	1	1	0	0	0	0	frD					rA					D																D	FP
lfsu	1	1	0	0	0	1	frD					rA					D																D	FP
lfd	1	1	0	0	1	0	frD					rA					D																D	FP
lfdu	1	1	0	0	1	1	frD					rA					D																D	FP
stfs	1	1	0	1	0	0	frS					rA					D																D	FP
stfsu	1	1	0	1	0	1	frS					rA					D																D	FP
stfd	1	1	0	1	1	0	frS					rA					D																D	FP
stfdu	1	1	0	1	1	1	frS					rA					D																D	FP
ld	1	1	1	0	1	0	rD					rA					DS														0	0	DS	64
ldu	1	1	1	0	1	0	rD					rA					DS														0	1	DS	64
lwa	1	1	1	0	1	0	rD					rA					DS														1	0	DS	64
fdivs	1	1	1	0	1	1	frD					frA					frB					///					1	0	0	1	0	0	A	FP
fdivs.	1	1	1	0	1	1	frD					frA					frB					///					1	0	0	1	0	1	A	FP.R
fsubs	1	1	1	0	1	1	frD					frA					frB					///					1	0	1	0	0	0	A	FP
fsubs.	1	1	1	0	1	1	frD					frA					frB					///					1	0	1	0	0	1	A	FP.R
fadds	1	1	1	0	1	1	frD					frA					frB					///					1	0	1	0	1	0	A	FP

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
fadds.	1	1	1	0	1	1	frD					frA					frB					///					1	0	1	0	1	1	A	**FP.R**
fres	1	1	1	0	1	1	frD					///					frB					///					1	1	0	0	0	0	A	**FP**
fres.	1	1	1	0	1	1	frD					///					frB					///					1	1	0	0	0	1	A	**FP.R**
fmuls	1	1	1	0	1	1	frD					frA					///					frC					1	1	0	0	1	0	A	**FP**
fmuls.	1	1	1	0	1	1	frD					frA					///					frC					1	1	0	0	1	1	A	**FP.R**
fmsubs	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	0	0	0	A	**FP**
fmsubs.	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	0	0	1	A	**FP.R**
fmadds	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	0	1	0	A	**FP**
fmadds.	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	0	1	1	A	**FP.R**
fnmsubs	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	1	0	0	A	**FP**
fnmsubs.	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	1	0	1	A	**FP.R**
fnmadds	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	1	1	0	A	**FP**
fnmadds.	1	1	1	0	1	1	frD					frA					frB					frC					1	1	1	1	1	1	A	**FP.R**
std	1	1	1	1	1	0	rS					rA					DS														0	0	DS	**64**
stdu	1	1	1	1	1	0	rS					rA					DS														0	1	DS	**64**
fcmpu	1	1	1	1	1	1	crD			//		frA					frB					0	0	0	0	0	0	0	0	0	0	/	X	**FP**
frsp	1	1	1	1	1	1	frD					///					frB					0	0	0	0	0	0	1	1	0	0	0	X	**FP**
frsp.	1	1	1	1	1	1	frD					///					frB					0	0	0	0	0	0	1	1	0	0	1	X	**FP.R**
fctiw	1	1	1	1	1	1	frD					///					frB					0	0	0	0	0	0	1	1	1	0	0	X	**FP**
fctiw.	1	1	1	1	1	1	frD					///					frB					0	0	0	0	0	0	1	1	1	0	1	X	**FP.R**
fctiwz	1	1	1	1	1	1	frD					///					frB					0	0	0	0	0	0	1	1	1	1	0	X	**FP**
fctiwz.	1	1	1	1	1	1	frD					///					frB					0	0	0	0	0	0	1	1	1	1	1	X	**FP.R**
fdiv	1	1	1	1	1	1	frD					frA					frB					///					1	0	0	1	0	0	A	**FP**
fdiv.	1	1	1	1	1	1	frD					frA					frB					///					1	0	0	1	0	1	A	**FP.R**
fadd	1	1	1	1	1	1	frD					frA					frB					///					1	0	1	0	1	0	A	**FP**
fsub	1	1	1	1	1	1	frD					frA					frB					///					1	0	1	0	0	0	A	**FP**
fsub.	1	1	1	1	1	1	frD					frA					frB					///					1	0	1	0	0	1	A	**FP.R**
fadd.	1	1	1	1	1	1	frD					frA					frB					///					1	0	1	0	1	1	A	**FP.R**
fsel	1	1	1	1	1	1	frD					frA					frB					frC					1	0	1	1	1	0	A	**FP**
fsel.	1	1	1	1	1	1	frD					frA					frB					frC					1	0	1	1	1	1	A	**FP.R**
fmul	1	1	1	1	1	1	frD					frA					///					frC					1	1	0	0	1	0	A	**FP**
fmul.	1	1	1	1	1	1	frD					frA					///					frC					1	1	0	0	1	1	A	**FP.R**
frsqrte	1	1	1	1	1	1	frD					///					frB					///					1	1	0	1	0	0	A	**FP**
frsqrte.	1	1	1	1	1	1	frD					///					frB					///					1	1	0	1	0	1	A	**FP.R**
fmsub	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	0	0	0	A	**FP**

Table B-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Category
fmsub.	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	0	0	1	A	**FP.R**
fmadd	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	0	1	0	A	**FP**
fmadd.	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	0	1	1	A	**FP.R**
fnmsub	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	1	0	0	A	**FP**
fnmsub.	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	1	0	1	A	**FP.R**
fnmadd	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	1	1	0	A	**FP**
fnmadd.	1	1	1	1	1	1	frD					frA					frB					frC					1	1	1	1	1	1	A	**FP.R**
fcmpo	1	1	1	1	1	1	crD			//		frA					frB					0	0	0	0	1	0	0	0	0	0	/	X	**FP**
mtfsb1	1	1	1	1	1	1	crbD					///										0	0	0	0	1	0	0	1	1	0	0	X	**FP**
mtfsb1.	1	1	1	1	1	1	crbD					///										0	0	0	0	1	0	0	1	1	0	1	X	**FP.R**
fneg	1	1	1	1	1	1	frD					///					frB					0	0	0	0	1	0	1	0	0	0	0	X	**FP**
fneg.	1	1	1	1	1	1	frD					///					frB					0	0	0	0	1	0	1	0	0	0	1	X	**FP.R**
mcrfs	1	1	1	1	1	1	crD			//		crfS			///							0	0	0	1	0	0	0	0	0	0	/	X	**FP**
mtfsb0	1	1	1	1	1	1	crbD					///										0	0	0	1	0	0	0	1	1	0	0	X	**FP**
mtfsb0.	1	1	1	1	1	1	crbD					///										0	0	0	1	0	0	0	1	1	0	1	X	**FP.R**
fmr	1	1	1	1	1	1	frD					///					frB					0	0	0	1	0	0	1	0	0	0	0	X	**FP**
fmr.	1	1	1	1	1	1	frD					///					frB					0	0	0	1	0	0	1	0	0	0	1	X	**FP.R**
mtfsfi	1	1	1	1	1	1	crD			///		///				W	IMM				/	0	0	1	0	0	0	0	1	1	0	0	X	**FP**
mtfsfi.	1	1	1	1	1	1	crD			///		///				W	IMM				/	0	0	1	0	0	0	0	1	1	0	1	X	**FP.R**
fnabs	1	1	1	1	1	1	frD					///					frB					0	0	1	0	0	0	1	0	0	0	0	X	**FP**
fnabs.	1	1	1	1	1	1	frD					///					frB					0	0	1	0	0	0	1	0	0	0	1	X	**FP.R**
fabs	1	1	1	1	1	1	frD					///					frB					0	1	0	0	0	0	1	0	0	0	0	X	**FP**
fabs.	1	1	1	1	1	1	frD					///					frB					0	1	0	0	0	0	1	0	0	0	1	X	**FP.R**
mffs	1	1	1	1	1	1	frD					///										1	0	0	1	0	0	0	1	1	1	0	X	**FP**
mffs.	1	1	1	1	1	1	frD					///										1	0	0	1	0	0	0	1	1	1	1	X	**FP.R**
mtfsf	1	1	1	1	1	1	L	FM								W	frB					1	0	1	1	0	0	0	1	1	1	0	XFX	**FP**
mtfsf.	1	1	1	1	1	1	L	FM								W	frB					1	0	1	1	0	0	0	1	1	1	1	XFX	**FP.R**
fctid	1	1	1	1	1	1	frD					///					frB					1	1	0	0	1	0	1	1	1	0	0	X	**FP**
fctid.	1	1	1	1	1	1	frD					///					frB					1	1	0	0	1	0	1	1	1	0	1	X	**FP.R**
fctidz	1	1	1	1	1	1	frD					///					frB					1	1	0	0	1	0	1	1	1	1	0	X	**FP**
fctidz.	1	1	1	1	1	1	frD					///					frB					1	1	0	0	1	0	1	1	1	1	1	X	**FP.R**
fcfid	1	1	1	1	1	1	frD					///					frB					1	1	0	1	0	0	1	1	1	0	0	X	**FP**
fcfid.	1	1	1	1	1	1	frD					///					frB					1	1	0	1	0	0	1	1	1	0	1	X	**FP.R**

[1] d = UIMM * 8

[2] d = UIMM * 4

[3] d = UIMM * 2

Appendix C
Simplified Mnemonics

This chapter describes simplified mnemonics, which are provided for easier coding of assembly language programs. Simplified mnemonics are defined for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions defined by the EIS and Power ISA™.

C.1 Simplified Mnemonics Overview

Simplified (or extended) mnemonics allow an assembly-language programmer to program using more intuitive mnemonics and symbols than the instructions and syntax defined by the instruction set architecture. For example, to code the conditional call "branch to an absolute target if CR4 specifies a greater than condition, setting the LR without simplified mnemonics," the programmer would write the branch conditional instruction, **bc 12,17,***target*. The simplified mnemonic, branch if greater than, **bgt cr4,***target*, incorporates the conditions. Not only is it easier to remember the symbols than the numbers when programming, it is also easier to interpret simplified mnemonics when reading existing code.

Although the Power ISA documents include a set of simplified mnemonics, these are not a formal part of the architecture, but rather a recommendation for assemblers that support the instruction set.

Many simplified mnemonics have been added to those originally included in the architecture documentation. Some assemblers created their own, and others have been added to support extensions to the instruction set. Simplified mnemonics have been added for new architecturally defined and new implementation-specific special-purpose registers (SPRs). These simplified mnemonics are described only in a very general way.

C.2 Subtract Simplified Mnemonics

This section describes simplified mnemonics for subtract instructions.

C.2.1 Subtract Immediate

There is no subtract immediate instruction, however, its effect is achieved by negating the immediate operand of an Add Immediate instruction, **addi**. Simplified mnemonics include this negation, making the intent of the computation more clear. These are listed in this table.

Table C-1. Subtract Immediate Simplified Mnemonics

Simplified Mnemonic	Standard Mnemonic
subi rD,**r**A,value	**addi r**D,**r**A,–value
subis rD,**r**A,value	**addis r**D,**r**A,–value

Table C-1. Subtract Immediate Simplified Mnemonics (continued)

Simplified Mnemonic	Standard Mnemonic
subic rD,**r**A,value	**addic r**D,**r**A,–value
subic. rD,**r**A,value	**addic. r**D,**r**A,–value

C.2.2 Subtract

Subtract from instructions subtract the second operand (**r**A) from the third (**r**B). The simplified mnemonics in this table use the more common order in which the third operand is subtracted from the second.

Table C-2. Subtract Simplified Mnemonics

Simplified Mnemonic	Standard Mnemonic[1]
sub[**o**][.] **r**D,**r**A,**r**B	**subf**[**o**][.] **r**D,**r**B,**r**A
subc[**o**][.] **r**D,**r**A,**r**B	**subfc**[**o**][.] **r**D,**r**B,**r**A

[1] **r**D,**r**B,**r**A is not the standard order for the operands. The order of **r**B and **r**A is reversed to show the equivalent behavior of the simplified mnemonic.

C.3 Rotate and Shift Simplified Mnemonics

Rotate and shift instructions provide powerful, general ways to manipulate register contents, but can be difficult to understand. Simplified mnemonics are provided for the following operations:

- Extract—Select a field of *n* bits starting at bit position *b* in the source register; left or right justify this field in the target register; clear all other bits of the target register.
- Insert—Select a left- or right-justified field of *n* bits in the source register; insert this field starting at bit position *b* of the target register; leave other bits of the target register unchanged. (No simplified mnemonic is provided for insertion of a left-justified field when operating on double words, because such an insertion requires more than one instruction.)
- Rotate—Rotate the contents of a register right or left *n* bits without masking.
- Shift—Shift the contents of a register right or left *n* bits, clearing vacated bits (logical shift).
- Clear—Clear the leftmost or rightmost *n* bits of a register.
- Clear left and shift left—Clear the leftmost *b* bits of a register, then shift the register left by *n* bits. This operation can be used to scale a (known nonnegative) array index by the width of an element.

C.3.1 Operations on Words

The simplified mnemonics in this table can be coded with a dot (.) suffix to cause the Rc bit to be set in the underlying instruction.

Table C-3. Word Rotate and Shift Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Extract and left justify word immediate	**extlwi r**A,**r**S,*n*,*b* (*n* > 0)	**rlwinm r**A,**r**S,*b*,**0**,*n* – 1
Extract and right justify word immediate	**extrwi r**A,**r**S,*n*,*b* (*n* > 0)	**rlwinm r**A,**r**S,*b* + *n*, 32 – *n*,**31**
Insert from left word immediate	**inslwi r**A,**r**S,*n*,*b* (*n* > 0)	**rlwimi r**A,**r**S,32 – *b*,*b*,(*b* + *n*) – 1
Insert from right word immediate	**insrwi r**A,**r**S,*n*,*b* (*n* > 0)	**rlwimi r**A,**r**S,32 – (*b* + *n*),*b*,(*b* + *n*) – 1
Rotate left word immediate	**rotlwi r**A,**r**S,*n*	**rlwinm r**A,**r**S,*n*,**0**,**31**
Rotate right word immediate	**rotrwi r**A,**r**S,*n*	**rlwinm r**A,**r**S,32 – *n*,**0**,**31**
Rotate word left	**rotlw r**A,**r**S,**r**B	**rlwnm r**A,**r**S,**r**B,**0**,**31**
Shift left word immediate	**slwi r**A,**r**S,*n* (*n* < 32)	**rlwinm r**A,**r**S,*n*,**0**,31 – *n*
Shift right word immediate	**srwi r**A,**r**S,*n* (*n* < 32)	**rlwinm r**A,**r**S,32 – *n*,*n*,**31**
Clear left word immediate	**clrlwi r**A,**r**S,*n* (*n* < 32)	**rlwinm r**A,**r**S,**0**,*n*,**31**
Clear right word immediate	**clrrwi r**A,**r**S,*n* (*n* < 32)	**rlwinm r**A,**r**S,**0**,**0**,31 – *n*
Clear left and shift left word immediate	**clrlslwi r**A,**r**S,*b*,*n* (*n* ≤ *b* ≤ 31)	**rlwinm r**A,**r**S,*n*,*b* – *n*,31 – *n*

Examples using word mnemonics are as follow:

1. Extract the sign bit (bit 0) of **r**S and place the result right-justified into **r**A.
   extrwi rA**,r**S**,1,0** equivalent to **rlwinm r**A**,r**S**,1,31,31**
2. Insert the bit extracted in (1) into the sign bit (bit 0) of **r**B.
   insrwi rB**,r**A**,1,0** equivalent to **rlwimi r**B**,r**A**,31,0,0**
3. Shift the contents of **r**A left 8 bits.
   slwi rA**,r**A**,8** equivalent to **rlwinm r**A**,r**A**,8,0,23**
4. Clear the high-order 16 bits of **r**S and place the result into **r**A.
   clrlwi rA**,r**S**,16** equivalent to **rlwinm r**A**,r**S**,0,16,31**

C.3.2 Operations on Double-words

The simplified mnemonics in Table C-3 can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

Table C-4. Double-word Rotate and Shift Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Extract and left justify double-word immediate	**extldi r**A,**r**S,*n*,*b* (*n* > 0)	**rldicr r**A,**r**S,*b*,*n* – 1
Extract and right justify double-word immediate	**extrdi r**A,**r**S,*n*,*b* (*n* > 0)	**rldicl r**A,**r**S,*b* + *n*, 64 – *n*

Table C-4. Double-word Rotate and Shift Simplified Mnemonics (continued)

Operation	Simplified Mnemonic	Equivalent to:
Insert from right double-word immediate	**insrdi r**A,**r**S,*n*,*b* (*n* > 0)	**rldimi r**A,**r**S,64 – (*b* + *n*),*b*
Rotate left double-word immediate	**rotldi r**A,**r**S,*n*	**rldicl r**A,**r**S,*n*,**0**
Rotate right double-word immediate	**rotrdi r**A,**r**S,*n*	**rldicl r**A,**r**S,64 – *n*,**0**
Rotate double-word left	**rotld r**A,**r**S,**r**B	**rldcl r**A,**r**S,**r**B,**0**
Shift left double-word immediate	**sldi r**A,**r**S,*n* (*n* < 64)	**rldicr r**A,**r**S,*n*,**63** – *n*
Shift right double-word immediate	**srdi r**A,**r**S,*n* (*n* < 64)	**rldicl r**A,**r**S,64 – *n*,*n*
Clear left double-word immediate	**clrldi r**A,**r**S,*n* (*n* < 64)	**rldicl r**A,**r**S,**0**,*n*
Clear right double-word immediate	**clrrdi r**A,**r**S,*n* (*n* < 64)	**rldicr r**A,**r**S,**0**,**63** – *n*
Clear left and shift left double-word immediate	**clrlsldi r**A,**r**S,*b*,*n* (*n* ≤ *b* ≤ 63)	**rldic r**A,**r**S,*n*,*b* – *n*

Examples using word mnemonics follow:

1. Extract the sign bit (bit 0) of **r**S and place the result right-justified into **r**A.
   extrdi rA,**r**S,**1**,**0** equivalent to **rldicl r**A,**r**S,**1**,**63**
2. Insert the bit extracted in (1) into the sign bit (bit 0) of **r**B.
   insrdi rB,**r**A,**1**,**0** equivalent to **rldimi r**B,**r**A,**63**,**0**
3. Shift the contents of **r**A left 8 bits.
   sldi rA,**r**A,**8** equivalent to **rldicr r**A,**r**A,**8**,**55**
4. Clear the high-order 32 bits of **r**S and place the result into **r**A.
   clrldi rA,**r**S,**32** equivalent to **rldicl r**A,**r**S,**0**,**32**

C.4 Branch Instruction Simplified Mnemonics

Branch conditional instructions can be coded with the operations, a condition to be tested, and a prediction, as part of the instruction mnemonic rather than as numeric operands (the BO and BI operands).

This table shows the four general types of branch instructions. Simplified mnemonics are defined only for branch instructions that include BO and BI operands; there is no need to simplify unconditional branch mnemonics.

Table C-5. Branch Instructions

Instruction Name	Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI

The BO and BI operands correspond to two fields in the instruction opcode, as this figure shows for Branch Conditional (**bc**, **bca**, **bcl**, and **bcla**) instructions.

