



A2O Processor

User's Manual

Preliminary

September 15, 2020
Version 0.666



© Copyright International Business Machines Corporation 2010, 2020

Printed in the United States of America

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.

Note: This document contains information on products in the sampling and/or initial production phases of development. This information is subject to change without notice. Verify with your IBM field applications engineer that you have the latest version of this document before finalizing a design.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN “AS IS” BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

IBM Systems and Technology Group
2070 Route 52, Bldg. 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at ibm.com®.
The IBM semiconductor solutions home page can be found at ibm.com/chips.

Version 0.666
September 15, 2020

Contents

List of Figures	21
List of Tables	23
Revision Log	29
About This Book	31
Who Should Use This Book	31
..... • How to Use This Book	31
..... • Notation	32
..... • Related Publications	33
List of Acronyms and Abbreviations	35
1. Overview	45
1.1 A2o Core Key Design Fundamentals	45
1.2 A20 core Features	45
1.3 A20 Core Organization	49
1.3.1 Instruction Unit	49
1.3.2 Execution Unit	50
1.3.3 Instruction and Data Caches	50
1.3.3.1 Instruction Cache	50
1.3.3.2 Data Cache	50
1.3.4 Memory Management Unit (MMU)	51
1.3.5 Timers	52
1.3.6 Debug Facilities	53
1.3.6.1 Debug Modes	53
1.3.6.2 Development Tool Support	53
1.3.7 Floating-Point Unit Organization	53
1.3.7.1 Arithmetic and Load/Store Pipelines	54
1.3.8 IEEE 754 and Architectural Compliance	54
1.3.8.1 IEEE 754 Compliance	54
1.3.9 Floating-Point Unit Implementation	55
1.3.9.1 Reciprocal Estimates	55
1.3.9.2 Denormalized B Operands	55
1.3.9.3 Non-IEEE mode	55
1.3.10 Floating-Point Unit Interfaces	55
1.3.10.1 A2 Processor Core Interface	55
1.3.10.2 Clock and Power Management Interface	55
1.4 Core Interfaces	56
1.4.1 System Interface	56
1.4.2 Auxiliary Execution Unit (AXU) Port	56
1.4.3 JTAG Port	57

2. CPU Programming Model	59
2.1 Logical Partitioning	59
2.1.1 Overview	59
2.2 Storage Addressing	60
2.2.1 Storage Operands	60
2.2.2 Effective Address Calculation	62
2.2.2.1 Data Storage Addressing Modes	62
2.2.2.2 Instruction Storage Addressing Modes	63
2.2.3 Byte Ordering	64
2.2.3.1 Structure Mapping Examples	64
2.2.3.2 Instruction Byte Ordering	65
2.2.3.3 Data Byte Ordering	66
2.2.3.4 Byte-Reverse Instructions	67
2.3 Multithreading	68
2.3.1 Thread Identification	68
2.3.1.1 Thread Identification Register (TIR)	68
2.3.1.2 Processor Identification Register (PIR)	68
2.3.1.3 Guest Processor Identification Register (GPIR)	69
2.3.2 Thread Run State	69
2.3.2.1 Thread Stop I/O Pin	69
2.3.2.2 Thread Control and Status Register (THRCTL)	69
2.3.2.3 Core Configuration Register 0 (CCRO)	69
2.3.2.4 Thread Enable Register (TENS, TENC)	70
2.3.2.5 Thread Enable Status Register (TENSr)	71
2.3.3 Wake On Interrupt	72
2.3.3.1 Core Configuration Register 1 (CCR1)	72
2.3.4 Thread Priority	73
2.3.4.1 Program Priority Register (PPR32)	73
2.3.4.2 Instruction Unit Configuration Register 1 (IUCR1)	74
2.3.5 Resources Shared between Threads	75
2.3.6 Shared Resources	75
2.3.6.1 Accessing Shared Resources	75
2.3.7 Duplicated Resources	76
2.3.8 Pipeline Sharing	77
2.3.8.1 Instruction Cache	78
2.3.8.2 Instruction Buffer and Decode Dependency	78
2.3.8.3 Instruction Dispatch	78
2.3.8.4 Instruction Issue	78
2.3.8.5 Ram Unit	78
2.3.8.6 Microcode Unit	78
2.3.8.7 Execution Units	78
2.4 Registers	79
2.4.1 Register Mapping	80
2.4.2 Register Types	81
2.4.2.1 General Purpose Registers	81
2.4.2.2 Special Purpose Registers	81
2.4.2.3 Condition Register	81
2.4.2.4 Machine State Register	82
2.5 32-Bit Mode	82
2.5.1 64-Bit Specific Instructions	82