0					5	6 10	11 15	16 29	30	31
0	1	0	0	0	0	BO	BI	BD	AA	LK

Figure C-1. Branch Conditional (bc) Instruction Format

The BO operand specifies branch operations that involve decrementing CTR. It is also used to determine whether testing a CR bit causes a branch to occur if the condition is true or false.

The BI operand identifies a CR bit to test (whether a comparison is less than or greater than, for example). The simplified mnemonics avoid the need to memorize the numerical values for BO and BI.

For example, **bc 16,0,***target* is a conditional branch that, as a BO value of 16 (0b1_0000) indicates, decrements CTR, then branches if the decremented CTR is not zero. The operation specified by BO is abbreviated as **d** (for decrement) and **nz** (for not zero), which replace the **c** in the original mnemonic; so the simplified mnemonic for **bc** becomes **bdnz**. The branch does not depend on a condition in the CR, so BI can be eliminated, reducing the expression to **bdnz** *target*.

In addition to CTR operations, the BO operand provides an optional prediction bit and a true or false indicator can be added. For example, if the previous instruction should branch only on an equal condition in CR0, the instruction becomes **bc 8,2,***target*. To incorporate a true condition, the BO value becomes 8 (as shown in Table C-7); the CR0 equal field is indicated by a BI value of 2 (as shown in Table C-8). Incorporating the branch-if-true condition adds a '**t**' to the simplified mnemonic, **bdnzt.** The equal condition, that is specified by a BI value of 2 (indicating the EQ bit in CR0) is replaced by the **eq** symbol. Using the simplified mnemonic and the **eq** operand, the expression becomes **bdnzt eq,***target*.

This example tests CR0[EQ]; however, to test the equal condition in CR5 (CR bit 22), the expression becomes **bc 8,22,***target*. The BI operand of 22 indicates CR[22] (CR5[2], or BI field 0b10110), as shown in Table C-8. This can be expressed as the simplified mnemonic. **bdnzt 4 * cr5 + eq,***target*.

The notation, **4 * cr5 + eq** may at first seem awkward, but it eliminates computing the value of the CR bit. It can be seen that (4 * 5) + 2 = 22. Note that although 32-bit registers in Power ISA processors are numbered 32–63, only values 0–31 are valid (or possible) for BI operands. The encoding of the field in the instruction uses numbering from 0 - 31 and the instruction converts this into the architecturally described bit number by adding 32.automatically translates the bit values; specifying a BI value of 22 selects bit 54.

C.4.1 Key Facts about Simplified Branch Mnemonics

The following key points are helpful in understanding how to use simplified branch mnemonics:

- All simplified branch mnemonics eliminate the BO operand, so if any operand is present in a branch simplified mnemonic, it is the BI operand (or a reduced form of it).
- If the CR is not involved in the branch, the BI operand can be deleted.
- If the CR is involved in the branch, the BI operand can be treated in the following ways:

- It can be specified as a numeric value, just as it is in the architecturally defined instruction, or it can be indicated with an easier to remember formula, **4** * **cr***n* **+** [test bit symbol], where *n* indicates the CR field number.
- The condition of the test bit (eq, lt, gt, and so) can be incorporated into the mnemonic, leaving the need for an operand that defines only the CR field.
  - If the test bit is in CR0, no operand is needed.
  - If the test bit is in CR1–CR7, the BI operand can be replaced with a **cr**S operand (that is, **cr1**, **cr2**, **cr3**, and so forth).

C.4.2 Eliminating the BO Operand

The 5-bit BO field, shown in Figure C-2, encodes the following operations in conditional branch instructions:

- Decrement count register (CTR)
  - And test if result is equal to zero
  - And test if result is not equal to zero
- Test condition register (CR)
  - Test condition true
  - Test condition false
- Branch prediction (taken, fall through). If the prediction bit, *y*, is needed, it is signified by appending a plus or minus sign as described in Section C.4.3, “Incorporating the BO Branch Prediction.”

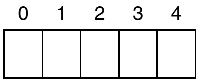


Figure C-2. BO Field (Bits 6–10 of the Instruction Encoding)

BO bits can be interpreted individually as described in this table.

Table C-6. BO Bit Encodings

BO Bit	Description
0	If set, ignore the CR bit comparison.
1	If set, the CR bit comparison is against true, if not set the CR bit comparison is against false
2	If set, the CTR is not decremented.
3	If BO[2] is set, this bit determines whether the CTR comparison is for equal to zero or not equal to zero.
4	Used for static branch prediction. Use of the this bit is optional and independent from the interpretation of the rest of the BO operand. Because simplified branch mnemonics eliminate the BO operand, this bit (the *t* bit) and other branch prediction hint bits (the “*a*” bit) are programmed by adding a plus or minus sign to the simplified mnemonic, as described in Section C.4.3, “Incorporating the BO Branch Prediction.”

Thus, a BO encoding of 10100 (decimal 20) means ignore the CR bit comparison and do not decrement the CTR—in other words, branch unconditionally. Encodings for the BO operand are shown in the following table. A z bit indicates that the bit is ignored. However, these bits should be cleared, as they may be assigned a meaning in a future version of the architecture.

As shown in the following table, the ‘**c**’ in the standard mnemonic is replaced with the operations otherwise specified in the BO field, (**d** for decrement, **z** for zero, **nz** for nonzero, **t** for true, and **f** for false).

Note that the test of when a the CTR reaches 0 varies between 32-bit mode and 64-bit mode. M = 32 in 32-bit mode (of a 64-bit implementation) and M = 0 in 64-bit mode. If the BO field specifies that the CTR is to be decremented, the entire 64-bit CTR is decremented regardless of the mode.

Table C-7. BO Operand Encodings

BO Field	Value[1] (Decimal)	Description	Symbol
0000*z*[2]	0	Decrement the CTR, then branch if the decremented CTR[M:63] ≠ 0; condition is FALSE.	**dnzf**
0001*z*	2	Decrement the CTR, then branch if the decremented CTR[M:63] = 0; condition is FALSE.	**dzf**
001*at*[3]	4	Branch if the condition is FALSE.[4] Note that ‘false’ and ‘four’ both start with ‘f’.	**f**
0100*z*	8	Decrement the CTR, then branch if the decremented CTR[M:63] ≠ 0; condition is TRUE.	**dnzt**
0101*z*	10	Decrement the CTR, then branch if the decremented CTR[M:63] = 0; condition is TRUE.	**dzt**
011*at*	12	Branch if the condition is TRUE. [2] Note that ‘true’ and ‘twelve’ both start with ‘t’.	**t**
1*a*00*t*[5]	16	Decrement the CTR, then branch if the decremented CTR[M:63] ≠ 0.	**dnz**[6]
1*a*01*t*[5]	18	Decrement the CTR, then branch if the decremented CTR[M:63] = 0.	**dz** [6]
1*z*1*zz* [5]	20	Branch always.	—

[1] Assumes t = z = 0. Section C.4.3, “Incorporating the BO Branch Prediction,” describes how to use simplified mnemonics to program the *y* bit for static prediction.

[2] A *z* bit indicates a bit that is ignored. However, these bits should be cleared, as they may be assigned a meaning in a future version of the architecture.

[3] The *a* and *t* bits are used for static branch prediction hints such that *at* = 0b00 specifies no hint, 0b10 specifies the branch is very likely not to be taken, and 0b11 specifies the branch is very likely to be taken.

[4] Instructions for which B0 is 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field, as described in Section C.4.6, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS).”

[5] Simplified mnemonics for branch instructions that do not test CR bits (BO = 16, 18, and 20) should specify only a target. Otherwise a programming error may occur.

[6] Notice that these instructions do not use the branch if condition true or false operations. For that reason, simplified mnemonics for these should not specify a BI operand.

C.4.3 Incorporating the BO Branch Prediction

As shown in Table C-7, the low-order bit (*t* bit) of the BO field along with the *a* bit provides a hint about whether the branch is likely to be taken (static branch prediction). Assemblers should clear these bits unless otherwise directed. This default action indicates the following:

- A branch conditional with a negative displacement field is predicted to be taken.
- A branch conditional with a nonnegative displacement field is predicted not to be taken (fall through).
- A branch conditional to an address in the LR or CTR is predicted not to be taken (fall through).

If the likely outcome (branch or fall through) of a given branch conditional instruction is known, a suffix can be added to the mnemonic that tells the assembler how to set the *at* bits. That is, '+' indicates that the branch is to be taken and '–' indicates that the branch is not to be taken. This suffix can be added to any branch conditional mnemonic, standard or simplified.

For relative and absolute branches (**bc**[**l**][**a**]), the setting of the *at* bits depends on whether the displacement field is negative or nonnegative. For negative displacement fields, coding the suffix '+' causes the bit to be cleared, and coding the suffix '–' causes the bit to be set. For nonnegative displacement fields, coding the suffix '+' causes the bit to be set, and coding the suffix '–' causes the bit to be cleared.

For branches to an address in the LR or CTR (**bclr**[**l**] or **bcctr**[**l**]), coding the suffix '+' causes the *at* bits to be set, and coding the suffix '–' causes the *at* bits to be set to 0b10.

Examples of branch prediction follow:

1. Branch if CR0 reflects less than condition, specifying that the branch should be predicted as taken.

   blt+ *target*

2. Same as (1), but target address is in the LR and the branch should be predicted as not taken.

   bltlr–

C.4.4 The BI Operand—CR Bit and Field Representations

With standard branch mnemonics, the BI operand is used when it is necessary to test a CR bit, as shown in the example in Section C.4, "Branch Instruction Simplified Mnemonics."

With simplified mnemonics, the BI operand is handled differently depending on whether the simplified mnemonic incorporates a CR condition to test, as follows:

- Some branch simplified mnemonics incorporate only the BO operand. These simplified mnemonics can use the architecturally defined BI operand to specify the CR bit, as follows:
  - The BI operand can be presented exactly as it is with standard mnemonics—as a decimal number, 0–31.
  - Symbols can be used to replace the decimal operand, as shown in the example in Section C.4, "Branch Instruction Simplified Mnemonics," where **bdnzt 4 * cr5 + eq,***target* could be used instead of **bdnzt 22,***target*. This is described in Section C.4.4.1.1, "Specifying a CR Bit."

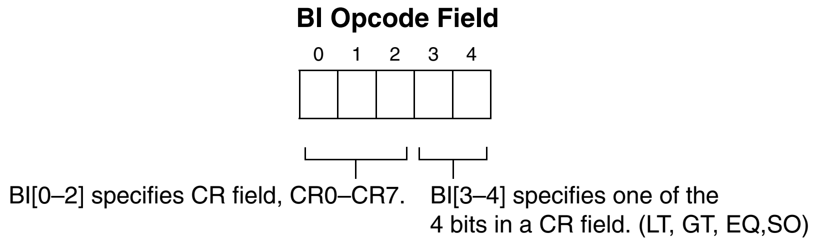
  The simplified mnemonics in Section C.4.5, "Simplified Mnemonics that Incorporate the BO Operand," use one of these two methods to specify a CR bit.
- Additional simplified mnemonics are specified that incorporate CR conditions that would otherwise be specified by the BI operand, so the BI operand is replaced by the **cr**S operand to specify the CR field, CR0–CR7. See Section C.4.4.1, "BI Operand Instruction Encoding."

  These mnemonics are described in Section C.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

C.4.4.1 BI Operand Instruction Encoding

The entire 5-bit BI field, shown in Figure C-3, represents the bit number for the CR bit to be tested. For standard branch mnemonics and for branch simplified mnemonics that do not incorporate a CR condition, the BI operand provides all 5 bits.

For simplified branch mnemonics described in Section C.4.6, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS),” the BI operand is replaced by a **cr**S operand. To understand this, it is useful to view the BI operand as comprised of two parts. As this figure shows, BI[0–2] indicates the CR field and BI[3–4] represents the condition to test.


BI Opcode Field
0
1
2
3
4
BI[0–2] specifies CR field, CR0–CR7.
BI[3–4] specifies one of the 4 bits in a CR field. (LT, GT, EQ,SO)

Simplified mnemonics based on CR conditions but not CTR values—BO = 12 (branch if true) and BO = 4 branch if false)	Specified by a separate, reduced BI operand (**cr**S)	Incorporated into the simplified mnemonic.
Standard branch mnemonics and simplified mnemonics based on CTR values	The BI operand specifies the entire 5-bit field. If CR0 is used, the bit can be identified by LT, GT, EQ, or SO. If CR1–CR7 are used, the form 4 * **cr**S + LT|GT|EQ|SO can be used.	

Figure C-3. BI Field (Bits 11–14 of the Instruction Encoding)

Integer record-form instructions update CR0 and floating-point record-form instructions update CR1 as described in Table C-8.

C.4.4.1.1 Specifying a CR Bit

Note that the AIM version the PowerPC architecture numbers CR bits 0–31 and Power ISA numbers them 32–63. However, no adjustment is necessary to the code; in Power ISA devices, 32 is automatically added to the BI value, as shown in Table C-8 and Table C-9.

Table C-8. CR0 and CR1 Fields as Updated by Integer and Floating-Point Instructions

CR*n* Bit	CR Bits (Operand)	BI		Description
		0–2	3–4	
CR0[0]	32(0)	000	00	Negative (LT)—Set when the result is negative.
CR0[1]	33(1)	000	01	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	34(2)	000	10	Zero (EQ)—Set when the result is zero.
CR0[3]	35(3)	000	11	Summary overflow (SO). Copy of XER[SO] at the instruction’s completion.
CR1[0]	36(4)	001	00	Copy of FPSCR[FX] at the instruction’s completion.
CR1[1]	37(5)	001	01	Copy of FPSCR[FEX] at the instruction’s completion.
CR1[2]	38(6)	001	10	Copy of FPSCR[VX] at the instruction’s completion.
CR1[3]	39(6)	001	11	Copy of FPSCR[OX] at the instruction’s completion.

Some simplified mnemonics incorporate only the BO field (as described Section C.4.2, “Eliminating the BO Operand”). If one of these simplified mnemonics is used and the CR must be accessed, the BI operand can be specified either as a numeric value or by using the symbols in Table C-9.

Compare word instructions (described in Section C.5, “Compare Word Simplified Mnemonics”), floating-point compare instructions, move to CR instructions, and others can also modify CR fields, so CR0 and CR1 may hold values that do not adhere to the meanings described in Table C-8. CR logical instructions, described in Section C.7, “Condition Register Logical Simplified Mnemonics,” can update individual CR bits.

Table C-9. BI Operand Settings for CR Fields for Branch Comparisons

CR*n* Bit	Bit Expression	CR Bits		BI		Description
		BI Operand)	Power ISA Bit Number	0–2	3–4	
CR*n*[0]	**4 * cr0 + lt** (or **lt**)	0	32	000	00	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A < **fr**B.
	4 * cr1 + lt	4	36	001		
	4 * cr2 + lt	8	40	010		
	4 * cr3+ lt	12	44	011		
	4 * cr4 + lt	16	48	100		
	4 * cr5 + lt	20	52	101		
	4 * cr6 + lt	24	56	110		
	4 * cr7 + lt	28	60	111		
CR*n*[1]	**4 * cr0 + gt** (or **gt**)	1	33	000	01	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A > **fr**B.
	4 * cr1 + gt	5	37	001		
	4 * cr2 + gt	9	41	010		
	4 * cr3+ gt	13	45	011		
	4 * cr4 + gt	17	49	100		
	4 * cr5 + gt	21	53	101		
	4 * cr6 + gt	25	57	110		
	4 * cr7 + gt	29	61	111		
CR*n*[2]	**4 * cr0 + eq** (or **eq**)	2	34	000	10	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
	4 * cr1 + eq	6	38	001		
	4 * cr2 + eq	10	42	010		
	4 * cr3+ eq	14	46	011		
	4 * cr4 + eq	18	50	100		
	4 * cr5 + eq	22	54	101		
	4 * cr6 + eq	26	58	110		
	4 * cr7 + eq	30	62	111		
CR*n*[3]	**4 * cr0 + so/un** (or **so/un**)	3	35	000	11	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at instruction completion. For floating-point compare instructions, one or both of frA and frB is a NaN.
	4 * cr1 + so/un	7	39	001		
	4* cr2 + so/un	11	43	010		
	4* cr3 + so/un	15	47	011		
	4* cr4 + so/un	19	51	100		
	4* cr5 + so/un	23	55	101		
	4* cr6 + so/un	27	59	110		
	4* cr7 + so/un	31	63	111		

To provide simplified mnemonics for every possible combination of BO and BI (that is, including bits that identified the CR field) would require 2^{10} = 1024 mnemonics, most of that would be only marginally useful. The abbreviated set in Section C.4.5, “Simplified Mnemonics that Incorporate the BO Operand,” covers useful cases. Unusual cases can be coded using a standard branch conditional syntax.

C.4.4.1.2 The crS Operand

The **cr**S symbols are shown in this table.

NOTE

Either the symbol or the operand value can be used in the syntax used with the simplified mnemonic.

Table C-10. CR Field Identification Symbols

Symbol	BI[0–2]	CR Bits
cr0 (default, can be eliminated from syntax)	000	32–35
cr1	001	36–39
cr2	010	40–43
cr3	011	44–47
cr4	100	48–51
cr5	101	52–55
cr6	110	56–59
cr7	111	60–63

To identify a CR bit, an expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol can be used (for example, **cr0 * 4 + eq**).

C.4.5 Simplified Mnemonics that Incorporate the BO Operand

The mnemonics in this table allow common BO operand encodings to be specified as part of the mnemonic, along with the absolute address (AA) and set link register bits (LK). There are no simplified mnemonics for relative and absolute unconditional branches. For these, the basic mnemonics **b**, **ba**, **bl**, and **bla** are used.

Table C-11. Branch Simplified Mnemonics

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc	bca	bclr	bcctr	bcl	bcla	bclrl	bcctrl
Branch unconditionally [1]	—	—	**blr**	**bctr**	—	—	**blrl**	**bctrl**
Branch if condition true	**bt**	**bta**	**btlr**	**btctr**	**btl**	**btla**	**btlrl**	**btctrl**
Branch if condition false	**bf**	**bfa**	**bflr**	**bfctr**	**bfl**	**bfla**	**bflrl**	**bfctrl**
Decrement CTR, branch if CTR ≠ 0 [1]	**bdnz**	**bdnza**	**bdnzlr**	—	**bdnzl**	**bdnzla**	**bdnzlrl**	—
Decrement CTR, branch if CTR ≠ 0 and condition true	**bdnzt**	**bdnzta**	**bdnztlr**	—	**bdnztl**	**bdnztla**	**bdnztlrl**	—

Table C-11. Branch Simplified Mnemonics (continued)

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc	bca	bclr	bcctr	bcl	bcla	bclrl	bcctrl
Decrement CTR, branch if CTR ≠ 0 and condition false	**bdnzf**	**bdnzfa**	**bdnzflr**	—	**bdnzfl**	**bdnzfla**	**bdnzflrl**	—
Decrement CTR, branch if CTR = 0 [1]	**bdz**	**bdza**	**bdzlr**	—	**bdzl**	**bdzla**	**bdzlrl**	—
Decrement CTR, branch if CTR = 0 and condition true	**bdzt**	**bdzta**	**bdztlr**	—	**bdztl**	**bdztla**	**bdztlrl**	—
Decrement CTR, branch if CTR = 0 and condition false	**bdzf**	**bdzfa**	**bdzflr**	—	**bdzfl**	**bdzfla**	**bdzflrl**	—

[1] Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise a programming error may occur.

This table shows the syntax for basic simplified branch mnemonics

Table C-12. Branch Instructions

Instruction	Standard Mnemonic	Syntax	Simplified Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr	N/A, syntax does not include BO	
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr	**b***x*[1](**b*x*a** **b*x*l** **b*x*la**)	BI[2]target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI	**b*x*lr** (**b*x*lrl**)	BI
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI	**b*x*ctr** (**b*x*ctrl**)	BI

[1] *x* stands for one of the symbols in Table C-7, where applicable.

[2] BI can be a numeric value or an expression as shown in Table C-10.

The simplified mnemonics in Table C-11 that test a condition require a corresponding CR bit as the first operand (as examples 2–5 below illustrate). The symbols in Table C-10 can be used in place of a numeric value.

C.4.5.1 Examples that Eliminate the BO Operand

The simplified mnemonics in Table C-11 are used in the following examples:

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR) (note that no CR bits are tested).
   bdnz *target* equivalent to **bc 16,0,***target*

   Because this instruction does not test a CR bit, the simplified mnemonic should specify only a target operand. Specifying a CR (for example, **bdnz** 0,*target* or **bdnz cr0,***target*) may be considered a programming error. Subsequent examples test conditions).
2. Same as (1) but branch only if CTR is nonzero and equal condition in CR0.
   bdnzt eq,*target* equivalent to **bc 8,2,***target*

   Other equivalents include **bdnzt 2,***target* or the unlikely **bdnzt 4*cr0+eq,***target*

3. Same as (2), but equal condition is in CR5.
   bdnzt 4 * cr5 + eq,*target* equivalent to **bc 8,22,***target*
   bdnzt 22,*target* would also work
4. Branch if bit 59 of CR is false.
   bf 27,*target* equivalent to **bc 4,27,***target*
   bf 4*cr6+so,*target* would also work
5. Same as (4), but set the link register. This is a form of conditional call.
   bfl 27,*target* equivalent to **bcl 4,27,***target*

This table lists simplified mnemonics and syntax for **bc** and **bca** without LR updating.

Table C-13. Simplified Mnemonics for bc and bca without LR Update

Branch Semantics	bc	Simplified Mnemonic	bca	Simplified Mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true[1]	**bc 12,BI,**target	**bt BI,**target	**bca 12,BI,**target	**bta BI,**target
Branch if condition false [1]	**bc 4,BI,**target	**bf BI,**target	**bca 4,BI,**target	**bfa BI,**target
Decrement CTR, branch if CTR ≠ 0	**bc 16,0,**target	**bdnz** target[2]	**bca 16,0,**target	**bdnza** target [2]
Decrement CTR, branch if CTR ≠ 0 and condition true	**bc 8,BI,**target	**bdnzt BI,**target	**bca 8,BI,**target	**bdnzta BI,**target
Decrement CTR, branch if CTR ≠ 0 and condition false	**bc 0,BI,**target	**bdnzf BI,**target	**bca 0,BI,**target	**bdnzfa BI,**target
Decrement CTR, branch if CTR = 0	**bc 18,0,**target	**bdz** target [2]	**bca 18,0,**target	**bdza** target [2]
Decrement CTR, branch if CTR = 0 and condition true	**bc 10,BI,**target	**bdzt BI,**target	**bca 10,BI,**target	**bdzta BI,**target
Decrement CTR, branch if CTR = 0 and condition false	**bc 2,BI,**target	**bdzf BI,**target	**bca 2,BI,**target	**bdzfa BI,**target

[1] Instructions for which B0 is either 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field, as described in Section C.4.6, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS).”

[2] Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise a programming error may occur.

This table lists simplified mnemonics and syntax for **bclr** and **bcctr** without LR updating.

Table C-14. Simplified Mnemonics for bclr and bcctr without LR Update

Branch Semantics	bclr	Simplified Mnemonic	bcctr	Simplified Mnemonic
Branch unconditionally	**bclr 20,0**	**blr** [1]	**bcctr 20,0**	**bctr** [1]
Branch if condition true [2]	**bclr 12,BI**	**btlr BI**	**bcctr 12,BI**	**btctr BI**
Branch if condition false [2]	**bclr 4,BI**	**bflr BI**	**bcctr 4,BI**	**bfctr BI**
Decrement CTR, branch if CTR ≠ 0	**bclr 16,BI**	**bdnzlr BI**	—	—
Decrement CTR, branch if CTR ≠ 0 and condition true	**bclr 8,BI**	**bdnztlr BI**	—	—

Table C-14. Simplified Mnemonics for bclr and bcctr without LR Update (continued)

Branch Semantics	bclr	Simplified Mnemonic	bcctr	Simplified Mnemonic
Decrement CTR, branch if CTR ≠ 0 and condition false	**bclr 0,BI**	**bdnzflr BI**	—	—
Decrement CTR, branch if CTR = 0	**bclr 18,0**	**bdzlr** [1]	—	—
Decrement CTR, branch if CTR = 0 and condition true	**bclr 8,BI**	**bdnztlr BI**	—	—
Decrement CTR, branch if CTR = 0 and condition false	**bclr 2,BI**	**bdzflr BI**	—	—

[1] Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.

[2] Instructions for which B0 is 12 (branch if condition true) or 4 (branch if condition false) do not depend on a CTR value and can be alternately coded by incorporating the condition specified by the BI field. See Section C.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

This table provides simplified mnemonics and syntax for **bcl** and **bcla**.

Table C-15. Simplified Mnemonics for bcl and bcla with LR Update

Branch Semantics	bcl	Simplified Mnemonic	bcla	Simplified Mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true [1]	**bcl 12,BI,**target	**btl BI,**target	**bcla 12,BI,**target	**btla BI,**target
Branch if condition false [1]	**bcl 4,BI,**target	**bfl BI,**target	**bcla 4,BI,**target	**bfla BI,**target
Decrement CTR, branch if CTR ≠ 0	**bcl 16,0,**target	**bdnzl** target [2]	**bcla 16,0,**target	**bdnzla** target [2]
Decrement CTR, branch if CTR ≠ 0 and condition true	**bcl 8,0,**target	**bdnztl BI,**target	**bcla 8,BI,**target	**bdnztla BI,**target
Decrement CTR, branch if CTR ≠ 0 and condition false	**bcl 0,BI,**target	**bdnzfl BI,**target	**bcla 0,BI,**target	**bdnzfla BI,**target
Decrement CTR, branch if CTR = 0	**bcl 18,BI,**target	**bdzl** target [2]	**bcla 18,BI,**target	**bdzla** target [2]
Decrement CTR, branch if CTR = 0 and condition true	**bcl 10,BI,**target	**bdztl BI,**target	**bcla 10,BI,**target	**bdztla BI,**target
Decrement CTR, branch if CTR = 0 and condition false	**bcl 2,BI,**target	**bdzfl BI,**target	**bcla 2,BI,**target	**bdzfla BI,**target

[1] Instructions for which B0 is either 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field. See Section C.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

[2] Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. A programming error may occur.

This table provides simplified mnemonics and syntax for **bclrl** and **bcctrl** with LR updating.

Table C-16. Simplified Mnemonics for bclrl and bcctrl with LR Update

Branch Semantics	bclrl	Simplified Mnemonic	bcctrl	Simplified Mnemonic
Branch unconditionally	**bclrl 20,0**	**blrl** [1]	**bcctrl 20,0**	**bctrl** [1]
Branch if condition true	**bclrl 12,BI**	**btlrl BI**	**bcctrl 12,BI**	**btctrl BI**

Table C-16. Simplified Mnemonics for bclrl and bcctrl with LR Update (continued)

Branch Semantics	bclrl	Simplified Mnemonic	bcctrl	Simplified Mnemonic
Branch if condition false	**bclrl 4,BI**	**bflrl BI**	**bcctrl 4,BI**	**bfctrl BI**
Decrement CTR, branch if CTR ≠ 0	**bclrl 16,0**	**bdnzlrl** [1]	—	—
Decrement CTR, branch if CTR ≠ 0 and condition true	**bclrl 8,BI**	**bdnztlrl BI**	—	—
Decrement CTR, branch if CTR ≠ 0 and condition false	**bclrl 0,BI**	**bdnzflrl BI**	—	—
Decrement CTR, branch if CTR = 0	**bclrl 18,0**	**bdzlrl** [1]	—	—
Decrement CTR, branch if CTR = 0 and condition true	**bclrl 10, BI**	**bdztlrl BI**	—	—
Decrement CTR, branch if CTR = 0 and condition false	**bclrl 2,BI**	**bdzflrl BI**	—	—

[1] Simplified mnemonics for branch instructions that do not test a CR bit should not specify one. A programming error may occur.

C.4.6 Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)

The mnemonics in Table C-19 are variations of the branch-if-condition-true (BO = 12) and branch-if-condition-false (BO = 4) encodings. Because these instructions do not depend on the CTR, the true/false conditions specified by BO can be combined with the CR test bit specified by BI to create a different set of simplified mnemonics that eliminates the BO operand and the portion of the BI operand (BI[3–4]) that specifies one of the four possible test bits. However, the simplified mnemonic cannot specify in which of the eight CR fields the test bit falls, so the BI operand is replaced by a **cr**S operand.

The standard codes shown in Table C-17 are used for the most common combinations of branch conditions. Note that for ease of programming, these codes include synonyms; for example, less than or equal (**le**) and not greater than (**ng**) achieve the same result.

NOTE

A CR field symbol, **cr0–cr7**, is used as the first operand after the simplified mnemonic. If CR0 is used, no **cr**S is necessary.

Table C-17. Standard Coding for Branch Conditions

Code	Description	Equivalent	Bit Tested
lt	Less than	—	LT
le	Less than or equal (equivalent to **ng**)	**ng**	GT
eq	Equal	—	EQ
ge	Greater than or equal (equivalent to **nl**)	**nl**	LT
gt	Greater than	—	GT
nl	Not less than (equivalent to **ge**)	**ge**	LT
ne	Not equal	—	EQ
ng	Not greater than (equivalent to **le**)	**le**	GT
so	Summary overflow	—	SO

Table C-17. Standard Coding for Branch Conditions (continued)

Code	Description	Equivalent	Bit Tested
ns	Not summary overflow	—	SO
un	Unordered (after floating-point comparison)	—	SO
nu	Not unordered (after floating-point comparison)	—	SO

This table shows the syntax for simplified branch mnemonics that incorporate CR conditions. Here, **cr**S replaces a BI operand to specify only a CR field (because the specific CR bit within the field is now part of the simplified mnemonic. Note that the default is CR0; if no **cr**S is specified, CR0 is used.

Table C-18. Branch Instructions and Simplified Mnemonics that Incorporate CR Conditions

Instruction	Standard Mnemonic	Syntax	Simplified Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr	—	
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr	**b***x* [1](**b***x***a b***x***l b***x***la**)	**cr**S[2],target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI	**b***x***lr** (**b***x***lrl**)	**cr**S
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI	**b***x***ctr** (**b***x***ctrl**)	**cr**S

[1] *x* stands for one of the symbols in Table C-17, where applicable.

[2] BI can be a numeric value or an expression as shown in Table C-10.

This table shows the simplified branch mnemonics incorporating conditions.

Table C-19. Simplified Mnemonics with Comparison Conditions

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc	bca	bclr	bcctr	bcl	bcla	bclrl	bcctrl
Branch if less than	**blt**	**blta**	**bltlr**	**bltctr**	**bltl**	**bltla**	**bltlrl**	**bltctrl**
Branch if less than or equal	**ble**	**blea**	**blelr**	**blectr**	**blel**	**blela**	**blelrl**	**blectrl**
Branch if equal	**beq**	**beqa**	**beqlr**	**beqctr**	**beql**	**beqla**	**beqlrl**	**beqctrl**
Branch if greater than or equal	**bge**	**bgea**	**bgelr**	**bgectr**	**bgel**	**bgela**	**bgelrl**	**bgectrl**
Branch if greater than	**bgt**	**bgta**	**bgtlr**	**bgtctr**	**bgtl**	**bgtla**	**bgtlrl**	**bgtctrl**
Branch if not less than	**bnl**	**bnla**	**bnllr**	**bnlctr**	**bnll**	**bnlla**	**bnllrl**	**bnlctrl**
Branch if not equal	**bne**	**bnea**	**bnelr**	**bnectr**	**bnel**	**bnela**	**bnelrl**	**bnectrl**
Branch if not greater than	**bng**	**bnga**	**bnglr**	**bngctr**	**bngl**	**bngla**	**bnglrl**	**bngctrl**
Branch if summary overflow	**bso**	**bsoa**	**bsolr**	**bsoctr**	**bsol**	**bsola**	**bsolrl**	**bsoctrl**
Branch if not summary overflow	**bns**	**bnsa**	**bnslr**	**bnsctr**	**bnsl**	**bnsla**	**bnslrl**	**bnsctrl**
Branch if unordered	**bun**	**buna**	**bunlr**	**bunctr**	**bunl**	**bunla**	**bunlrl**	**bunctrl**
Branch if not unordered	**bnu**	**bnua**	**bnulr**	**bnuctr**	**bnul**	**bnula**	**bnulrl**	**bnuctrl**

Instructions using the mnemonics in Table C-19 indicate the condition bit, but not the CR field. If no field is specified, CR0 is used. The CR field symbols defined in Table C-10 (**cr0**–**cr7**) are used for this operand, as shown in examples 2–4 below.

C.4.6.1 Branch Simplified Mnemonics that Incorporate CR Conditions: Examples

The following examples use the simplified mnemonics shown in Table C-19:

1. Branch if CR0 reflects not-equal condition.
   bne *target* equivalent to **bc 4,2,***target*
2. Same as (1) but condition is in CR3.
   bne cr3,*target* equivalent to **bc 4,14,***target*
3. Branch to an absolute target if CR4 specifies greater than condition, setting the LR. This is a form of conditional call.
   bgtla cr4,*target* equivalent to **bcla 12,17,***target*
4. Same as (3), but target address is in the CTR.
   bgtctrl cr4 equivalent to **bcctrl 12,17**

C.4.6.2 Branch Simplified Mnemonics that Incorporate CR Conditions: Listings

This table shows simplified branch mnemonics and syntax for **bc** and **bca** without LR updating.

Table C-20. Simplified Mnemonics for bc and bca without Comparison Conditions or LR Update

Branch Semantics	bc	Simplified Mnemonic	bca	Simplified Mnemonic
Branch if less than	**bc 12**,BI[1],target	**blt cr**S target	**bca 12**,BI[1],target	**blta cr**S target
Branch if less than or equal	**bc 4**,BI[2],target	**ble cr**S target	**bca 4**,BI[2],target	**blea cr**S target
Branch if not greater than		**bng cr**S target		**bnga cr**S target
Branch if equal	**bc 12**,BI[3],target	**beq cr**S target	**bca 12**,BI[3],target	**beqa cr**S target
Branch if greater than or equal	**bc 4**,BI[1],target	**bge cr**S target	**bca 4**,BI[1],target	**bgea cr**S target
Branch if not less than		**bnl cr**S target		**bnla cr**S target
Branch if greater than	**bc 12**,BI[2],target	**bgt cr**S target	**bca 12**,BI[2],target	**bgta cr**S target
Branch if not equal	**bc 4**,BI[3],target	**bne cr**S target	**bca 4**,BI[3],target	**bnea cr**S target
Branch if summary overflow	**bc 12**,BI[4],target	**bso cr**S target	**bca 12**,BI[4],target	**bsoa cr**S target
Branch if unordered		**bun cr**S target		**buna cr**S target
Branch if not summary overflow	**bc 4**,BI[4],target	**bns cr**S target	**bca 4**,BI[4],target	**bnsa cr**S target
Branch if not unordered		**bnu cr**S target		**bnua cr**S target

Note:

[1] The value in the BI operand selects CR*n*[0], the LT bit.

[2] The value in the BI operand selects CR*n*[1], the GT bit.

[3] The value in the BI operand selects CR*n*[2], the EQ bit.

[4] The value in the BI operand selects CR*n*[3], the SO bit.

This table shows simplified branch mnemonics and syntax for **bclr** and **bcctr** without LR updating.

Table C-21. Simplified Mnemonics for bclr and bcctr without Comparison Conditions or LR Update

Branch Semantics	bclr	Simplified Mnemonic	bcctr	Simplified Mnemonic
Branch if less than	**bclr 12**,BI[1],target	**bltlr cr**S target	**bcctr 12**,BI[1],target	**bltctr cr**S target
Branch if less than or equal	**bclr 4**,BI[2],target	**blelr cr**S target	**bcctr 4**,BI[2],target	**blectr cr**S target
Branch if not greater than		**bnglr cr**S target		**bngctr cr**S target
Branch if equal	**bclr 12**,BI[3],target	**beqlr cr**S target	**bcctr 12**,BI[3],target	**beqctr cr**S target
Branch if greater than or equal	**bclr 4**,BI[1],target	**bgelr cr**S target	**bcctr 4**,BI[1],target	**bgectr cr**S target
Branch if not less than		**bnllr cr**S target		**bnlctr cr**S target
Branch if greater than	**bclr 12**,BI[2],target	**bgtlr cr**S target	**bcctr 12**,BI[2],target	**bgtctr cr**S target
Branch if not equal	**bclr 4**,BI[3],target	**bnelr cr**S target	**bcctr 4**,BI[3],target	**bnectr cr**S target
Branch if summary overflow	**bclr 12**,BI[4],target	**bsolr cr**S target	**bcctr 12**,BI[4],target	**bsoctr cr**S target
Branch if unordered		**bunlr cr**S target		**bunctr cr**S target
Branch if not summary overflow	**bclr 4**,BI[4],target	**bnslr cr**S target	**bcctr 4**,BI[4],target	**bnsctr cr**S target
Branch if not unordered	—	**bnulr cr**S target	—	**bnuctr cr**S target

Note:

[1] The value in the BI operand selects CR*n*[0], the LT bit.

[2] The value in the BI operand selects CR*n*[1], the GT bit.

[3] The value in the BI operand selects CR*n*[2], the EQ bit.

[4] The value in the BI operand selects CR*n*[3], the SO bit.

This table shows simplified branch mnemonics and syntax for **bcl** and **bcla**.

Table C-22. Simplified Mnemonics for bcl and bcla with Comparison Conditions and LR Update

Branch Semantics	bcl	Simplified Mnemonic	bcla	Simplified Mnemonic
Branch if less than	**bcl 12**,BI[1],target	**bltl cr**S target	**bcla 12**,BI[1],target	**bltla cr**S target
Branch if less than or equal	**bcl 4**,BI[2],target	**blel cr**S target	**bcla 4**,BI[2],target	**blela cr**S target
Branch if not greater than		**bngl cr**S target		**bngla cr**S target
Branch if equal	**bcl 12**,BI[3],target	**beql cr**S target	**bcla 12**,BI[3],target	**beqla cr**S target
Branch if greater than or equal	**bcl 4**,BI[1],target	**bgel cr**S target	**bcla 4**,BI[1],target	**bgela cr**S target
Branch if not less than		**bnll cr**S target		**bnlla cr**S target
Branch if greater than	**bcl 12**,BI[2],target	**bgtl cr**S target	**bcla 12**,BI[2],target	**bgtla cr**S target
Branch if not equal	**bcl 4**,BI[3],target	**bnel cr**S target	**bcla 4**,BI[3],target	**bnela cr**S target
Branch if summary overflow	**bcl 12**,BI[4],target	**bsol cr**S target	**bcla 12**,BI[4],target	**bsola cr**S target
Branch if unordered	—	**bunl cr**S target	—	**bunla cr**S target
Branch if not summary overflow	**bcl 4**,BI[4],target	**bnsl cr**S target	**bcla 4**,BI[4],target	**bnsla cr**S target

Table C-22. Simplified Mnemonics for bcl and bcla with Comparison Conditions and LR Update (continued)

Branch Semantics	bcl	Simplified Mnemonic	bcla	Simplified Mnemonic
Branch if not unordered	—	**bnul cr**S target	—	**bnula cr**S target

Note:

[1] The value in the BI operand selects CR*n*[0], the LT bit.
[2] The value in the BI operand selects CR*n*[1], the GT bit.
[3] The value in the BI operand selects CR*n*[2], the EQ bit.
[4] The value in the BI operand selects CR*n*[3], the SO bit.