2.5.2 32-Bit Instruction Selection	82
2.6 Instruction Categories	82
2.7 Instruction Classes	84
2.7.1 Defined Instruction Class	84
2.7.2 Illegal Instruction Class	85
2.7.3 Reserved Instruction Class	85
2.8 Implemented Instruction Set Summary	85
2.8.1 Integer Instructions	86
2.8.1.1 Integer Storage Access Instructions	86
2.8.1.2 Integer Arithmetic Instructions	88
2.8.1.3 Integer Logical Instructions	89
2.8.1.4 Integer Compare Instructions	89
2.8.1.5 Integer Trap Instructions	89
2.8.1.6 Integer Rotate Instructions	90
2.8.1.7 Integer Shift Instructions	90
2.8.1.8 Integer Population Count Instructions	90
2.8.1.9 Integer Select Instruction	90
2.8.1.10 Binary Coded Decimal Assist Instructions	91
2.8.2 Branch Instructions	91
2.8.3 Processor Control Instructions	91
2.8.3.1 Condition Register Logical Instructions	91
2.8.3.2 Register Management Instructions	92
2.8.3.3 System Linkage Instructions	92
2.8.3.4 Processor Control Instructions	92
2.8.4 Storage Control Instructions	93
2.8.4.1 Cache Management Instructions	93
2.8.4.2 Storage Visibility Instructions	93
2.8.4.3 TLB Management Instructions	94
2.8.4.4 Processor Synchronization Instruction	94
2.8.4.5 Load and Reserve and Store Conditional Instructions	94
2.8.4.6 Storage Synchronization Instructions	94
2.8.4.7 Wait Instruction	95
2.8.5 Initiate Coprocessor Instructions	95
2.8.5.1 Cache Initialization Instructions	95
2.8.5.2 Debug Instructions	96
2.9 Branch Processing	96
2.9.1 Branch Addressing	96
2.9.2 Branch Instruction BI Field	97
2.9.3 Branch Instruction BO Field	97
2.9.4 Branch Prediction	98
2.9.4.1 Branch Decoder	98
2.9.4.2 Branch Direction Prediction	99
2.9.4.3 Software Prediction	100
2.9.4.4 Dynamic Hardware Prediction	100
2.9.4.5 Performance Model Dynamic Predictor	101
2.9.4.6 During BHT update, the bimodal table is always updated with the last direction. Only the gshare counter used during prediction is updated. The other gshare counter is not updated. Bimodal Branch History	101
2.9.4.7 Bimodal Branch History	101
2.9.5 Branch Control Registers	102

2.9.5.1 Link Register (LR)	102
2.9.5.2 Count Register (CTR)	102
2.9.5.3 Condition Register (CR)	103
2.9.5.4 Target Address Register	106
2.10 Integer Processing	106
2.10.1 General Purpose Registers (GPRs)	106
2.10.2 Integer Exception Register (XER)	107
2.10.2.1 Summary Overflow (SO) Field	108
2.10.2.2 Overflow (OV) Field	108
2.10.2.3 Carry (CA) Field	108
2.10.2.4 Transfer Byte Count (TBC) Field	109
2.11 Processor Control	109
2.11.1 Special Purpose Registers General (SPRG0–SPRG8)	110
2.11.2 External Process ID Load Context (EPLC) Register	115
2.11.3 External Process ID Store Context (EPSC) Register	115
2.12 Privileged Modes	116
2.12.1 Privileged Instructions	117
2.12.1.1 Cache Locking Instructions	117
2.12.2 Privileged SPRs	118
2.13 Speculative Accesses	118
2.14 Synchronization	118
2.14.1 Context Synchronization	118
2.14.2 Execution Synchronization	120
2.14.3 Storage Ordering and Synchronization	120
2.15 Software Transactional Memory Acceleration	121
2.15.1 Summary	121
2.15.2 Implementation	121
2.15.2.1 L1 D-Cache	122
2.15.3 Watch Operation Ordering Requirements	122
2.15.4 Impact on Existing Software	122
3. FU Programming Model	123
3.1 Storage Addressing	123
3.1.1 Storage Operands	123
3.1.2 Effective Address Calculation	124
3.1.3 Data Storage Addressing Modes	124
3.2 Floating-Point Exceptions	125
3.3 Floating-Point Registers	125
3.3.1 Register Types	126
3.3.1.1 Floating-Point Registers (FPR0–FPR31)	126
3.3.1.2 Floating-Point Status and Control Register (FPSCR)	127
3.4 Floating-Point Data Formats	129
3.4.1 Value Representation	130
3.4.2 Binary Floating-Point Numbers	131
3.4.2.1 Normalized Numbers	131
3.4.2.2 Denormalized Numbers	132
3.4.2.3 Zero Values	132
3.4.3 Infinities	132
3.4.3.1 Not a Numbers	132