This table shows the simplified branch mnemonics and syntax for **bclrl** and **bcctrl** with LR updating.

Table C-23. Simplified Mnemonics for bclrl and bcctrl with Comparison Conditions and LR Update

Branch Semantics	bclrl	Simplified Mnemonic	bcctrl	Simplified Mnemonic
Branch if less than	**bclrl 12,**BI[1]**,target**	**bltlrl cr**S target	**bcctrl 12,**BI[1]**,target**	**bltctrl cr**S target
Branch if less than or equal	**bclrl 4,**BI[2]**,target**	**blelrl cr**S target	**bcctrl 4,**BI[2]**,target**	**blectrl cr**S target
Branch if not greater than		**bnglrl cr**S target		**bngctrl cr**S target
Branch if equal	**bclrl 12,**BI[3]**,target**	**beqlrl cr**S target	**bcctrl 12,**BI[3]**,target**	**beqctrl cr**S target
Branch if greater than or equal	**bclrl 4,**BI[1]**,target**	**bgelrl cr**S target	**bcctrl 4,**BI[1]**,target**	**bgectrl cr**S target
Branch if not less than		**bnllrl cr**S target		**bnlctrl cr**S target
Branch if greater than	**bclrl 12,**BI[2]**,target**	**bgtlrl cr**S target	**bcctrl 12,**BI[2]**,target**	**bgtctrl cr**S target
Branch if not equal	**bclrl 4,**BI[3]**,target**	**bnelrl cr**S target	**bcctrl 4,**BI[3]**,target**	**bnectrl cr**S target
Branch if summary overflow	**bclrl 12,**B[4]**,target**	**bsolrl cr**S target	**bcctrl 12,**BI[4]**,target**	**bsoctrl cr**S target
Branch if unordered	—	**bunlrl cr**S target	—	**bunctrl cr**S target
Branch if not summary overflow	**bclrl 4,**BI[4]**,target**	**bnslrl cr**S target	**bcctrl 4,**BI[4]**,target**	**bnsctrl cr**S target
Branch if not unordered	—	**bnulrl cr**S target	—	**bnuctrl cr**S target

Note:

[1] The value in the BI operand selects CR*n*[0], the LT bit.
[2] The value in the BI operand selects CR*n*[1], the GT bit.
[3] The value in the BI operand selects CR*n*[2], the EQ bit.
[4] The value in the BI operand selects CR*n*[3], the SO bit.

C.5 Compare Word Simplified Mnemonics

In compare word instructions, the L operand indicates a word (L = 0) or a double-word (L = 1). Simplified mnemonics in this table eliminate the L operand for word comparisons.

Table C-24. Word Compare Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Compare Word Immediate	**cmpwi cr**D,**r**A,SIMM	**cmpi cr**D,**0**,**r**A,SIMM
Compare Word	**cmpw cr**D,**r**A,**r**B	**cmp cr**D,**0**,**r**A,**r**B
Compare Logical Word Immediate	**cmplwi cr**D,**r**A,UIMM	**cmpli cr**D,**0**,**r**A,UIMM
Compare Logical Word	**cmplw cr**D,**r**A,**r**B	**cmpl cr**D,**0**,**r**A,**r**B

As with branch mnemonics, the **cr**D field of a compare instruction can be omitted if CR0 is used, as shown in examples 1 and 3 below. Otherwise, the target CR field must be specified as the first operand. The following examples use word compare mnemonics:

1. Compare **r**A with immediate value 100 as signed 32-bit integers and place result in CR0.
   cmpwi rA**,100** equivalent to **cmpi 0,0,r**A**,100**
2. Same as (1), but place results in CR4.
   cmpwi cr4,rA**,100** equivalent to **cmpi 4,0,r**A**,100**
3. Compare **r**A and **r**B as unsigned 32-bit integers and place result in CR0.
   cmplw rA**,r**B equivalent to **cmpl 0,0,r**A**,r**B

C.6 Compare Double-word Simplified Mnemonics

In compare double-word instructions, the L operand indicates a word (L = 0) or a double-word (L = 1). Simplified mnemonics in Table C-24 eliminate the L operand for double-word comparisons.

Table C-25. Double-word Compare Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Compare Double-word Immediate	**cmpdi cr**D,**r**A,SIMM	**cmpi cr**D,**1**,**r**A,SIMM
Compare Double-word	**cmpd cr**D,**r**A,**r**B	**cmp cr**D,**1**,**r**A,**r**B
Compare Logical Double-word Immediate	**cmpldi cr**D,**r**A,UIMM	**cmpli cr**D,**1**,**r**A,UIMM
Compare Logical Double-word	**cmpld cr**D,**r**A,**r**B	**cmpl cr**D,**1**,**r**A,**r**B

As with branch mnemonics, the **cr**D field of a compare instruction can be omitted if CR0 is used, as shown in examples 1 and 3 below. Otherwise, the target CR field must be specified as the first operand. The following examples use word compare mnemonics:

1. Compare **r**A with immediate value 100 as signed 64-bit integers and place result in CR0.
   cmpdi rA**,100** equivalent to **cmpi 0,1,r**A**,100**
2. Same as (1), but place results in CR4.
   cmpdi cr4,rA**,100** equivalent to **cmpi 4,1**,**r**A**,100**
3. Compare **r**A and **r**B as unsigned 64-bit integers and place result in CR0.
   cmpld rA**,r**B equivalent to **cmpl 0,1,r**A**,r**B

C.7 Condition Register Logical Simplified Mnemonics

The CR logical instructions, shown in Table C-26, can be used to set, clear, copy, or invert a given CR bit. Simplified mnemonics allow these operations to be coded easily. Note that the symbols defined in Table C-9 can be used to identify the CR bit.

Table C-26. Condition Register Logical Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to
Condition register set	**crset** b*x*	**creqv** b*x*,b*x*,b*x*
Condition register clear	**crclr** b*x*	**crxor** b*x*,b*x*,b*x*
Condition register move	**crmove** b*x*,b*y*	**cror** b*x*,b*y*,b*y*
Condition register not	**crnot** b*x*,b*y*	**crnor** b*x*,b*y*,b*y*

Examples using the CR logical mnemonics follow:

1. Set CR[57].
   crset 25 equivalent to **creqv 25,25,25**
2. Clear CR0[SO].
   crclr so equivalent to **crxor 3,3,3**
3. Same as (2), but clear CR3[SO].
   crclr 4 * cr3 + so equivalent to **crxor 15,15,15**
4. Invert the CR0[EQ].**crnot eq,eq** equivalent to**crnor 2,2,2**
5. Same as (4), but CR4[EQ] is inverted and the result is placed into CR5[EQ].
   crnot 4 * cr5 + eq, 4 * cr4 + eq equivalent to **crnor 22,18,18**

C.8 Trap Instructions Simplified Mnemonics

The codes in this table are for the most common combinations of trap conditions.

Table C-27. Standard Codes for Trap Instructions

Code	Description	TO Encoding	<	>	=	<U[1]	>U [2]
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1

Table C-27. Standard Codes for Trap Instructions (continued)

Code	Description	TO Encoding	<	>	=	<U[1]	>U [2]
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1
lng	Logically not greater than	6	0	0	1	1	0
—	Unconditional	31	1	1	1	1	1

[1] The symbol '<U' indicates an unsigned less-than evaluation is performed.

[2] The symbol '>U' indicates an unsigned greater-than evaluation is performed.

The mnemonics in Table C-28 are variations of trap instructions, with the most useful TO values represented in the mnemonic rather than specified as a numeric operand.

Table C-28. Trap Simplified Mnemonics

Trap Semantics	32-Bit Comparison		64-Bit Comparison	
	twi Immediate	tw Register	tdi Immediate	td Register
Trap unconditionally	—	**trap**	—	—
Trap if less than	**twlti**	**twlt**	**tdlti**	**tdlt**
Trap if less than or equal	**twlei**	**twle**	**tdlei**	**tdle**
Trap if equal	**tweqi**	**tweq**	**tdeqi**	**tdeq**
Trap if greater than or equal	**twgei**	**twge**	**tdgei**	**tdge**
Trap if greater than	**twgti**	**twgt**	**tdgti**	**tdgt**
Trap if not less than	**twnli**	**twnl**	**tdnli**	**tdnl**
Trap if not equal	**twnei**	**twne**	**tdnei**	**tdne**
Trap if not greater than	**twngi**	**twng**	**tdngi**	**tdng**
Trap if logically less than	**twllti**	**twllt**	**tdllti**	**tdllt**
Trap if logically less than or equal	**twllei**	**twlle**	**tdllei**	**tdlle**
Trap if logically greater than or equal	**twlgei**	**twlge**	**tdlgei**	**tdlge**
Trap if logically greater than	**twlgti**	**twlgt**	**tdlgti**	**tdlgt**
Trap if logically not less than	**twlnli**	**twlnl**	**tdlnli**	**tdlnl**
Trap if logically not greater than	**twlngi**	**twlng**	**tdlngi**	**tdlng**

The following examples use the simplified trap mnemonics:

1. Trap if **r**A is not zero.
   twnei rA**,0** equivalent to **twi 24,r**A**,0**
2. Trap if **r**A is not equal to **r**B.
   twne rA**, r**B equivalent to **tw 24,r**A**,r**B
3. Trap if **r**A is logically greater than 0x7FF.
   twlgti rA**,** 0x7FF equivalent to **twi 1,r**A**,** 0x7FF

4. Trap unconditionally.
   trap equivalent to **tw 31,0,0**

Trap instructions evaluate a trap condition as follows: The contents of **r**A are compared with either the sign-extended SIMM field or the contents of **r**B, depending on the trap instruction.

The comparison results in five conditions that are ANDed with operand TO. If the result is not 0, the trap exception handler is invoked. See this table for these conditions.

Table C-29. TO Operand Bit Encoding

TO Bit	ANDed with Condition
0	Less than, using signed comparison
1	Greater than, using signed comparison
2	Equal
3	Less than, using unsigned comparison
4	Greater than, using unsigned comparison

C.9 Simplified Mnemonics for Accessing SPRs

The **mtspr** and **mfspr** instructions specify a special-purpose register (SPR) as a numeric operand. Simplified mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as a numeric operand. The pattern for **mtspr** and **mfspr** simplified mnemonics is straightforward: replace the **-spr** portion of the mnemonic with the abbreviation for the spr (for example XER, SRR0, or LR), eliminate the SPRN operand, leaving the source or destination GPR operand, **r**S or **r**D.

Following are examples using the SPR simplified mnemonics:

1. Copy the contents of the low-order 32 bits of **r**S to the XER.
   mtxer rS equivalent to **mtspr 1,r**S
2. Copy the contents of the LR to **r**D.
   mflr rD equivalent to **mfspr r**D**,8**
3. Copy the contents of **r**S to the CTR.
   mtctr rS equivalent to **mtspr 9,r**S

The architecture describes extended mnemonics for accessing CTR, LR, and XER only. However, some assemblers support other SPRs in the same fashion as shown in the following examples:

1. Copy the contents of the low-order 32 bits of **r**S to CSRR1.
   mtcsrr1 rS equivalent to **mtspr 59,r**S
2. Copy the contents of IVOR0 to **r**D.
   mfivor0 rD equivalent to **mfspr r**D**,400**
3. Copy the contents of **r**S to the SRR0.
   mtsrr0 rS equivalent to **mtspr 26,r**S

There is an additional simplified mnemonic convention for accessing SPRGs. These are shown in this table along with the equivalent simplified mnemonic using the formula described above.

Table C-30. Additional Simplified Mnemonics for Accessing SPRGs

SPR	Move to SPR		Move from SPR	
	Simplified Mnemonic	Equivalent to	Simplified Mnemonic	Equivalent to
SPRGs	**mtspr**g *n,* **r**S	**mtspr** 272 + *n*,**r**S	**mfsprg r**D*, n*	**mfspr r**D,272 + *n*
	mtsprg*n,* **r**S		**mfsprg***n* **r**D	

C.10 Recommended Simplified Mnemonics

This section describes commonly-used operations (such as no-op, load immediate, load address, move register, and complement register).

C.10.1 No-Op (nop)

Many instructions can be coded so that, effectively, no operation is performed. A mnemonic is provided for the preferred form of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred form is the following:

Table C-31. No-Op Simplified Mnemonic

Operation	Simplified Mnemonic	Equivalent to
No operation	**nop**	**ori 0,0,0**

C.10.2 Load Immediate (li)

The **addi** and **addis** instructions can be used to load an immediate value into a register. Additional mnemonics are provided to convey the idea that no addition is being performed but that data is being moved from the immediate operand of the instruction to a register.

1. Load a 16-bit signed immediate value into **r**D.
   li rD**,**value equivalent to **addi r**D**,0,**value
2. Load a 16-bit signed immediate value, shifted left by 16 bits, into **r**D.
   lis rD**,**value equivalent to **addis r**D**,0,**value

C.10.3 Load Address (la)

This mnemonic permits computing the value of a base-displacement operand, using the **addi** instruction that normally requires a separate register and immediate operands.

The **la** mnemonic is useful for obtaining the address of a variable specified by name, allowing the assembler to supply the base register number and compute the displacement. If the variable *v* is located at offset d*v* bytes from the address in **r***v,* and the assembler has been told to use **r***v* as a base for references to the data structure containing *v*, the **la r**D,*v* mnemonic causes the address of *v* to be loaded into **r**D, as shown in this table:

Table C-32. Load Address Simplified Mnemonic

Operation	Simplified Mnemonic	Equivalent to
Load address	**la r**D,d(**r**A)	**addi r**D,**r**A,d
	la rD,*v*	**addi r**D,**r***v*,d*v*

C.10.4 Move Register (mr)

Several instructions can be coded to copy the contents of one register to another. A simplified mnemonic is provided that signifies that no computation is being performed, but merely that data is being moved from one register to another.

The instruction in the following table copies the contents of **r**S into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

Table C-33. Move Register Simplified Mnemonic

Operation	Simplified Mnemonic	Equivalent to
Move register	**mr r**A,**r**S	**or r**A,**r**S,**r**S

C.10.5 Complement Register (not)

Several instructions can be coded in a way that they complement the contents of one register and place the result into another register. A simplified mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of **r**S and places the result into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

Table C-34. Complement Register Simplified Mnemonic

Operation	Simplified Mnemonic	Equivalent to
Complement register	**not r**A,**r**S	**nor r**A,**r**S,**r**S

C.10.6 Move to Condition Register (mtcr)

This mnemonic permits copying the contents of a GPR to the CR, using the same syntax as the **mfcr** instruction.

Table C-35. Move to Condition Register Simplified Mnemonic

Operation	Simplified Mnemonic	Equivalent to
Move to condition register	**mtcr r**S	**mtcrf** 0xFF,**r**S

C.10.7 Sync (sync)

The **sync** extended mnemonics provide simpler mnemonics for specifying certain **sync** operations:

Table C-36. Sync Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to
Lightweight sync	**lwsync**	**sync 1**
Heavyweight sync	**hwsync**	**sync 0**
Book E/PowerPC compatibility	**sync**	**sync 0**
	msync	

C.10.8 Integer Select (isel)

The following mnemonics simplify the most common variants of the **isel** instruction that access CR0:

Table C-37. Integer Select Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to
Integer select less than	**isellt r**D**,r**A**,r**B	**isel r**D**,r**A**,r**B**,0**
Integer select greater than	**iselgt r**D**,r**A**,r**B	**isel r**D**,r**A**,r**B**,1**
Integer select equal	**iseleq r**D**,r**A**,r**B	**isel r**D**,r**A**,r**B**,2**

C.10.9 System Call (sc)

The following mnemonics are provided for **sc** encodings:

Table C-38. System Call Simplified Mnemonic

Operation	Simplified Mnemonic	Equivalent to
System call (to operating system)	**sc**	**sc 0**

C.10.10 TLB Invalidate Local Indexed

The following simplified mnemonics are provided for **tlbilx** encodings:

Table C-39. TLB Invalidate Local Indexed Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to
TLB invalidate local indexed	**tlbilxlpid**	**tlbilx 0,0**
	tlbilxpid	**tlbilx 1,0,0**
	tlbilxva rA,**r**B	**tlbilx 3,r**A,**r**B
	tlbilxva rB	**tlbilx 3,0,r**B

Appendix D
Programming Examples

This appendix gives examples of how memory synchronization instructions can be used to emulate various synchronization primitives and to provide more complex forms of synchronization. It also describes multiple precision shifts and floating-point conversions.

D.1 Synchronization

Examples in this appendix have a common form. After possible initialization, a conditional sequence begins with a load and reserve instruction that may be followed by memory accesses and computations that include neither a load and reserve nor a store conditional. The sequence ends with a store conditional with the same target address as the initial load and reserve. In most of the examples, failure of the store conditional causes a branch back to the load and reserve for a repeated attempt. On the assumption that contention is low, the conditional branch in the examples is optimized for the case in which the store conditional succeeds, by setting the branch-prediction bit appropriately. These examples focus on techniques for the correct modification of shared memory locations: see note 4 in Section D.1.4, "Synchronization Notes," for a discussion of how the retry strategy can affect performance.

Load and reserve and store conditional instructions depend on the coherence mechanism of the system. Stores to a given location are coherent if they are serialized in some order, and no processor is able to observe a subset of those stores as occurring in a conflicting order. See Section 6.4.8.1, "Memory Access Ordering," for details.

Each load operation, whether ordinary or load and reserve, returns a value that has a well-defined source. The source can be the store or store conditional instruction that wrote the value, an operation by some other mechanism that accesses memory (for example, an I/O device), or the initial state of memory.

The function of an atomic read/modify/write operation is to read a location and write its next value, possibly as a function of its current value, all as a single atomic operation. We assume that locations accessed by read/modify/write operations are accessed coherently, so the concept of a value being the next in the sequence of values for a location is well defined. The conditional sequence, as defined above, provides the effect of an atomic read/modify/write operation, but not with a single atomic instruction. Let *addr* be the location that is the common target of the load and reserve and store conditional instructions. Then the guarantee the architecture makes for the successful execution of the conditional sequence is that no store into *addr* by another processor or mechanism has intervened between the source of the load and reserve and the store conditional.

For each of these examples, it is assumed that a similar sequence of instructions is used by all processes requiring synchronization on the accessed data.

NOTE

Because memory synchronization instructions have implementation dependencies (for example, the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (such as, test and set or compare and swap) needed by application programs. Application programs should use these library programs, rather than use memory synchronization instructions directly.

D.1.1 Synchronization Primitives

The following examples show how the **lwarx** and **stwcx.** instructions can be used to implement various synchronization primitives. Implementations may provide other sizes of load and reserve and store conditional instructions (byte, halfword, or doubleword) and those instructions may be used in place of the **lwarx**/**stwcx.** instructions in the examples. If other sizes are used, the same operand size must be used for both the load and reserve and store conditional instructions.

The sequences used to emulate the various primitives consist primarily of a loop using **lwarx** and **stwcx.**. No additional synchronization is necessary, because the **stwcx.** will fail, clearing EQ, if the word loaded by **lwarx** has changed before the **stwcx.** is executed: see Section 4.6.1.16, "Atomic Update Primitives Using **lwarx** and **stwcx.**," for details.

The examples assume that all memory references are to well behaved coherent memory and the least efficient (but supported by all processors) memory barriers are shown. Older processors may not implement newer, more efficient memory barriers shown in comments in the code examples. Using the newer more efficient memory barriers can improve performance. If all memory references are not to well behaved coherent memory, different algorithms may be required or which memory barriers are used may change.

D.1.1.1 Memory Barriers

Many of the examples below require memory barriers to ensure the order of reads or writes (loads or stores). The choice of which barrier to use will depend on the storage attributes of the memory locations, the type of accesses (load or store) to be performed, whether the implementation supports a particular barrier, and how the barrier performs on the implementation. Since newer, more efficient memory barriers have been defined as the architecture has evolved, not all barriers will be supported by all implementations. However, at a minimum, all implementations support **sync 0** (**msync**) and **mbar 0**.For a more detailed description of memory barriers and storage ordering see Section 6.4.8, "Shared Memory". Consult the core reference manual for information about which memory barriers are supported and any performance considerations.

D.1.1.2 Fetch and No-op

The fetch and no-op primitive atomically loads the current value in a word in memory.

In this example it is assumed that the address of the word to be loaded is in GPR3 and the data loaded are returned in GPR4.

```
loop:    lwarx     r4,0,r3      #load and reserve
         stwcx.    r4,0,r3      #store old value if still reserved
         bc        4,2,loop     #loop if lost reservation
```

If the **stwcx.** succeeds, it stores to the target location the same value that was loaded by the preceding **lwarx**. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the **lwarx** was the current value, that is, that the source of the value loaded by the **lwarx** was the last store to the location that preceded the **stwcx.** in the coherence order for the location.

D.1.1.3 Fetch and Store

The fetch and store primitive atomically loads and replaces a word in memory. In this example it is assumed that the address of the word to be loaded and replaced is in GPR3, the new value is in GPR4, and the old value is returned in GPR5.

```
loop:    lwarx     r5,0,r3      #load and reserve
         stwcx.    r4,0,r3      #store new value if still reserved
         bc        4,2,loop     #loop if lost reservation
```

D.1.1.4 Fetch and Add

The fetch and add primitive atomically increments a word in memory. In this example it is assumed that the address of the word to be incremented is in GPR3, the increment is in GPR4, and the old value is returned in GPR5.

```
loop:    lwarx     r5,0,r3      #load and reserve
         add       r0,r4,r5     #increment word
         stwcx.    r0,0,r3      #store new value if still reserved
         bc        4,2,loop     #loop if lost reservation
```

D.1.1.5 Fetch and AND

The Fetch and AND primitive atomically ANDs a value into a word in memory.

In this example it is assumed that the address of the word to be ANDed is in GPR3, the value to AND into it is in GPR4, and the old value is returned in GPR5.

```
loop:    lwarx     r5,0,r3      #load and reserve
         and       r0,r4,r5     #AND word
         stwcx.    r0,0,r3      #store new value if still reserved
         bc        4,2,loop     #loop if lost reservation
```

This sequence can be changed to perform another Boolean operation atomically on a word in memory by changing the **and** to the desired Boolean instruction (**or**, **xor**, etc.).

D.1.1.6 Test and Set

This version of the test and set primitive atomically loads a word from memory, sets the word in memory to a nonzero value if the value loaded is zero, and sets the EQ bit of CR Field 0 to indicate whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR3, the new value (nonzero) is in GPR4, and the old value is returned in GPR5.

```
loop:    lwarx     r5,0,r3       #load and reserve
         cmpwi     r5,0          #done if word
         bc        4,2,done      #not equal to 0
         stwcx.    r4,0,r3       #try to store non-0
         bc        4,2,loop      #loop if lost reservation
done:
```

D.1.1.7 Compare and Swap

The compare and swap primitive atomically compares a value in a register with a word in memory, if they are equal stores the value from a second register into the word in memory, if they are unequal loads the word from memory into the first register, and sets CR0[EQ] to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR3, the comparand is in GPR4 and the old value is returned there, and the new value is in GPR5.

```
loop:    lwarx     r6,0,r3       #load and reserve
         cmpw      r4,r6         #1st 2 operands equal?
         bc        4,2,exit      #skip if not
         stwcx.    r5,0,r3       #store new value if still reserved
         bc        4,2,loop      #loop if lost reservation
exit:    or        r4,r6,r6      #return value from memory
```

Notes:

1. The semantics given for compare and swap above are based on those of the IBM System/370 compare and swap instruction. Other architectures may define a compare and swap instruction differently.
2. Compare and swap is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by **lwarx** and **stwcx.**. A major weakness of a System/370-style compare and swap instruction is that, although the instruction itself is atomic, it checks only that the old and current values of the word being tested are equal, with the result that programs that use such a compare and swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The sequence shown above has the same weakness.
3. In some applications the second **bc** and/or the **or** can be omitted. The **bc** is needed only if the application requires that if CR0[EQ] on exit indicates not equal then GPR4 and GPR6 are not equal. The **or** is needed only if the application requires that if the comparands are not equal then the word from memory is loaded into the register with which it was compared (rather than into a third register). If any of these instructions is omitted, the resulting compare and swap does not obey System/370 semantics.

D.1.2 Lock Acquisition and Release

This example gives an algorithm for locking that demonstrates the use of synchronization with an atomic read/modify/write operation. A shared memory location, the address of which is an argument of the lock and unlock procedures, given by GPR3, is used as a lock, to control access to some shared resource such as a shared data structure. The lock is open when its value is 0 and closed (locked) when its value is 1. Before accessing the shared resource the program executes the lock procedure, which sets the lock by changing its value from 0 to 1. To do this, the lock procedure calls test_and_set, which executes the code sequence shown in the test and set example of Section D.1.1, “Synchronization Primitives,” thereby atomically loading the old value of the lock, writing to the lock the new value (1) given in GPR4, returning the old value in GPR5 (not used below), and setting the EQ bit of CR Field 0 according to whether the value loaded is 0. The lock procedure repeats the test_and_set until it succeeds in changing the value of the lock from 0 to 1.

Because the shared resource must not be accessed until the lock has been set, the lock procedure contains a method to ensure that any reads to the shared resource must occur after the lock has been acquired. This requries that the lock procedure order the subsequent load instructions of the shared resource after the read of the lock or to discard the reads of the shared resource if they were loaded speculatively. An **isync** after the **bc** that checks for the success of test_and_set will discard any subsequent speculative loads and require the loads to be performed again after the lock has been acquired. The **isync** delays all subsequent instructions until all preceding instructions have completed.

Alternatively, a memory barrier can be used instead of an **isync**.

```
lock:    mfspr     r6,LR         #save Link Register
         addi      r4,r0,1       #obtain lock:
loop:    bl        test_and_set  #  test-and-set
         bc        4,2,loop      #  retry til old = 0
# Delay subsequent instructions til prior instructions finish
         isync                   # (or use memory barrier: sync, mbar, lwsync)
         mtspr     LR,r6         #restore Link Register
         blr                     #return
```

The unlock procedure stores a 0 to the lock location. Most applications that use locking require, for correctness, that if the access to the shared resource includes stores, the program must perform a memory barrier instruction that orders the stores to the shared resources and the store to the lock which releases the lock. The memory barrier ensures that the program’s modifications are performed with respect to other processors before the store that releases the lock is performed with respect to those processors. In this example, the unlock procedure begins with a memory barrier for this purpose. **sync 0** is shown in the example, however any memory barrier may be used that orders stores with stores for the appropriate storage attributes of the memory locations.

```
unlock:  sync      0             #(or use memory barrier: mbar, lwsync)
         addi      r1,r0,0       #before lock release
         stw       r1,0(r3)      #store 0 to lock location
         blr                     #return
```

D.1.3 List Insertion

This example shows how **lwarx** and **stwcx.** can be used to implement simple insertion into a singly linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below and requires a more complicated strategy such as using locks.)

The next element pointer from the list element after which the new element is to be inserted, here called the parent element, is stored into the new element, so that the new element points to the next element in the list: this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR3, the address of the new element is in GPR4, and the next element pointer is at offset 0 from the start of the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements: see Section 4.6.1.16, “Atomic Update Primitives Using **lwarx** and **stwcx.**”

```
    loop:    lwarx     r2,0,r3       #get next pointer
             stw       r2,0(r4)      #store in new element
             sync      0             #order stw before stwcx.(can omit if not MP)
                                     # (or use memory barrier: mbar, lwsync)
             stwcx.    r4,0,r3       #add new element to list
             bc        4,2,loop      #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, livelock can occur. (Livelock is a state in which processors interact in a way such that no processor makes progress.)

If list elements cannot be allocated such that each element’s next element pointer is in a different reservation granule, livelock can be avoided with this more complicated sequence:

```
             lwz       r2,0(r3)      #get next pointer
    loop1:   or        r5,r2,r2      #keep a copy
             stw       r2,0(r4)      #store in new element
             sync      0             #order stw before stwcx.
                                     # (or use memory barrier: mbar, lwsync)
    loop2:   lwarx     r2,0,r3       #get it again
             cmpw      r2,r5         #loop if changed (someone
             bc        4,2,loop1     #  else progressed)
             stwcx.    r4,0,r3       #add new element to list
             bc        4,2,loop2     #loop if failed
```

D.1.4 Synchronization Notes

1. In general, **lwarx** and **stwcx.** should be paired, with the same effective address used for both. The only exception is that an unpaired **stwcx.** to any (scratch) effective address can be used to clear any reservation held by the processor.
2. It is acceptable to execute a **lwarx** for which no **stwcx.** is executed. For example, this occurs in the test and set sequence shown above if the value loaded is not zero.
3. To increase the likelihood that forward progress is made, it is important that looping on **lwarx**/**stwcx.** pairs be minimized. For example, in the sequence shown above for test and set, this is achieved by testing the old value before attempting the store: were the order reversed, more **stwcx.** instructions might be executed, and reservations might more often be lost between the **lwarx** and the **stwcx.**.
4. The manner in which **lwarx** and **stwcx.** are communicated to other processors and mechanisms, and between levels of the memory subsystem within a given processor is implementation-dependent (see Section 4.6.1.16, “Atomic Update Primitives Using **lwarx** and **stwcx.**”). In some implementations performance may be improved by minimizing looping on a **lwarx** instruction that fails to return a desired value. For example, in the test and set example shown above, to stay in the loop until the word loaded is zero, `bne-  $+12` can be changed to `bne-loop`. However, in some implementations better performance may be obtained by using an ordinary load instruction to do the initial checking of the value, as follows.

```
loop:    lwz       r5,0(r3)      #load the word
         cmpi      cr0,0,r5,0    #loop back if word
         bc        4,2,loop      #  not equal to 0
         lwarx     r5,0,r3       #try again, reserving
         cmpi      cr0,0,r5,0    #  (likely to succeed)
         bc        4,2,loop
         stwcx.    r4,0,r3       #try to store non-0
         bc        4,2,loop      #loop if lost reservation
```

5. In a multiprocessor, livelock is possible if a loop containing a **lwarx**/**stwcx.** pair also contains an ordinary store instruction for which any byte of the affected memory area is in the reservation granule: see Section 4.6.1.16, “Atomic Update Primitives Using **lwarx** and **stwcx.**.” For example, the first code sequence shown in Section D.1.3, “List Insertion,” can cause livelock if two list elements have next element pointers in the same reservation granule.

D.2 Multiple-Precision Shifts

This section gives examples of how multiple-precision shifts can be programmed.

A multiple-precision shift is defined to be a shift of an N-doubleword quantity (64-bit implementations) or an N-word quantity (32-bit implementations), where N>1. The quantity to be shifted is contained in N registers. The shift amount is specified either by an immediate value in the instruction or by a value in a register.

The examples shown below distinguish between the cases N=2 and N>2. If N=2, the shift amount may be in the range 0 through 127 (64-bit implementations) or 0–63 (32-bit implementations), which are the maximum ranges supported by the *Shift* instructions used. However if N>2, the shift amount must be in the range 0 through 63 (64-bit implementations) or 0–31 (32-bit implementations) for the examples to yield the desired result. The specific instance shown for N>2 is N=3: extending those code sequences to larger N is straightforward, as is reducing them to the case N=2 when the more stringent restriction on shift amount is met. For shifts with immediate shift amounts only the case N=3 is shown, because the more stringent restriction on shift amount is always met.

In the examples it is assumed that GPRs 2 and 3 (and 4) contain the quantity to be shifted, and that the result is to be placed into the same registers, except for the immediate left shifts in 64-bit implementations, for which the result is placed into GPRs 3, 4, and 5. In all cases, for both input and result, the lowest-numbered register contains the highest-order part of the data and highest-numbered register contains the lowest-order part. For non-immediate shifts, the shift amount is assumed to be in GPR6. For immediate shifts, the shift amount is assumed to be greater than 0. GPRs 0 and 31 are used as scratch registers.

For N>2, the number of instructions required is 2N–1 (immediate shifts) or 3N–1 (non-immediate shifts).

Table D-1. Shifts in 32- and 64-Bit Implementations

64-Bit Implementations	32-Bit Implementations

Shift Left Immediate, N=3 (shift amount < 64)
`rldicr r5,r4,sh,63-sh`
`rldimi r4,r3,0,sh`
`rldicl r4,r4,sh,0`
`rldimi r3,r2,0,sh`
`rldicl r3,r3,sh,0`
	Shift Left Immediate, N=3 (shift amount < 32)
	`rlwinm r2,r2,sh,0,31-sh`
	`rlwimi r2,r3,sh,32-sh,31`
	`rlwinm r3,r3,sh,0,31-sh`
	`rlwimi r3,r4,sh,32-sh,31`
	`rlwinm r4,r4,sh,0,31-sh`

Shift Left, N=2 (shift amount < 128)
`subfic r31,r6,64`
`sld r2,r2,r6`
`srd r0,r3,r31`
`or r2,r2,r0`
`addi r31,r6,-64`
`sld r0,r3,r31`
`or r2,r2,r0`
`sld r3,r3,r6`
	Shift Left, N=2 (shift amount < 64)
	`subfic r31,r6,32`
	`slw r2,r2,r6`
	`srw r0,r3,r31`
	`or r2,r2,r0`
	`addi r31,r6,-32`
	`slw r0,r3,r31`
	`or r2,r2,r0`
	`slw r3,r3,r6`

Shift Left, N=3 (shift amount < 64)
`subfic r31,r6,64`
`sld r2,r2,r6`
`srd r0,r3,r31`
`or r2,r2,r0`
`sld r3,r3,r6`
`srd r0,r4,r31`
`or r3,r3,r0`
`sld r4,r4,r6`
	Shift Left, N=3 (shift amount < 32)
	`subfic r31,r6,32`
	`slw r2,r2,r6`
	`srw r0,r3,r31`
	`or r2,r2,r0`
	`slw r3,r3,r6`
	`srw r0,r4,r31`
	`or r3,r3,r0`
	`slw r4,r4,r6`

Shift Right Immediate, N=3 (shift amount < 64)
`rldimi r4,r3,0,64-sh`
`rldicl r4,r4,64-sh,0`
`rldimi r3,r2,0,64-sh`
`rldicl r3,r3,64-sh,0`
`rldicl r2,r2,64-sh,sh`
	Shift Right Immediate, N=3 (shift amount < 32)
	`rlwinm r4,r4,32-sh,sh,31`
	`rlwimi r4,r3,32-sh,0,sh-1`
	`rlwinm r3,r3,32-sh,sh,31`
	`rlwimi r3,r2,32-sh,0,sh-1`
	`rlwinm r2,r2,32-sh,sh,31`

Shift Right, N=2 (shift amount < 128)
`subfic r31,r6,64`
`srd r3,r3,r6`
`sld r0,r2,r31`
`or r3,r3,r0`
`addi r31,r6,-64`
`srd r0,r2,r31`
`or r3,r3,r0`
`srd r2,r2,r6`
	Shift Right, N=2 (shift amount < 64)
	`subfic r31,r6,32`
	`srw r3,r3,r6`
	`slw r0,r2,r31`
	`or r3,r3,r0`
	`addi r31,r6,-32`
	`srw r0,r2,r31`
	`or r3,r3,r0`
	`srw r2,r2,r6`

Shift Right, N=3 (shift amount < 64)
`subfic r31,r6,64`
`srd r4,r4,r6`
`sld r0,r3,r31`
`or r4,r4,r0`
`srd r3,r3,r6`
`sld r0,r2,r31`
`or r3,r3,r0`
`srd r2,r2,r6`
	Shift Right, N=3 (shift amount < 32)
	`subfic r31,r6,32`
	`srw r4,r4,r6`
	`slw r0,r3,r31`
	`or r4,r4,r0`
	`srw r3,r3,r6`
	`slw r0,r2,r31`
	`or r3,r3,r0`
	`srw r2,r2,r6`

Table D-1. Shifts in 32- and 64-Bit Implementations (continued)

64-Bit Implementations	32-Bit Implementations

Shift Right Algebraic Immediate, N=3 (shift amnt < 64)
`rldimi r4,r3,0,64-sh`
`rldicl r4,r4,64-sh,0`
`rldimi r3,r2,0,64-sh`
`rldicl r3,r3,64-sh,0`
`sradi r2,r2,sh`
	Shift Right Algebraic Immediate, N=3 (shift amnt < 32)
	`rlwinm r4,r4,32-sh,sh,31`
	`rlwimi r4,r3,32-sh,0,sh-1`
	`rlwinm r3,r3,32-sh,sh,31`
	`rlwimi r3,r2,32-sh,0,sh-1`
	`srawi r2,r2,sh`

Shift Right Algebraic, N=2 (shift amount < 128)
`subfic r31,r6,64`
`srd r3,r3,r6`
`sld r0,r2,r31`
`or r3,r3,r0`
`addic. r31,r6,-64`
`srad r0,r2,r31`
`bc 4,1,$+8`
`ori r3,r0,0`
`srad r2,r2,r6`
	Shift Right Algebraic, N=2 (shift amount < 64)
	`subfic r31,r6,32`
	`srw r3,r3,r6`
	`slw r0,r2,r31`
	`or r3,r3,r0`
	`addic. r31,r6,-32`
	`sraw r0,r2,r31`
	`bc 4,1,$+8`
	`ori r3,r0,0`
	`sraw r2,r2,r6`

Shift Right Algebraic, N=3 (shift amount < 64)
`subfic r31,r6,64`
`srd r4,r4,r6`
`sld r0,r3,r31`
`or r4,r4,r0`
`srd r3,r3,r6`
`sld r0,r2,r31`
`or r3,r3,r0`
`srad r2,r2,r6`
	Shift Right Algebraic, N=3 (shift amount < 32)
	`subfic r31,r6,32`
	`srw r4,r4,r6`
	`slw r0,r3,r31`
	`or r4,r4,r0`
	`srw r3,r3,r6`
	`slw r0,r2,r31`
	`or r3,r3,r0`
	`sraw r2,r2,r6`

D.3 Floating-Point Conversions <FP>

This section gives examples of how floating-point conversion instructions can be used to perform various conversions.

NOTE

Some of the examples use the optional **fsel** instruction. Care must be taken in using **fsel** if IEEE-754 compatibility is required, or if the values being tested can be NaNs or infinities: see Section D.4.1, “Floating-Point Selection Notes.”

D.3.1 Conversion from a Floating-Point Number to Floating-Point Integer

The full convert to floating-point integer function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR1 and the result is returned in FPR3.