3.4.4 Sign of Result	133
3.4.5 Normalization and Denormalization	134
3.4.6 Data Handling and Precision	134
3.4.7 Rounding	135
3.5 Floating-Point Execution Models	136
3.5.1 Execution Model for IEEE Operations	137
3.5.2 Execution Model for Multiply-Add Type Instructions	139
3.6 Floating-Point Instructions	139
3.6.1 Instructions by Category	140
3.6.2 Load and Store Instructions	141
3.6.3 Floating-Point Store Instructions	142
3.6.4 Floating-Point Move Instructions	144
3.6.5 Floating-Point Arithmetic Instructions	144
3.6.5.1 Floating-Point Multiply-Add Instructions	145
3.6.6 Floating-Point Rounding and Conversion Instructions	145
3.6.7 Floating-Point Compare Instructions	146
3.6.8 Floating-Point Status and Control Register Instructions	147
4. Initialization	149
4.1 Core Reset	149
4.2 A2 Core State After Reset	149
4.3 Core Reset Request and Status Signals	149
4.3.1 Core Reset Requests	149
4.3.1.1 From Debug	150
4.3.1.2 From Watchdog Timer	150
4.3.2 Reset Request Status	150
4.3.2.1 Debug Facility Reset Status	151
4.3.2.2 Timer Facility Reset Status	151
4.4 Initialization Software Requirements	152
5. Instruction and Data Caches	153
5.1 Data Cache Array Organization and Operation	153
5.2 Instruction Cache Array Organization and Operation	154
5.3 Cache Line Replacement Policy	154
5.4 Instruction Cache Controller	154
5.4.1 ICC Operations	155
5.4.2 Instruction Cache Coherency	155
5.4.2.1 Self-Modifying Code	156
5.4.2.2 Instruction Cache Synonyms	156
5.4.3 Instruction Cache Control and Debug	156
5.4.3.1 Instruction Cache Management and Debug Instruction Summary	156
5.4.3.2 Instruction Cache Parity Operations	157
5.4.3.3 Simulating Instruction Cache Parity Errors for Software Testing	157
5.5 Data Cache Controller	157
5.5.1 DCC Operations	158
5.5.1.1 Load and Store Alignment	159
5.5.1.2 Load Operations	159
5.5.1.3 Store Operations	160
5.5.1.4 Data Read and Instruction Fetch Interface Requests	160