```
         mtfsb0   23       #clear VXCVI
         fctid[z] f3,f1    #convert to integer
         fcfid    f3,f3    #convert back again
         mcrfs    7,5      #VXCVI to CR
         bc       4,31,continue#skip if VXCVI was 0
         fmr      f3,f1    #input was fp integer
continue:
```

D.3.2 Conversion from Floating-Point Number to Signed Integer Doubleword <64>

The full convert to signed integer doubleword function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR1, the result is returned in GPR3, and a doubleword at displacement ‘disp’ from the address in GPR1 can be used as scratch space.

```
fctid[z] f2,f1            #convert to integer
stfd     f2,disp(r1)      #store float
ld       r3,disp(r1)      #load doubleword
```

D.3.3 Conversion from Floating-Point Number to Unsigned Integer Doubleword <64>

The full convert to unsigned integer doubleword function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR1, the value 0 is in FPR0, the value 2^{64}–2048 is in FPR3, the value 2^{63} is in FPR4 and GPR4, the result is returned in GPR3, and a doubleword at displacement ‘disp’ from the address in GPR1 can be used as scratch space.

```
fsel     f2,f1,f1,f0      #use 0 if < 0
fsub     f5,f3,f1         #use max if > max
fsel     f2,f5,f2,f3
fsub     f5,f2,f4         #subtract $2^{63}$
fcmpu    cr2,f2,f4        #use diff if ≥ $2^{63}$
fsel     f2,f5,f5,f2
fctid[z] f2,f2            #convert to integer
stfd     f2,disp(r1)      #store float
ld       r3,disp(r1)      #load dword
bc       12,8,$+8         #add $2^{63}$ if input
add      r3,r3,r4         #  was ≥ $2^{63}$
```

D.3.4 Conversion from Floating-Point Number to Signed Integer Word

The full convert to signed integer word function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR1, the result is returned in GPR3, and a double word at displacement ‘disp’ from the address in GPR1 can be used as scratch space.

```
fctiw[z] f2,f1            #convert to integer
stfd     f2,disp(r1)      #store float
lwa      r3,disp+4(r1)    #load word algebraic
                          #(use lwz on a 32-bit implementation)
```

D.3.5 Conversion from Floating-Point Number to Unsigned Integer Word

In a 32-bit implementation

The full convert to unsigned integer word function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR1, the value 0 is in FPR0, the value 2^{32}–1 is in FPR3, the value 2^{31} is in FPR4, the result is returned in GPR3, and a doubleword at displacement ‘disp’ from the address in GPR1 can be used as scratch space.

```
fsel     f2,f1,f1,f0      #use 0 if < 0
fsub     f5,f3,f1         #use max if > max
fsel     f2,f5,f2,f3
fsub     f5,f2,f4         #subtract $2^{31}$
```

```
fcmpu    cr2,f2,f4          #use diff if ≥ $2^{31}$
fsel     f2,f5,f5,f2
fctiw[z] f2,f2              #convert to integer
stfd     f2,disp(r1)        #store float
lwz      r3,disp+4(r1)      #load word
bc       12,8,$+8           #add $2^{31}$ if input
xoris    r3,r3,0x8000       #  was ≥ $2^{31}$
```

In a 64-bit implementation

The full convert to unsigned integer word function can be implemented with the sequence shown below, assuming the floating-point value to be converted is in FPR1, the value 0 is in FPR0, the value 2^{32}–1 is in FPR3, and the result is returned in GPR3, and a doubleword at displacement ‘disp’ from the address in GPR1 can be used as scratch space.

```
fsel     f2,f1,f1,f0        #use 0 if < 0
fsub     f4,f3,f1           #use max if > max
fsel     f2,f4,f2,f3
fctid[z] f2,f2              #convert to integer
stfd     f2,disp(r1)        #store float
lwz      r3,disp+4(r1)      #load word
```

D.3.6 Conversion from Signed Integer Doubleword to Floating-Point Number <64>

The full convert from signed integer doubleword function, using the rounding mode specified by FPSCR[RN], can be implemented with the sequence shown below, assuming the integer value to be converted is in GPR3, the result is returned in FPR1, and a doubleword at displacement ‘disp’ from the address in GPR1 can be used as scratch space.

```
std      r3,disp(r1)        #store dword
lfd      f1,disp(r1)        #load float
fcfid    f1,f1              #convert to fp int
```

D.3.7 Conversion from Unsigned Integer Doubleword to Floating-Point Number <64>

The full convert from unsigned integer doubleword function, using the rounding mode specified by FPSCR[RN], can be implemented with the sequence shown below, assuming the integer value to be converted is in GPR3, the value 2^{32} is in FPR4, the result is returned in FPR1, and two doublewords at displacement ‘disp’ from the address in GPR1 can be used as scratch space.

```
rldicl   r2,r3,32,32        #isolate high half
rldicl   r0,r3,0,32         #isolate low half
std      r2,disp(r1)        #store dword
std      r0,disp+8(r1)      #store dword
lfd      f2,disp(r1)        #load float
lfd      f1,disp+8(r1)      #load float
fcfid    f2,f2              #convert each half to fp int
fcfid    f1,f1              #
fmadd    f1,f4,f2,f1        #($2^{32}$) * high + low
```

D.3.8 Conversion from Signed Integer Word to Floating-Point Number <64>

The full convert from signed integer word function, using the rounding mode specified by FPSCR[RN], can be implemented with the sequence shown below, assuming the integer value to be converted is in GPR3, the result is returned in FPR1, and a doubleword at displacement ‘disp’ from the address in GPR1 can be used as scratch space. (The result is exact.)

```
extsw    r3,r3              #extend sign
std      r3,disp(r1)        #store dword
lfd      f1,disp(r1)        #load float
fcfid    f1,f1              #convert to fp int
```

D.3.9 Conversion from Unsigned Integer Word to Floating-Point Number <64>

The full convert from unsigned integer word function, using the rounding mode specified by FPSCR[RN], can be implemented with the sequence shown below, assuming the integer value to be converted is in GPR3, the result is returned in FPR1, and a doubleword at displacement ‘disp’ from the address in GPR1 can be used as scratch space.

```
rldicl   r0,r3,0,32         #zero extend
std      r0,disp(r1)        #store dword
lfd      f1,disp(r1)        #load float
fcfid    f1,f1              #convert to fp int
```

D.4 Floating-Point Selection

This section gives examples of how the optional floating select instruction (**fsel**) can be used to implement floating-point minimum and maximum functions, and certain simple forms of if-then-else constructions, without branching.

The examples show program fragments in an imaginary, C-like, high-level programming language, and the corresponding program fragment using **fsel** and other instructions. In the examples, a, b, x, y, and z are floating-point variables, which are assumed to be in FPRs fa, fb, fx, fy, and fz. FPR fs is assumed to be available for scratch space.

Additional examples can be found in Section D.3, “Floating-Point Conversions <FP>.”

Warning: Care must be taken in using **fsel** if IEEE-754 compatibility is required, or if the values being tested can be NaNs or infinities: see Section D.4.1, “Floating-Point Selection Notes.”

Table D-2. Comparison to Zero

High-Level Language:	Power ISA:	Notes

`if a ≥ 0.0 then x ← y`
`         else x ← z`
	`fsel   fx,fa,fy,fz`	(1)

Table D-2. Comparison to Zero (continued)

High-Level Language:	Power ISA:	Notes

if a > 0.0 then x ← y
else x ← z
	fneg fs,fa
	fsel fx,fs,fz,fy
		(1,2)

if a = 0.0 then x ← y
else x ← z
	fsel fx,fa,fy,fz
	fneg fs,fa
	fsel fx,fs,fx,fz
		(1)

Table D-3. Minimum and Maximum

High-Level Language:	Power ISA:	Notes
x ← min(a,b)	fsub fs,fa,fb fsel fx,fs,fb,fa	(3,4,5)
x ← max(a,b)	fsub fs,fa,fb fsel fx,fs,fa,fb	(3,4,5)

Table D-4. Simple If-Then-Else Constructions

High-Level Language:	Power ISA:	Notes

if a ≥ b then x ← y
else x ← z
	fsub fs,fa,fb
	fsel fx,fs,fy,fz
		(4,5)

if a > b then x ← y
else x ← z
	fsub fs,fb,fa
	fsel fx,fs,fz,fy
		(3,4,5)

if a = b then x ← y
else x ← z
	fsub fs,fa,fb
	fsel fx,fs,fy,fz
	fneg fs,fs
	fsel fx,fs,fx,fz
		(4,5)

D.4.1 Floating-Point Selection Notes

The following notes apply to the preceding examples and to the corresponding cases using the other three arithmetic relations (<, ≤, and ≠). They should also be considered when any other use of **fsel** is contemplated.

In these notes, the optimized program is the Power ISA program shown, and the unoptimized program (not shown) is the corresponding Power ISA program that uses **fcmpu** and branch conditional instructions instead of **fsel**.

1. The unoptimized program affects FPSCR[VXSNAN] and therefore may cause the system error handler to be invoked if the corresponding exception is enabled; the optimized program does not affect this bit. This property of the optimized program is incompatible with the IEEE-754 standard.
2. The optimized program gives the incorrect result if *a* is a NaN.
3. The optimized program gives the incorrect result if *a* and/or *b* is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).

4. The optimized program gives the incorrect result if *a* and *b* are infinities of the same sign. (Here it is assumed that invalid operation exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if invalid operation exceptions are enabled, because in that case the target register of the subtraction is unchanged.)
5. The optimized program affects FPSCR[OX, UX, XX,VXISI], and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled; the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE-754 standard.

Appendix E
Floating-Point Models

This appendix describes the execution model for IEEE-754 operations and gives examples of how the floating-point conversion instructions can be used to perform various conversions as well as providing models for floating-point instructions.

E.1 Execution Model for IEEE-754 Operations

The following description uses double-precision arithmetic as an example; single-precision arithmetic is similar except that the fraction field is a 23-bit field and the single-precision guard, round, and sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION field. IEEE-754-conforming significand arithmetic is performed with a floating-point accumulator where bits 0–55, shown in Figure E-1, comprise the significand of the intermediate result.

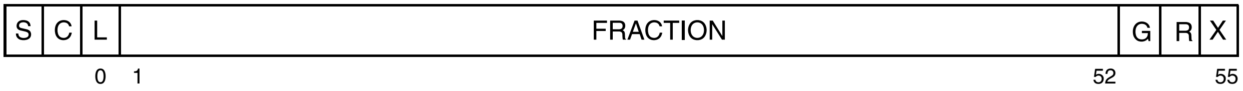


Figure E-1. IEEE-754 64-Bit Execution Model

The bits and fields for the IEEE-754 double-precision execution model are defined in Table E-1.

Table E-1. IEEE-754 64-Bit Execution Model Field Descriptions

Bits	Description
S	Sign bit.
C	Carry bit that captures the carry out of the significand.
L	Leading unit bit of the significand that receives the implicit bit from the operands.
FRACTION	52-bit field that accepts the fraction of the operands.
G R X	Guard, round, and sticky. These bits are extensions to the low-order accumulator bits. G and R are required for post normalization of the result. G, R, and X are required during rounding to determine if the intermediate result is equally near the two nearest representable values. This is shown in Table E-2. X serves as an extension to the G and R bits by representing the logical OR of all bits that may appear to the low-order side of the R bit, due to either shifting the accumulator right or to other generation of low-order result bits. G and R participate in the left shifts with zeros being shifted into the R bit.

Table E-2 shows the significance of G, R, and bits with respect to the intermediate result (IR), the next lower in magnitude representable number (NL), and the next higher in magnitude representable number (NH).

Table E-2. Interpretation of G, R, and X Bits

G	R	X	Interpretation
0	0	0	IR is exact
0	0	1	
0	1	0	IR closer to NL
0	1	1	
1	0	0	IR midway between NL & NH
1	0	1	
1	1	0	IR closer to NH
1	1	1	

The significand of the intermediate result is made up of the L bit, the FRACTION, and the G, R, and X bits.

The infinitely precise intermediate result of an operation is the result normalized in bits L, FRACTION, G, R, and X of the floating-point accumulator.

After normalization, the intermediate result is rounded, using the rounding mode specified by FPSCR[RN]. If rounding causes a carry into C, the significand is shifted right one position and the exponent is incremented by one. This causes an inexact result and possibly exponent overflow. Fraction bits to the left of the bit position used for rounding are stored into the FPR, and low-order bit positions, if any, are set to zero.

Four user-selectable rounding modes are provided through FPSCR[RN] as described in Section 3.4.3.6, “Rounding.” For rounding, the conceptual guard, round, and sticky bits are defined in terms of accumulator bits.

Table E-3 shows the positions of the guard, round, and sticky bits for double-precision and single-precision floating-point numbers in the IEEE-754 execution model.

Table E-3. Location of the Guard, Round, and Sticky Bits—IEEE-754 Execution Model

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	OR of 26–52 G,R,X

Rounding can be treated as though the significand were shifted right, if required, until the least-significant bit to be retained is in the low-order bit position of the FRACTION. If any of the guard, round, or sticky bits are nonzero, the result is inexact.

Z1 and Z2, defined in Section 3.4.3.6, “Rounding,” can be used to approximate the result in the target format when one of the following rules is used:

- Round to nearest
  - — Guard bit = 0: The result is truncated. (Result exact (GRX = 000) or closest to next lower value in magnitude (GRX = 001, 010, or 011).
  - — Guard bit = 1: Depends on round and sticky bits:

    Case a: If the round or sticky bit is one (inclusive), the result is incremented (result closest to next higher value in magnitude (GRX = 101, 110, or 111)).

    Case b: If the round and sticky bits are zero (result midway between closest representable values) then if the low-order bit of the result is one, the result is incremented. Otherwise (the low-order bit of the result is zero) the result is truncated (this is the case of a tie rounded to even).

  If during the round-to-nearest process, truncation of the unrounded number produces the maximum magnitude for the specified precision, the following occurs:
  - — Guard bit = 1: Store infinity with the sign of the unrounded result.
  - — Guard bit = 0: Store the truncated (maximum magnitude) value.
- Round toward zero—Choose the smaller in magnitude of Z1 or Z2. If the guard, round, or sticky bit is nonzero, the result is inexact.
- Round toward +infinity—Choose Z1.
- Round toward –infinity—Choose Z2.

Where a result is to have fewer than 53 bits of precision because the instruction is a floating round to single-precision or single-precision arithmetic instruction, the intermediate result either is normalized or is placed in correct denormalized form before being rounded.

E.2 Multiply-Add Type Instruction Execution Model

The architecture uses of a special instruction form that performs up to three operations in one instruction (a multiply, an add, and a negate). With this comes an ability to produce a more exact intermediate result as an input to the rounder. Single-precision arithmetic is similar except that the fraction field is smaller. Note that rounding occurs only after add; therefore, the computation of the sum and product together are infinitely precise before the final result is rounded to a representable format.

As Figure E-2 shows, multiply-add significand arithmetic is considered to be performed with a floating-point accumulator, where bits 1–106 comprise the significand of the intermediate result.

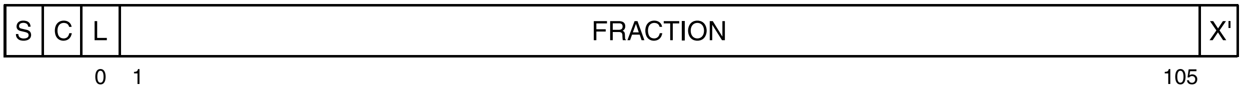


Figure E-2. Multiply-Add 64-Bit Execution Model

The first part of the operation is a multiply. The multiply has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), the significand is shifted right one position, placing the L bit into the most-significant bit of the FRACTION and placing the C bit into the L bit. All 106 bits (L bit plus the fraction) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X' bit. The add operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the add is then normalized, with all bits of the add result, except the X' bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual guard, round, and sticky bits are defined in terms of accumulator bits. Figure E-4 shows the positions of these bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

Table E-4. Location of G, R, and X Bits—Multiply-Add Execution Model

Format	Guard	Round	Sticky
Double	53	54	OR of 55–105, X'
Single	24	25	OR of 26–105, X'

The rules for rounding the intermediate result are the same as those given in Section E.1, "Execution Model for IEEE-754 Operations."

If the instruction is floating negative multiply-add or floating negative multiply-subtract, the final result is negated.

Floating-point multiply-add instructions combine a multiply and an add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide, and all 106 bits take part in the add/subtract portion of the instruction.

Status bits are set as follows:

- Overflow, underflow, and inexact exception bits, the FR and FI bits, and the FPRF field are set based on the final result of the operation, not on the result of the multiplication.
- Invalid operation exception bits are set as if the multiplication and the addition were performed using two separate instructions (for example, an **fmul** instruction followed by an **fadd** instruction). That is, multiplication of infinity by 0 or of anything by an SNaN, causes the corresponding exception bits to be set.

E.3 Floating-Point Conversions

This section provides examples of floating-point conversion instructions. Note that some of the examples use the optional Floating Select (**fsel)** instruction. Care must be taken in using **fsel** if IEEE-754 compatibility is required, or if the values being tested can be NaNs or infinities.

E.3.1 Conversion from Floating-Point Number to Signed Fixed-Point Integer Word

The full convert to signed fixed-point integer word function can be implemented with the following sequence, assuming that the floating-point value to be converted is in FPR1, the result is returned in GPR3, and a double word at displacement (disp) from the address in GPR1 can be used as scratch space.

```
fctiw[z] f2,f1            #convert to fx int
stfd     f2,disp(r1)      #store float
lwa      r3,disp + 4(r1)  #load word algebraic
                          #(use lwz on a 32-bit implementation)
```

E.3.2 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word

In a 32-bit implementation, the full convert to unsigned fixed-point integer word function can be implemented with the sequence below, assuming that the floating-point value to be converted is in FPR1, the value zero is in FPR0, the value $2^{32} – 1$ is in FPR3, the value 2^{31} is in FPR4, the result is returned in GPR3, and a double word at displacement (disp) from the address in GPR1 can be used as scratch space.

```
fsel     f2,f1,f1,f0      #use 0 if < 0
fsub     f5,f3,f1         #use max if > max
fsel     f2,f5,f2,f3
fsub     f5,f2,f4         #subtract 2**31
fcmpu    cr2,f2,f4        #use diff if ≥ 2**31
fsel     f2,f5,f5,f2
fctiw[z] f2,f2            #convert to fx int
stfd     f2,disp(r1)      #store float
lwz      r3,disp + 4(r1)  #load word
blt      cr2,$+8          #add 2**31 if input
xoris    r3,r3,0x8000     #was ≥ 2**31
```

E.4 Floating-Point Models

This section describes models for floating-point instructions.

E.4.1 Floating-Point Round to Single-Precision Model

The following describes the operation of the **frsp** instruction.