5.5.1.5 Data Write Interface Requests	160
5.5.1.6 Storage Access Ordering	161
5.5.2 Data Cache Coherency	161
5.5.3 Data Cache Control	161
5.5.3.1 Data Cache Management Instruction Summary	161
5.5.3.2 dcbt and dcbtst Operation	162
5.5.3.3 Cache Locking Mechanisms	163
5.5.3.4 Data Cache Parity Operations	167
5.5.3.5 Simulating Data Cache Parity Errors for Software Testing	167
5.5.3.6 Data Cache Disable	167
6. Memory Management	169
6.1 MMU Overview	169
6.1.1 Support for Power ISA MMU Architecture	170
6.2 Page Identification	170
6.2.1 Virtual Address Formation	171
6.2.2 Address Space Identifier Convention	171
6.2.3 Exclusion Range (X-bit) Operation	172
6.2.4 TLB Match Process	173
6.3 Address Translation	175
6.4 Access Control	177
6.4.1 Execute Access	177
6.4.2 Write Access	177
6.4.3 Read Access	178
6.4.4 Access Control Applied to Cache Management Instructions	178
6.5 Storage Attributes	179
6.5.1 Write-Through (W)	180
6.5.2 Caching Inhibited (I)	180
6.5.3 Memory Coherence Required (M)	180
6.5.4 Guarded (G)	180
6.5.5 Endian (E)	181
6.5.6 User-Definable (U0–U3)	181
6.5.7 Supported Storage Attribute Combinations	181
6.5.8 Aliasing	181
6.6 Translation Lookaside Buffer	182
6.7 Effective to Real Address Translation Arrays	187
6.7.1 ERAT Context Synchronization	188
6.7.2 ERAT Reset Behavior	189
6.7.3 Atomic Update of ERAT Entries	189
6.7.4 ERAT LRU Round-Robin Replacement Mode	189
6.7.5 ERAT LRU Replacement Watermark	190
6.7.6 ERAT (TLB Lookaside Information) Coherency and Back-Invalidation	190
6.7.7 ERAT External PID (EPID) Context and Instruction Dependencies	192
6.8 Logical to Real Address Translation Array (Category E.HV.LRAT)	193
6.9 TLB Management Instructions (Architected)	196
6.9.1 TLB Read and Write Instructions (tlbre and tlbwe)	197
6.9.2 TLB Search Instruction (tlbsx[.])	199
6.9.3 TLB Search and Reserve Instruction (tlbsrx.)	199
6.9.4 TLB Invalidate Virtual Address (Indexed) Instruction (tlbivax)	200

6.9.5 TLB Invalidate Local (Indexed) Instruction (tlbilx)	202
6.9.6 TLB Sync Instruction (tlbsync)	202
6.10 ERAT Management Instructions (Non-Architected)	203
6.10.1 ERAT Read and Write Instructions (eratre and eratwe)	203
6.10.2 ERAT Search Instruction (eratsx[.])	204
6.10.3 ERAT Invalidate Virtual Address (Indexed) Instruction (erativax)	205
6.10.4 ERAT Invalidate Local (Indexed) Instruction (eratilx)	208
6.11 32-Bit Mode Memory Management Behavior	208
6.11.1 32-Bit Mode TLB Read and Write Instructions (tlbre and tlbwe)	209
6.11.2 32-Bit Mode TLB Search Instruction (tlbsx[.])	209
6.11.3 32-Bit Mode TLB Search and Reserve Instruction (tlbsrx.)	209
6.11.4 32-Bit Mode TLB Invalidate Virtual Address (Indexed) Instruction (tlbivax)	210
6.11.5 32-Bit Mode TLB Invalidate Local (Indexed) Instruction (tlbilx)	210
6.11.6 32-Bit Mode TLB Sync Instruction (tlbsync)	210
6.11.7 32-Bit Mode ERAT Read and Write Instructions (eratre and eratwe)	210
6.11.8 32-Bit Mode ERAT Search Instruction (eratsx[.])	211
6.11.9 32-Bit Mode ERAT Invalidate Virtual Address (Indexed) Instruction (erativax)	211
6.11.10 32-Bit Mode ERAT Invalidate Local (Indexed) Instruction (eratilx)	212
6.12 Page Reference and Change Status Management	212
6.13 TLB and ERAT Parity Operations	213
6.13.1 Parity Errors Generated from tlbre or eratre	214
6.13.2 Simulating TLB and ERAT Parity Errors for Software Testing	215
6.14 ERAT-Only Mode Operation	216
6.15 TLB Reservations and TLB Write Conditional (Category E.TWC)	216
6.16 Hardware Page Table Walking (Category E.PT)	221
6.16.1 Searching the TLB for Direct and Indirect Entries	221
6.16.2 Indirect TLB Entry Page and Sub-Page Sizes	222
6.16.3 Hardware Page Table Entry Format	223
6.16.4 Calculation of Hardware Page Table Entry Real Address	224
6.16.5 Hardware Page Table Errors and Exceptions	225
6.16.6 Hardware Page Table Storage Control Attributes	225
6.16.7 TLB Update After Hardware Page Table Translation	226
6.17 Storage Control Registers (Architected)	228
6.17.1 Process ID Register (PID)	228
6.17.2 Logical Partition ID Register (LPIDR)	229
6.17.3 External PID Load Context (EPLC) Register	230
6.17.4 External PID Store Context (EPSC) Register	231
6.17.5 MMU Assist Register 0 (MAS0)	232
6.17.6 MMU Assist Register 1 (MAS1)	233
6.17.7 MMU Assist Register 2 (MAS2)	234
6.17.8 MMU Assist Register 2 Upper (MAS2U)	236
6.17.9 MMU Assist Register 3 (MAS3)	237
6.17.10 MMU Assist Register 4 (MAS4)	239
6.17.11 MMU Assist Register 5 (MAS5)	240
6.17.12 MMU Assist Register 6 (MAS6)	241
6.17.13 MMU Assist Register 7 (MAS7)	242
6.17.14 MMU Assist Register 8 (MAS8)	243
6.17.15 MAS0_MAS1 Register	244
6.17.16 MAS5_MAS6 Register	245
6.17.17 MAS7_MAS3 Register	246

6.17.18 MAS8_MAS1 Register	247
6.17.19 MMU Configuration Register (MMUCFG)	248
6.17.20 MMU Control and Status Register 0 (MMUCSR0)	249
6.17.21 TLB 0 Configuration Register (TLBOCFG)	250
6.17.22 TLB 0 Page Size Register (TLBOPS)	252
6.17.23 LRAT Configuration Register (LRATCFG)	253
6.17.24 LRAT Page Size Register (LRATPS)	254
6.17.25 Embedded Page Table Configuration Register (EPTCFG)	256
6.17.26 Logical Page Exception Register (LPER)	257
6.17.27 Logical Page Exception Register Upper (LPERU)	258
6.17.28 MAS Register Update Summary	259
6.18 Storage Control Registers (Non-Architected)	261
6.18.1 Memory Management Unit Control Register 0 (MMUCRO)	261
6.18.2 Memory Management Unit Control Register 1 (MMUCR1)	264
6.18.3 Memory Management Unit Control Register 2 (MMUCR2)	271
6.18.4 Memory Management Unit Control Register 3 (MMUCR3)	274
7. CPU Interrupts and Exceptions	277
7.1 Overview	277
7.2 Directed Interrupts	277
7.3 Interrupt Classes	278
7.3.1 Asynchronous Interrupts	278
7.3.2 Synchronous Interrupts	278
7.3.2.1 Synchronous, Precise Interrupts	278
7.3.2.2 Synchronous, Imprecise Interrupts	279
7.3.3 Critical and Noncritical Interrupts	280
7.3.4 Machine Check Interrupts	280
7.4 Interrupt Processing	281
7.4.1 Partially Executed Instructions	283
7.5 Interrupt Processing Registers	284
7.5.1 Register Mapping	285
7.5.2 Machine State Register (MSR)	285
7.5.3 Machine State Register Protect (MSRP)	287
7.5.4 Embedded Processor Control Register (EPCR)	288
7.5.5 Save/Restore Register 0 (SRR0)	289
7.5.6 Save/Restore Register 1 (SRR1)	290
7.5.7 Guest Save/Restore Register 0 (GSRR0)	292
7.5.8 Guest Save/Restore Register 1 (GSRR1)	292
7.5.9 Critical Save/Restore Register 0 (CSRR0)	294
7.5.10 Critical Save/Restore Register 1 (CSRR1)	295
7.5.11 Machine Check Save/Restore Register 0 (MCSRR0)	297
7.5.12 Machine Check Save/Restore Register 1 (MCSRR1)	297
7.5.13 Data Exception Address Register (DEAR)	299
7.5.14 Guest Data Exception Address Register (GDEAR)	299
7.5.15 Interrupt Vector Prefix Register (IVPR)	301
7.5.16 Guest Interrupt Vector Prefix Register (GIVPR)	301
7.5.17 Exception Syndrome Register (ESR)	302
7.5.18 Guest Exception Syndrome Register (GESR)	303
7.5.19 Machine Check Status Register (MCSR)	305