```
If frB[1–11] < 897 and frB[1–63] > 0 then
    Do
        If FPSCR[UE] = 0 then goto Disabled Exponent Underflow
        If FPSCR[UE] = 1 then goto Enabled Exponent Underflow
    End

If frB[1–11] > 1150 and frB[1–11] < 2047 then
    Do
        If FPSCR[OE] = 0 then goto Disabled Exponent Overflow
        If FPSCR[OE] = 1 then goto Enabled Exponent Overflow
    End

If frB[1–11] > 896 and frB[1–11] < 1151 then goto Normal Operand

If frB[1–63] = 0 then goto Zero Operand
If frB[1–11] = 2047 then
    Do
        If frB[12–63] = 0 then goto Infinity Operand
        If frB[12] = 1 then goto QNaN Operand
        If frB[12] = 0 and frB[13–63] > 0 then goto SNaN Operand
    End
```

Disabled Exponent Underflow:

```
sign ← frB[0]
If frB[1–11] = 0 then
    Do
        exp ← –1022
        frac[0–52] ← 0b0 || frB[12–63]
    End
If frB[1–11] > 0 then
    Do
        exp ← frB[1–11] – 1023
        frac[0–52] ← 0b1 || frB[12–63]
    End
Denormalize operand:
    G || R || X ← 0b000
    Do while exp < –126
        exp ← exp + 1
        frac[0–52] || G || R || X ← 0b0 || frac || G || (R | X)
    End
FPSCR[UX] ← frac[24–52] || G || R || X > 0
Round single(sign,exp,frac[0–52],G,R,X)
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
If frac[0–52] = 0 then
    Do
        frD[0] ← sign
        frD[1–63] ← 0
        If sign = 0 then FPSCR[FPRF] ← “+zero”
        If sign = 1 then FPSCR[FPRF] ← “–zero”
    End
If frac[0–52] > 0 then
    Do
        If frac[0] = 1 then
            Do
                If sign = 0 then FPSCR[FPRF] ← “+normal number”
                If sign = 1 then FPSCR[FPRF] ← “–normal number”
            End
        If frac[0] = 0 then
            Do
                If sign = 0 then FPSCR[FPRF] ← “+denormalized number”
                If sign = 1 then FPSCR[FPRF] ← “–denormalized number”
            End
        Normalize operand:
            Do while frac[0] = 0
                exp ← exp – 1
                frac[0–52] ← frac[1–52] || 0b0
            End
        frD[0] ← sign
        frD[1–11] ← exp + 1023
        frD[12–63] ← frac[1–52]
    End
Done
```

Enabled Exponent Underflow

```
FPSCR[UX] ← 1
sign ← frB[0]
If frB[1–11] = 0 then
```

```
      Do
         exp ← –1022
         frac[0–52] ← 0b0 || frB[12–63]
      End
If frB[1–11] > 0 then
      Do
         exp ← frB[1–11] – 1023
         frac[0–52] ← 0b1 || frB[12–63]
      End

Normalize operand:
      Do while frac[0] = 0
         exp ← exp – 1
         frac[0–52] ← frac[1–52] || 0b0
      End
Round single(sign,exp,frac[0–52],0,0,0)
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
exp ← exp + 192
frD[0] ← sign
frD[1–11] ← exp + 1023
frD[12–63] ← frac[1–52]
If sign = 0 then FPSCR[FPRF] ← "+normal number"
If sign = 1 then FPSCR[FPRF] ← "–normal number"
Done
```

Disabled Exponent Overflow

```
FPSCR[OX] ← 1
If FPSCR[RN] = 0b00 then              /* Round to Nearest */
      Do
         If frB[0] = 0 then frD ← 0x7FF0_0000_0000_0000
         If frB[0] = 1 then frD ← 0xFFF0_0000_0000_0000
         If frB[0] = 0 then FPSCR[FPRF] ← "+infinity"
         If frB[0] = 1 then FPSCR[FPRF] ← "–infinity"
      End
If FPSCR[RN] = 0b01 then              /* Round Truncate */
      Do
         If frB[0] = 0 then frD ← 0x47EF_FFFF_E000_0000
         If frB[0] = 1 then frD ← 0xC7EF_FFFF_E000_0000
         If frB[0] = 0 then FPSCR[FPRF] ← "+normal number"
         If frB[0] = 1 then FPSCR[FPRF] ← "–normal number"
      End
If FPSCR[RN] = 0b10 then              /* Round to +Infinity */
      Do
         If frB[0] = 0 then frD ← 0x7FF0_0000_0000_0000
         If frB[0] = 1 then frD ← 0xC7EF_FFFF_E000_0000
         If frB[0] = 0 then FPSCR[FPRF] ← "+infinity"
         If frB[0] = 1 then FPSCR[FPRF] ← "–normal number"
      End
If FPSCR[RN] = 0b11 then              /* Round to -Infinity */
      Do
         If frB[0] = 0 then frD ← 0x47EF_FFFF_E000_0000
         If frB[0] = 1 then frD ← 0xFFF0_0000_0000_0000
         If frB[0] = 0 then FPSCR[FPRF] ← "+normal number"
         If frB[0] = 1 then FPSCR[FPRF] ← "–infinity"
      End
FPSCR[FR] ← undefined
FPSCR[FI] ← 1
FPSCR[XX] ← 1
Done
```

Enabled Exponent Overflow

```
sign ← frB[0]
exp ← frB[1–11] – 1023
    frac[0–52] ← 0b1 || frB[12–63]
    Round single(sign,exp,frac[0–52],0,0,0)
    FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
Enabled Overflow
    FPSCR[OX] ← 1
    exp ← exp – 192
    frD[0] ← sign
    frD[1–11] ← exp + 1023
    frD[12–63] ← frac[1–52]
    If sign = 0 then FPSCR[FPRF] ← "+normal number"
    If sign = 1 then FPSCR[FPRF] ← "–normal number"
Done
```

Zero Operand

```
frD ← frB
If frB[0] = 0 then FPSCR[FPRF] ← "+zero"
If frB[0] = 1 then FPSCR[FPRF] ← "–zero"
FPSCR[FR FI] ← 0b00
Done
```

Infinity Operand

```
frD ← frB
If frB[0] = 0 then FPSCR[FPRF] ← "+infinity"
If frB[0] = 1 then FPSCR[FPRF] ← "–infinity"
Done
```

QNaN Operand:

```
frD ← frB[0–34] || 0b0_0000_0000_0000_0000_0000_0000_0000
FPSCR[FPRF] ← "QNaN"
FPSCR[FR FI] ← 0b00
Done
```

SNaN Operand

```
FPSCR[VXSNAN] ← 1
If FPSCR[VE] = 0 then
    Do
      frD[0–11] ← frB[0–11]
      frD[12] ← 1
      frD[13–63] ← frB[13–34] || 0b0_0000_0000_0000_0000_0000_0000_0000
      FPSCR[FPRF] ← "QNaN"
    End
FPSCR[FR FI] ← 0b00
Done
```

Normal Operand

```
sign ← frB[0]
exp ← frB[1–11] – 1023
frac[0–52] ← 0b1 || frB[12–63]
Round single(sign,exp,frac[0–52],0,0,0)
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
If exp > +127 and FPSCR[OE] = 0 then go to Disabled Exponent Overflow
If exp > +127 and FPSCR[OE] = 1 then go to Enabled Overflow
frD[0] ← sign
frD[1–11] ← exp + 1023
frD[12–63] ← frac[1–52]
If sign = 0 then FPSCR[FPRF] ← "+normal number"
If sign = 1 then FPSCR[FPRF] ← "–normal number"
Done
```

Round Single (sign,exp,frac[0–52],G,R,X)

```
inc ← 0
lsb ← frac[23]
gbit ← frac[24]
rbit ← frac[25]
xbit ← (frac[26–52] || G || R || X) ≠ 0
If FPSCR[RN] = 0b00 then
    Do
        If sign || lsb || gbit || rbit || xbit = 0bu11uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0bu011u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc ← 1
    End
If FPSCR[RN] = 0b10 then
    Do
        If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
    End
If FPSCR[RN] = 0b11 then
    Do
        If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
    End
frac[0–23] ← frac[0–23] + inc
If carry_out =1 then
    Do
        frac[0–23] ← 0b1 || frac[0–22]
        exp ← exp + 1
    End
frac[24–52] ← (29)0
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
Return
```

E.4.2 Floating-Point Convert to Integer Model

The following algorithm describes the operation of the floating-point convert to integer instructions. In this example, 'u' represents an undefined hexadecimal digit.

```
If Floating Convert to Integer Word
    Then Do
        Then round_mode ← FPSCR[RN]
        tgt_precision ← "32-bit integer"
    End
If Floating Convert to Integer Word with round toward Zero
    Then Do
        round_mode ← 0b01
        tgt_precision ← "32-bit integer"
    End
If Floating Convert to Integer Double Word
    Then Do
        round_mode ← FPSCR[RN]
        tgt_precision ← "64-bit integer"
    End
If Floating Convert to Integer Double Word with Round toward Zero
    Then Do
        round_mode ← 0b01
```

```
      tgt_precision ← “64-bit integer”
    End
sign ← frB[0]
If frB[1–11] = 2047 and frB[12–63] = 0 then goto Infinity Operand
If frB[1–11] = 2047 and frB[12] = 0 then goto SNaN Operand
If frB[1–11] = 2047 and frB[12] = 1 then goto QNaN Operand
If frB[1–11] > 1054 then goto Large Operand


If frB[1–11] > 0 then exp ← frB[1–11] – 1023 /* exp – bias */
If frB[1–11] = 0 then exp ← –1022
If frB[1–11] > 0 then frac[0–64]← 0b01 || frB[12–63] || (11)0 /*normal*/
If frB[1–11] = 0 then frac[0–64]← 0b00 || frB[12–63] || (11)0 /*denormal*/


gbit || rbit || xbit ← 0b000
Do i = 1,63 – exp                          /*do the loop 0 times if exp = 63*/
    frac[0–64] || gbit || rbit || xbit ← 0b0 || frac[0–64] || gbit || (rbit | xbit)
End
```

Round Integer (sign,frac[0–64],gbit,rbit,xbit,round_mode)

In this example, ‘u’ represents an undefined hexadecimal digit. Comparisons ignore u bits.

```
If sign = 1 then frac[0–64] ← ¬frac[0–64] + 1 /* needed leading 0 for $–2^{64}$ < frB < $–2^{63}$*/

If tgt_precision = “32-bit integer” and frac[0–64] > $+2^{31} – 1$
    then goto Large Operand
If tgt_precision = “64-bit integer” and frac[0–64] > $+2^{63} – 1$
    then goto Large Operand
If tgt_precision = “32-bit integer” and frac[0–64] < $–2^{31}$ then goto Large Operand

FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]

If tgt_precision = “64-bit integer” and frac[0–64] < $–2^{63}$ then goto Large Operand
If tgt_precision = “32-bit integer”
    then frD ← 0xxuuu_uuuu || frac[33–64]
If tgt_precision = “64-bit integer” then frD ← frac[1–64]
FPSCR[FPRF] ← undefined
Done
```

Round Integer(sign,frac[0–64],gbit,rbit,xbit,round_mode)

In this example, ‘u’ represents an undefined hexadecimal digit. Comparisons ignore u bits.

```
inc ← 0
If round_mode = 0b00 then
    Do
      If sign || frac[64] || gbit || rbit || xbit = 0bu11uu then inc ← 1
      If sign || frac[64] || gbit || rbit || xbit = 0bu011u then inc ← 1
      If sign || frac[64] || gbit || rbit || xbit = 0bu01u1 then inc ← 1
    End
If round_mode = 0b10 then
    Do
        If sign || frac[64] || gbit || rbit || xbit = 0b0u1uu then inc ←1
        If sign || frac[64] || gbit || rbit || xbit = 0b0uu1u then inc ← 1
        If sign || frac[64] || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
      End
If round_mode = 0b11 then
    Do
      If sign || frac[64] || gbit || rbit || xbit = 0b1u1uu then inc ← 1
      If sign || frac[64] || gbit || rbit || xbit = 0b1uu1u then inc ← 1
```

```
        If sign || frac[64] || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
    End
frac[0–64] ← frac[0–64] + inc
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
Return
```

Infinity Operand

```
FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE] = 0 then Do
    If tgt_precision = “32-bit integer” then
        Do
            If sign = 0 then frD ← 0xuuuu_uuuu_7FFF_FFFF
            If sign = 1 then frD ← 0xuuuu_uuuu_8000_0000
        End
    Else
        Do
            If sign = 0 then frD ← 0x7FFF_FFFF_FFFF_FFFF
            If sign = 1 then frD ← 0x8000_0000_0000_0000
        End
    FPSCR[FPRF] ← undefined
    End
Done
```

SNaN Operand

```
FPSCR[FR FI VXCVI VXSNAN] ← 0b0011
If FPSCR[VE] = 0 then
    Do
        If tgt_precision = “32-bit integer”
            then frD ← 0xuuuu_uuuu_8000_0000
        If tgt_precision = “64-bit integer”
            then frD ← 0x8000_0000_0000_0000
        FPSCR[FPRF] ← undefined
    End
Done
```

QNaN Operand

```
FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE] = 0 then
    Do
        If tgt_precision = “32-bit integer” then frD ← 0xuuuu_uuuu_8000_0000
        If tgt_precision = “64-bit integer” then frD ← 0x8000_0000_0000_0000
        FPSCR[FPRF] ← undefined
    End
Done
```

Large Operand

```
FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE] = 0 then Do
    If tgt_precision = “32-bit integer” then
        Do
            If sign = 0 then frD ← 0xuuuu_uuuu_7FFF_FFFF
            If sign = 1 then frD ← 0xuuuu_uuuu_8000_0000
        End
    Else
        Do
            If sign = 0 then frD ← 0x7FFF_FFFF_FFFF_FFFF
            If sign = 1 then frD ← 0x8000_0000_0000_0000
        End
    FPSCR[FPRF] ← undefined
    End
Done
```

E.4.3 Floating-Point Convert from Integer Model

The following describes the operation of floating-point convert from integer instructions.

```
sign ← frB[0]
exp ← 63
frac[0–63] ← frB

If frac[0–63] = 0 then go to Zero Operand

If sign = 1 then frac[0–63] ← ¬frac[0–63] + 1

Do while frac[0] = 0
    frac[0–63] ← frac[1–63] || '0'
    exp ← exp – 1
End
```

Round Float(sign,exp,frac[0–63],FPSCR[RN])

```
If sign = 1 then FPSCR[FPRF] ← “–normal number”
If sign = 0 then FPSCR[FPRF] ← “+normal number”
frD[0] ← sign
frD[1–11] ← exp + 1023
frD[12–63] ← frac[1–52]
Done
```

Zero Operand

```
FPSCR[FR FI] ← 0b00
FPSCR[FPRF] ← “+zero”
frD ← 0x0000_0000_0000_0000
Done
```

Round Float(sign,exp,frac[0–63],round_mode)

In this example ‘u’ represents an undefined hexadecimal digit. Comparisons ignore u bits.

```
inc ← 0
lsb ← frac[52]
gbit ← frac[53]
rbit ← frac[54]
xbit ← frac[55–63] > 0
If round_mode = 0b00 then
    Do

      If sign || lsb || gbit || rbit || xbit = 0bu11uu then inc ← 1
      If sign || lsb || gbit || rbit || xbit = 0bu011u then inc ← 1
      If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc ← 1
    End
If round_mode = 0b10 then
    Do
      If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1
      If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1
      If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
    End
If round_mode = 0b11 then
    Do
      If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
      If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
      If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
    End
```

```
frac[0–52] ← frac[0–52] + inc
If carry_out = 1 then exp ← exp + 1
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
Return
```

E.5 Floating-Point Selection

The following are examples of how the optional **fsel** instruction can be used to implement floating-point minimum and maximum functions, and certain simple forms of if-then-else constructions, without branching.

The examples show program fragments in an imaginary, C-like, high-level programming language, and the corresponding program fragment using **fsel** and other instructions. In the examples, floating-point variables *a*, *b*, *x*, *y*, and *z* are assumed to be in FPRs *fa*, *fb*, *fx*, *fy*, and *fz*. FPR *fs* is assumed to be available for scratch space.

Additional examples can be found in Section E.3, “Floating-Point Conversions.”

Note that care must be taken in using **fse**l if IEEE-754 compatibility is required, or if the values being tested can be NaNs or infinities; see Section E.5.4, “Notes.”

E.5.1 Comparison to Zero

This section provides examples in a program fragment code sequence for the comparison to zero case.

High-level language:	**Power ISA:**

if a ≥ 0.0 then x ← y
else x ← z
	fsel fx, fa, fy, fz (see Section E.5.4, “Notes” number 1)

if a > 0.0 then x ← y
else x ← z
	fneg fs, fa
	fsel fx, fs, fz, fy (see Section E.5.4, “Notes” numbers 1 and 2)

if a = 0.0 then x ← y
else x ← z
	fsel fx, fa, fy, fz
	fneg fs, fa
	fsel fx, fs, fx, fz (see Section E.5.4, “Notes” number 1)

E.5.2 Minimum and Maximum

This section provides program fragment code examples for minimum and maximum cases.

High-level language:	**Power ISA:**
x ← min(a, b)	**fsub** fs, fa, fb (see Section E.5.4, “Notes” numbers 3, 4, and 5) **fsel** fx, fs, fb, fa
x ← max(a, b)	**fsub** fs, fa, fb (see Section E.5.4, “Notes” numbers 3, 4, and 5) **fsel** fx, fs, fa, fb

E.5.3 Simple If-Then-Else Constructions

This section provides examples in a program fragment code sequence for simple if-then-else statements.

High-level language:	Power ISA:

if a ≥ b then x ← y
else x ← z
	fsub fs, fa, fb
	fsel fx, fs, fy, fz (see Section E.5.4, “Notes” numbers 4 and 5)

if a >b then x ← y
else x ← z
	fsub fs, fb, fa
	fsel fx, fs, fz, fy (see Section E.5.4, “Notes” numbers 3, 4, and 5)

if a = b then x ← y
else x ← z
	fsub fs, fa, fb
	fsel fx, fs, fy, fz
	fneg fs, fs
	fsel fx, fs, fx, fz (see Section E.5.4, “Notes” numbers 4 and 5)

E.5.4 Notes

The following notes apply to the examples found in Section E.5.1, “Comparison to Zero,” Section E.5.2, “Minimum and Maximum,” and Section E.5.3, “Simple If-Then-Else Constructions,” and to the corresponding cases using the other three arithmetic relations (<, ≤, and ≠). These notes should also be considered when any other use of **fsel** is contemplated.

In these notes the optimized program is the program shown and the unoptimized program (not shown) is the corresponding program that uses **fcmpu** and branch conditional instructions instead of **fsel**.

1. The unoptimized program affects FPSCR[VXSNAN] and so may cause the system error handler to be invoked if the corresponding exception is enabled; the optimized program does not affect this bit. This property of the optimized program is incompatible with the IEEE-754 standard.
2. The optimized program gives the incorrect result if ‘a’ is a NaN.
3. The optimized program gives the incorrect result if ‘a’ and/or ‘b’ is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).
4. The optimized program gives the incorrect result if ‘a’ and ‘b’ are infinities of the same sign. (Here it is assumed that invalid operation exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if invalid operation exceptions are enabled, because in that case the target register of the subtraction is unchanged.)
5. The optimized program affects FPSCR[OX,UX,XX,VXISI], and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled, while the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE-754 standard.

E.6 Floating-Point Load Instructions

There are two basic forms of load instruction—single-precision and double-precision. Because FPRs support only double-precision format, single-precision load floating-point instructions convert single-precision data to double-precision format. The conversion and loading steps follow:

Let WORD[0–31] be the floating point single-precision operand accessed from memory.

Normalized Operand