7.6 Interrupt Definitions	306
7.6.1 Critical Input Interrupt	309
7.6.2 Machine Check Interrupt	310
7.6.2.1 Machine Check Status Register (MCSR)	312
7.6.3 Data Storage Interrupt	313
7.6.4 Instruction Storage Interrupt	317
7.6.5 External Input Interrupt	319
7.6.6 Alignment Interrupt	320
7.6.7 Program Interrupt	321
7.6.8 Floating-Point Unavailable Interrupt	325
7.6.9 System Call Interrupt	325
7.6.10 Auxiliary Processor Unavailable Interrupt	326
7.6.11 Decrementer Interrupt	326
7.6.12 Guest Decrementer Interrupt	326
7.6.13 Fixed-Interval Timer Interrupt	327
7.6.14 Guest Fixed-Interval Timer Interrupt	327
7.6.15 Watchdog Timer Interrupt	328
7.6.16 Guest Watchdog Timer Interrupt	328
7.6.17 Data TLB Error Interrupt	329
7.6.18 Instruction TLB Error Interrupt	330
7.6.19 Vector Unavailable Interrupt	331
7.6.20 Debug Interrupt	331
7.6.21 Processor Doorbell Interrupt	335
7.6.22 Processor Doorbell Critical Interrupt	336
7.6.23 Guest Processor Doorbell Interrupt	336
7.6.24 Guest Processor Doorbell Critical Interrupt	337
7.6.25 Guest Processor Doorbell Machine Check Interrupt	337
7.6.26 Embedded Hypervisor System Call Interrupt	338
7.6.27 Embedded Hypervisor Privilege Interrupt	338
7.6.28 LRAT Error Interrupt	339
7.6.29 User Decrementer Interrupt	340
7.6.30 Performance Monitor Interrupt	340
7.7 Processor Messages	341
7.7.1 Processor Message Handling and Filtering	341
7.7.2 Doorbell Message Filtering	342
7.7.3 Doorbell Critical Message Filtering	343
7.7.4 Guest Doorbell Message Filtering	344
7.7.5 Guest Doorbell Critical Message Filtering	344
7.7.6 Guest Doorbell Machine Check Message Filtering	345
7.8 Interrupt Ordering and Masking	346
7.8.1 Interrupt Ordering Software Requirements	347
7.8.2 Interrupt Order	348
7.9 Exception Priorities	349
7.9.1 Exception Priorities for Integer Load, Store, and Cache Management Instructions	350
7.9.2 Exception Priorities for Floating-Point Load and Store Instructions	351
7.9.3 Exception Priorities for Floating-Point Instructions (Other)	351
7.9.4 Exception Priorities for Privileged Instructions	352
7.9.5 Exception Priorities for Trap Instructions	352
7.9.6 Exception Priorities for System Call Instruction	352
7.9.7 Exception Priorities for Branch Instructions	353

7.9.8 Exception Priorities for Return From Interrupt Instructions	353
7.9.9 Exception Priorities for Reserved Instructions	353
7.9.10 Exception Priorities for All Other Instructions	354
8. FU Interrupts and Exceptions	355
8.1 Floating-Point Exceptions	355
8.2 Exceptions List	356
8.3 Floating-Point Interrupts	359
8.3.1 Floating-Point Unavailable Interrupt	359
8.3.2 Floating-Point Assist Interrupt	359
8.4 Floating-Point Exception Behavior	359
8.4.1 Invalid Operation Exception	359
8.4.1.1 Action	360
8.4.2 Zero Divide Exception	361
8.4.2.1 Action	361
8.4.3 Overflow Exception	362
8.4.3.1 Action	362
8.4.4 Underflow Exception	363
8.4.4.1 Action	363
8.4.5 Inexact Exception	364
8.4.5.1 Action	364
8.5 Exception Priorities for Floating-Point Load and Store Instructions	364
8.6 Exception Priorities for Other Floating-Point Instructions	365
8.7 QNaN	365
8.8 Updating FPRs on Exceptions	366
8.9 Floating-Point Status and Control Register (FPSCR)	366
8.10 Updating the Condition Register	369
8.10.1 Condition Register (CR)	369
8.10.2 Updating CR Fields	370
8.10.3 Generation of QNaN Results	370
9. Timer Facilities	371
9.1 Time Base	373
9.1.1 Reading the Time Base	374
9.1.2 Writing the Time Base	374
9.2 Decrementer (DEC)	374
9.3 Guest Decrementer (GDEC)	376
9.4 User Decrementer (UDEC)	377
9.5 Fixed Interval Timer (FIT)	378
9.6 Guest Fixed Interval Timer (GFIT)	379
9.7 Watchdog Timer	379
9.8 Guest Watchdog Timer	381
9.9 Timer Control Register (TCR)	383
9.10 Guest Timer Control Register (GTCR)	385
9.11 Timer Status Register (TSR)	385
9.12 Guest Timer Status Register (GTSR)	386
9.13 Freezing the Timer Facilities	386
9.14 Selection of the Timer Clock Source	386

9.15 Selecting the Timer Clock Frequency Divide Value	386
9.16 Synchronizing Timers Across Multiple Cores	386
10. Debug Facilities	389
10.1 Implications of Hypervisor on Debug Controls	389
10.2 Support for Development Tools	389
10.3 Debug Modes	389
10.3.1 Internal Debug Mode	390
10.3.2 External Debug Mode	390
10.3.3 Trace Debug Mode	391
10.4 Debug Events	392
10.4.1 Instruction Address Compare (IAC) Debug Event	392
10.4.1.1 IAC Debug Event Fields	393
10.4.1.2 IAC Debug Event Processing	394
10.4.2 Data Address Compare (DAC) Debug Event	395
10.4.2.1 DAC Debug Event Fields	395
10.4.2.2 DAC Debug Event Processing	397
10.4.2.3 DAC Debug Events Applied to Instructions that Result in Multiple Storage Accesses	397
10.4.2.4 DAC Debug Events Applied to Various Instruction Types	398
10.4.3 Data Value Compare (DVC) Debug Event	399
10.4.3.1 DVC Debug Event Fields	399
10.4.3.2 DVC Debug Event Processing	400
10.4.3.3 DVC Debug Events Applied to Instructions that Result in Multiple Storage Accesses	400
10.4.3.4 DVC Debug Events Applied to Various Instruction Types	401
10.4.3.5 DVC Debug Events Applied to Floating-Point Loads and Stores	401
10.4.4 Instruction Complete (ICMP) Debug Event	401
10.4.5 Branch Taken (BRT) Debug Event	402
10.4.6 Trap (TRAP) Debug Event	402
10.4.7 Return (RET) Debug Event	402
10.4.8 Interrupt (IRPT) Debug Event	403
10.4.9 Unconditional Debug Event (UDE)	404
10.4.10 Instruction Value Compare (IVC) Debug Event	404
10.4.11 Debug Event Summary	405
10.5 Debug Reset	405
10.6 Debug Timer Freeze	405
10.7 Debug Registers	405
10.7.1 Debug Control Register 0 (DBCR0)	406
10.7.2 Debug Control Register 1 (DBCR1)	408
10.7.3 Debug Control Register 2 (DBCR2)	409
10.7.4 Debug Control Register 3 (DBCR3)	411
10.7.5 Debug Status Register (DBSR)	412
10.7.6 Debug Status Register Write Register (DBSRWR)	413
10.7.7 Instruction Address Compare Registers (IAC1–IAC4)	414
10.7.8 Data Address Compare Registers (DAC1–DAC2)	416
10.7.9 Data Value Compare Registers (DVC1–DVC2)	417
10.7.10 Instruction Address Register (IAR)	418
10.7.11 Instruction Match Mask Registers (IMMR)	419