```
If WORD[1–8] > 0 and WORD[1–8] < 255
  frD[0–1] ← WORD[0–1]
  frD[2] ← ¬ WORD[1]
  frD[3] ← ¬ WORD[1]
  frD[4] ← ¬ WORD[1]
  frD[5–63] ← WORD[2–31] || (29)0
```

Denormalized Operand

```
If WORD[1–8] = 0 and WORD[9–31] ≠ 0
  sign ← WORD[0]
  exp ← –126
  frac[0–52] ← 0b0 || WORD[9–31] || (29)0
  normalize the operand
    Do while frac[0] = 0
      frac ← frac[1–52] || 0b0
      exp ← exp – 1
    End
  frD[0] ← sign
  frD[1–11] ← exp + 1023
  frD[12–63] ← frac[1–52]
```

Infinity / QNaN / SNaN / Zero

```
If WORD[1–8] = 255 or WORD[1–31] = 0
  frD[0–1] ← WORD[0–1]
  frD[2] ← WORD[1]
  frD[3] ← WORD[1]
  frD[4] ← WORD[1]
  frD[5–63] ← WORD[2–31] || (29)0
```

For double-precision floating-point load instructions, no conversion is required as the data from memory is copied directly into the FPRs.

Many floating-point load instructions have an update form in which **r**A is updated with the EA. For these forms, if operand **r**A ≠ 0, the effective address (EA) is placed into register **r**A and the memory element (word or double word) addressed by the EA is loaded into the FPR specified by operand **fr**D; if operand **r**A = 0, the instruction form is invalid.

E.7 Floating-Point Store Instructions

There are three basic forms of store instruction—single-precision, double-precision, and integer. The integer form is provided by the optional **stfiwx** instruction. Because the FPRs support only floating-point double format for floating-point data, single-precision store floating-point instructions convert double-precision data to single-precision format prior to storing the operands into memory. The conversion steps follow:

Let WORD[0–31] be the word written to in memory.

No Denormalization Required (includes Zero/Infinity/NaN)

```
if frS[1–11] > 896 or frS[1–63] = 0 then
  WORD[0–1] ← frS[0–1]
  WORD[2–31] ← frS[5–34]
```

Denormalization Required

```
if 874 ≤ frS[1–11] ≤ 896 then
         sign ← frS[0]
         exp ← frS[1–11] – 1023
         frac ← 0b1 || frS[12–63]
         Denormalize operand
                  Do while exp < –126
                           frac ← 0b0 || frac[0–62]
                           exp ← exp + 1
                  End
         WORD[0] ← sign
         WORD[1–8] ← 0x00
         WORD[9–31] ← frac[1–23]
else WORD ← undefined
```

Note that if the value to be stored by a single-precision store floating-point instruction is larger in magnitude than the maximum number representable in single format, the first case mentioned, “No Denormalization Required,” applies. The result stored in WORD is then a well-defined value but is not numerically equal to the value in the source register (that is, the result of a single-precision load floating-point from WORD does not compare equal to the contents of the original source register).

Note that the description of conversion steps presented here is only a model. The actual implementation may vary from this description but must produce results equivalent to what this model would produce.

Note that for double-precision store floating-point instructions and for the store floating-point as integer word instruction, no conversion is required as the data from the FPR is copied directly into memory.

Glossary

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this book. Some of the terms and definitions included in the glossary are reprinted from IEEE Standard 754-1985™, *IEEE Standard for Binary Floating-Point Arithmetic*, copyright ©1985 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE.

A

Architecture. A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible *implementations*.

Asynchronous interrupt. *interrupts* that are caused by events external to the processor's execution. In this document, the term *asynchronous interrupt* is used interchangeably with the word *interrupt*.

Atomic access. A bus access that attempts to be part of a read-write operation to the same address uninterrupted by any other access to that address (the term refers to transactions that are indivisible). The architecture implements *atomic accesses* through the **lwarx**/**stwcx.** instruction pair.

B

Biased exponent. An *exponent* whose range of values is shifted by a constant (bias). Typically a bias is provided to allow a range of positive values to express a range that includes both positive and negative values.

Big-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *most-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the *most-significant byte*. See *Little-endian*.

Boundedly undefined. A characteristic of certain operation results that are not rigidly prescribed by the Power ISA. Boundedly-undefined results for a given operation may vary among implementations and between execution attempts in the same implementation.

Although the architecture does not prescribe the exact behavior for when results are allowed to be *boundedly undefined*, the results of executing instructions in contexts where results are allowed to be *boundedly undefined* are constrained to ones that could have been achieved by executing an arbitrary sequence of defined instructions, in valid form, starting in the state the machine was in before attempting to execute the given instruction.

Branch prediction. The process of guessing whether a branch will be taken. Such predictions can be correct or incorrect; the term ‘predicted’ as it is used here does not imply that the prediction is correct (successful). The architecture defines a means for *static branch* prediction as part of the instruction encoding.

Branch resolution. The determination of whether a branch is taken or not taken. A branch is said to be resolved when the processor can determine which instruction path to take. If the branch is resolved as predicted, the instructions following the predicted branch that may have been speculatively executed can complete (see *Completion*). If the branch is not resolved as predicted, instructions on the mispredicted path, and any results of speculative execution, are purged from the pipeline and fetching continues from the nonpredicted path.

C

Cache. High-speed memory containing recently accessed data or instructions (subset of main memory).

Cache block. A small region of contiguous memory that is copied from memory into a *cache*. The size of a *cache block* may vary among processors; the maximum block size is one *page*. In Power ISA processors, *cache coherency* is maintained on a cache-block basis. Note that the term *cache block* is often used interchangeably with ‘cache line.’

Cache coherency. An attribute wherein an accurate and common view of memory is provided to all devices that share the same memory system. Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor’s cache.

Cache flush. An operation that removes from a cache any data from a specified address range. This operation ensures that any modified data within the specified address range is written back to main memory. This operation is generated typically by a Data Cache Block Flush (**dcbf**) instruction.

Caching-inhibited. A memory update policy in which the cache is bypassed and the load or store is performed to or from main memory.

Cast out. A *cache block* that must be written to memory when a cache miss causes a *cache block* to be replaced.

Changed bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the changed bit if any store is performed into the *page*. See also *Page access history bits* and *Referenced bit*.

Clean. An operation that causes a *cache block* to be written to memory, if modified, and then left in a valid, unmodified state in the cache.

Clear. To cause a bit or bit field to register a value of zero. See also *Set*.

Completion. Completion occurs when an instruction has finished executing, written back any results, and is removed from the completion queue (CQ). When an instruction completes, it is guaranteed that this instruction and all previous instructions can cause no interrupts.

Context synchronization. An operation that ensures that all instructions in execution complete past the point where they can produce an *interrupt*, that all instructions in execution complete in the context in which they began execution, and that all subsequent instructions are *fetched* and executed in the new context. Context synchronization may result from executing specific instructions (such as **isync** or **rfi**) or when certain events occur (such as an *interrupt*).

D

Denormalized number. A nonzero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

E

Effective address (EA). The 32-bit address specified for a load, store, or an instruction fetch. This address is then submitted to the MMU for translation to either a *physical memory* address or an I/O address.

Exception. A condition that, if enabled, generates an interrupt.

Execution synchronization. A mechanism by which all instructions in execution are architecturally complete before beginning execution (appearing to begin execution) of the next instruction. Similar to context synchronization but doesn't force the contents of the instruction buffers to be deleted and refetched.

Exponent. In the binary representation of a floating-point number, the exponent is the component that normally signifies the integer power to which the value two is raised in determining the value of the represented number. See also *Biased exponent*.

F

Fetch. Instruction retrieval from either the cache or main memory and placing them into the instruction queue.

Finish. Finishing occurs in the last cycle of execution. In this cycle, the CQ entry is updated to indicate that the instruction has finished executing.

Floating-point register (FPR). Any of the 32 registers in the floating-point register file. These registers provide the source operands and destination results for floating-point instructions. Load instructions move data from memory to FPRs and store instructions move data from FPRs to memory. The FPRs are 64 bits wide and store floating-point values in double-precision format.

Floating-point unit. The functional unit in a processor responsible for executing all floating-point instructions.

Flush. An operation that causes a cache block to be invalidated and the data, if modified, to be written to memory.

Fraction. In the binary representation of a floating-point number, the field of the *significand* that lies to the right of its implied binary point.

G

General-purpose register (GPR). Any of the 32 registers in the general-purpose register file. These registers provide the source operands and destination results for all integer data manipulation instructions. Integer load instructions move data from memory to GPRs and store instructions move data from GPRs to memory.

Guarded. The guarded attribute pertains to out-of-order execution. When a page is designated as guarded, instructions and data cannot be accessed out-of-order.

H

Harvard architecture. An architectural model featuring separate caches and other memory management resources for instructions and data.

I

IEEE Std 754™ (or **IEEE 754).** A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point numbers.

Illegal instructions. A class of instructions that are not implemented for a particular processor. These include instructions not defined by the architecture. In addition, for 32-bit implementations, instructions that are defined only for 64-bit implementations are considered to be illegal instructions. For 64-bit implementations instructions that are defined only for 32-bit implementations are considered to be illegal instructions.

Implementation. A particular processor that conforms to the architecture, but may differ from other architecture-compliant implementations for example in design, feature set, and implementation of *optional* features. The architecture has many different implementations.

Imprecise interrupt. A type of *synchronous interrupt* that is allowed not to adhere to the precise interrupt model (see *Precise interrupt*). The architecture allows only floating-point exceptions to be handled imprecisely.

Integer unit. The functional unit responsible for executing all integer instructions.

In order. An aspect of an operation that adheres to a sequential model. An operation is said to be performed in-order if, at the time that it is performed, it is known to be required by the sequential execution model. See *Out-of-order.*

Instruction latency. The total number of clock cycles necessary to execute an instruction and make ready the results of that instruction.

Interrupt. A condition encountered by the processor that requires special, supervisor-level processing.

Interrupt handler. A software routine that executes when an interrupt is taken. Normally, the interrupt handler corrects the condition that caused the interrupt, or performs some other meaningful task (that may include aborting the program that caused the interrupt).

K

Kill. An operation that causes a *cache block* to be invalidated without writing any modified data to memory.

L

Latency. The number of clock cycles necessary to execute an instruction and make ready the results of that execution for a subsequent instruction.

L2 cache. See *Secondary cache*.

Least-significant bit (lsb). The bit of least value in an address, register, field, data element, or instruction encoding.

Least-significant byte (LSB). The byte of least value in an address, register, data element, or instruction encoding.

Little-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *least-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the *most-significant byte*. See *Big-endian*.

M

Mantissa. The decimal part of logarithm.

Memory access ordering. The specific order in which the processor performs load and store memory accesses and the order in which those accesses complete.

Memory-mapped accesses. Accesses whose addresses use the page or block address translation mechanisms provided by the MMU and that occur externally with the bus protocol defined for memory.

Memory coherency. An aspect of caching in which it is ensured that an accurate view of memory is provided to all devices that share system memory.

Memory consistency. Refers to agreement of levels of memory with respect to a single processor and system memory (for example, on-chip cache, secondary cache, and system memory).

Memory management unit (MMU). The functional unit that is capable of translating an *effective* (logical) *address* to a physical address, providing protection mechanisms, and defining caching methods.

Most-significant bit (msb). The highest-order bit in an address, registers, data element, or instruction encoding.

Most-significant byte (MSB). The highest-order byte in an address, registers, data element, or instruction encoding.

N

NaN. An abbreviation for not a number; a symbolic entity encoded in floating-point format. There are two types of NaNs—signaling NaNs and quiet NaNs.

No-op. No-operation. A single-cycle operation that does not affect registers or generate bus activity.

Normalization. A process by which a floating-point value is manipulated such that it can be represented in the format for the appropriate precision (single- or double-precision). For a floating-point value to be representable in the single- or double-precision format, the leading implied bit must be a 1.

O

OEA (operating environment architecture). The level of the architecture that describes the memory management model, supervisor-level registers, synchronization requirements, and the interrupt model. It also defines the time-base feature from a supervisor-level perspective. Implementations that conform to the OEA also conform to the UISA and VEA.

Optional. A feature, such as an instruction, a register, or an interrupt, that is defined by the architecture but not required to be implemented.

Out-of-order. An aspect of an operation that allows it to be performed ahead of one that may have preceded it in the sequential model, for example, speculative operations. An operation is said to be performed out-of-order if, at the time that it is performed, it is not known to be required by the sequential execution model. See *In-order.*

Out-of-order execution. A technique that allows instructions to be issued and completed in an order that differs from their sequence in the instruction stream.

Overflow. An condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, if two 32-bit numbers are multiplied, the result may not be representable in 32 bits. Since 32-bit registers cannot represent this sum, an overflow condition occurs.

P

Page. A region in memory. The OEA defines a page as a 4-Kbyte area of memory, aligned on a 4-Kbyte boundary.

Page fault. A page fault is a condition that occurs when the processor attempts to access a memory location that does not reside within a *page* not currently resident in *physical memory.* On Power ISA processors, a page fault interrupt condition occurs when a matching, valid *page table entry* (PTE[V] = 1) cannot be located.

Physical memory. The actual memory that can be accessed through the system's memory bus.

Pipelining. A technique that breaks operations, such as instruction processing or bus transactions, into smaller distinct stages or tenures (respectively) so that a subsequent operation can begin before the previous one has completed.

Precise interrupts. A category of interrupt for which the pipeline can be stopped so instructions that preceded the faulting instruction can complete and subsequent instructions can be flushed and redispatched after interrupt handling has completed. See *Imprecise interrupts*.

Primary opcode. The most-significant 6 bits (bits 0–5) of the instruction encoding that identifies the type of instruction.

Program order. The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the cache.

Protection boundary. A boundary between *protection domains*.

Q

Quiet NaN. A type of *NaN* that can propagate through most arithmetic operations without signaling interrupts. A quiet NaN is used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when invalid. See *Signaling NaN*.

R

Record bit. Bit 31 (or the Rc bit) in the instruction encoding. When it is set, updates the condition register (CR) to reflect the result of the operation.

Referenced bit. One of two *page history bits* found in each *page table entry*. The processor sets the *referenced bit* whenever the page is accessed for a read or write. See also *Page access history bits*.

Register indirect addressing. A form of addressing that specifies one GPR that contains the address for the load or store.

Register indirect with immediate index addressing. A form of addressing that specifies an immediate value to be added to the contents of a specified GPR to form the target address for the load or store.

Register indirect with index addressing. A form of addressing that specifies that the contents of two GPRs be added together to yield the target address for the load or store.

Rename register. Temporary buffers used by instructions that have finished execution but have not completed.

Reservation. The processor establishes a reservation on a *cache block* of memory space when it executes an **lwarx** instruction to read a memory semaphore into a GPR.

Reservation station. A buffer between the dispatch and execute stages that allows instructions to be dispatched even though the results of instructions on which the dispatched instruction may depend are not available.

RISC (reduced instruction set computing). An *architecture* characterized by fixed-length instructions with nonoverlapping functionality and by a separate set of load and store instructions that perform memory accesses.

S

Secondary cache. A cache memory that is typically larger and has a longer access time than the primary cache. A secondary cache may be shared by multiple devices. Also referred to as L2, or level-2, cache.

Set (*v*). To write a nonzero value to a bit or bit field; the opposite of *clear.* The term 'set' may also be used to generally describe the updating of a bit or bit field.

Set (*n*). A subdivision of a *cache*. Cacheable data can be stored in a given location in one of the sets, typically corresponding to its lower-order address bits. Because several memory locations can map to the same location, cached data is typically placed in the set whose *cache block* corresponding to that address was used least recently. See *Set-associative*.

Set-associative. Aspect of cache organization in which the cache space is divided into sections, called *sets*. The cache controller associates a particular main memory address with the contents of a particular set, or region, within the cache.

Signaling NaN. A type of *NaN* that generates an invalid operation program interrupt when it is specified as arithmetic operands. See *Quiet NaN*.

Significand. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

Simplified mnemonics. Assembler mnemonics that represent a more complex form of a common operation.

Snooping. Monitoring addresses driven by a bus master to detect the need for coherency actions.

Split-transaction. A transaction with independent request and response tenures.

Stall. An occurrence when an instruction cannot proceed to the next stage.

Static branch prediction. Mechanism by which software (for example, compilers) can hint to the machine hardware about the direction a branch is likely to take.

Superscalar. A superscalar processor is one that can dispatch multiple instructions concurrently from a conventional linear instruction stream. In a superscalar implementation, multiple instructions can be in the same stage at the same time.

Supervisor mode. The privileged operation state of a processor. In supervisor mode, software, typically the operating system, can access all control registers and can access the supervisor memory space, among other privileged operations.

Synchronization. A process to ensure that operations occur strictly *in order*. See *Context synchronization* and *Execution synchronization*.

Synchronous interrupt. An *interrupt* that is generated by the execution of a particular instruction or instruction sequence. There are two types of synchronous interrupts, *precise* and *imprecise*.

System memory. The physical memory available to a processor.

T

TLB (translation lookaside buffer). A cache that holds recently-used *page table entries*.

Throughput. The measure of the number of instructions that are processed per clock cycle.

U

UISA (user instruction set architecture). The level of the architecture to which user-level software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, floating-point memory conventions and interrupt model as seen by user programs, and the memory and programming models.

Underflow. A condition that occurs during arithmetic operations when the result cannot be represented accurately in the destination register. For example, underflow can happen if two floating-point fractions are multiplied and the result requires a smaller *exponent* and/or *mantissa* than the single-precision format can provide. In other words, the result is too small to be represented accurately.

User mode. The operating state of a processor used typically by application software. In user mode, software can access only certain control registers and can access only user memory space. No privileged operations can be performed. Also referred to as problem state.

V

VEA (virtual environment architecture). The level of the *architecture* that describes the memory model for an environment in which multiple devices can access memory, defines aspects of the cache model, defines cache control instructions, and defines the time-base facility from a user-level perspective. *Implementations* that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

Virtual address. An intermediate address used in the translation of an *effective address* to a physical address.

Virtual memory. The address space created using the memory management facilities of the processor. Program access to *virtual memory* is possible only when it coincides with *physical memory.*

W

Way. A location in the cache that holds a cache block, its tags and status bits.

Word. A 32-bit data element.

Write-back. A cache memory update policy in which processor write cycles are directly written only to the cache. External memory is updated only indirectly, for example, when a modified cache block is *cast out* to make room for newer data.

Write-through. A cache memory update policy in which all processor write cycles are written to both the cache and memory.