10.7.12 Instruction Match Registers (IMR)	419
10.8 Debug configuration and control functions	419
10.8.1 Core debug modes	420
10.8.2 Additional debug functions	420
10.8.3 Recoverable error counter	421
10.8.4 PC Configuration Register 0 (PCCR0) Definition	421
10.9 Thread control and status	422
10.9.1 Thread control functions	422
10.9.1.1 Example procedure to perform instruction stepping	423
10.9.2 Thread status	423
10.9.3 Thread Control and Status Register (THRCTL) Definition	424
10.10 Instruction Stuffing	426
10.10.1 Ram registers	426
10.10.2 Ram instructions	426
10.10.2.1 Supported Ram instructions	427
10.10.2.2 Using scratch registers as temporary storage	427
10.10.3 Ram control bits	427
10.10.4 Ram status bits	428
10.10.5 Ram data	429
10.10.6 Ram Mode overview	430
10.10.6.1 Basic Ram process	430
10.10.6.2 Example Ram procedure	430
10.11 Direct Access to I-Cache and D-Cache Directories	432
10.11.1 General Read D-Cache Directory Sequence for L1 D-Cache	432
10.11.2 Instruction Unit Debug Register 0 (IUDBG0)	433
10.11.3 Instruction Unit Debug Register 1 (IUDBG1)	433
10.11.4 Instruction Unit Debug Register 2 (IUDBG2)	434
10.11.5 Execution Unit Debug Register 0 (XUDBG0)	434
10.11.6 Execution Unit Debug Register 1 (XUDBG1)	435
10.11.7 Execution Unit Debug Register 2 (XUDBG2)	435
10.12 Support for the Debugger Notify Halt (dnh) instruction	436
10.12.1 DNH Data Register (DNHDR) Definition	436
10.13 Trace and Trigger Bus	437
10.13.1 Trace and Trigger Bus Overview	437
10.13.2 Unit Level Trace and Trigger Bus Implementation	438
10.13.3 Debug Select Registers	439
11. Performance Events and Event Selection	441
11.1 Event Bus Overview	442
11.2 A2 Core Event Multiplexer and Performance Event Controls	442
11.2.1 Enabling performance event and trace bus latches	442
11.2.2 Performance analysis operating modes	442
11.2.3 Options for selecting debug bus signals	443
11.2.4 Speculative and non-speculative performance event selection	443
11.2.5 Core performance event selection to external event bus	443
11.2.6 Core Event Select Registers	444
11.3 Unit level performance event selection	446
11.3.1 Unit event multiplexer component	446
11.3.2 Performance Monitor Event Tags and Count Modes	448

11.3.3 Unit performance event tables	448
11.4 Unit performance event tables	449
11.4.1 AXU0 performance events table	449
11.4.2 IU performance events table	450
11.4.3 LQ performance events table	453
11.4.4 MMU performance events table	458
11.4.5 RV performance events table	460
11.4.6 XU performance events tables	461
11.5 Unit event select registers	463
11.5.1 AXU0 event select register	463
11.5.2 IU event select registers	464
11.5.3 LQ event select registers	466
11.5.4 MMU event select registers	467
11.5.5 RV event select registers	469
11.5.6 XU event select registers	471
11.6 A2 support for core instruction trace	472
11.7 A2 support for instruction sampling	472
11.7.1 Sampled Instruction Address Register (SIAR)	473
11.7.2 Instruction sampling control bits	473
11.7.2.1 Performance Monitor Alert Enable (PMAE)	473
11.7.2.2 Performance Monitor Alert Occurred (PMAO)	473
11.7.2.3 Combined PMAO and PMAE function	473
12. Implementation Dependent Instructions	475
12.1 Miscellaneous	475
12.1.1 Attention (attn)	475
12.2 TLB Management Instructions	476
12.2.1 TLB Read Entry (tlbre)	476
12.2.2 TLB Write Entry (tlbwe)	478
12.2.3 TLB Search Indexed (tlbsx[.])	480
12.2.4 TLB Search and Reserve Indexed (tlbsrx.)	482
12.2.5 TLB Invalidate Virtual Address Indexed (tlbivax)	484
12.2.6 TLB Invalidate Local Indexed (tlbilx)	487
12.3 ERAT Management Instructions	490
12.3.1 ERAT Read Entry (eratre)	490
12.3.2 ERAT Write Entry (eratwe)	493
12.3.3 ERAT Search Indexed (eratsx[.])	496
12.3.4 ERAT Invalidate Virtual Address Indexed (erativax)	498
12.3.5 ERAT Invalidate Local Indexed (eratilx)	501
12.4 Software Transactional Memory Instructions	503
12.4.1 Load Doubleword and Watch Indexed X-Form (ldawx.	504
12.4.2 Watch Check All X-Form (wchkall)	505
12.4.3 Watch Clear X-Form (wclr)	506
12.5 Coprocessor Instructions	507
12.5.1 Initiate Coprocessor Store Word Indexed (icswx[.])	509
12.5.1.1 General Registers	510
12.5.1.2 Initial Execution	511
12.5.2 Initiate Coprocessor Store Word External Process ID Indexed (icswepx[.])	512
12.5.3 Execution	512

12.5.3.1 Condition Register 0	513
12.5.4 Coprocessor-Request Block	513
12.5.4.1 Available Coprocessor Register (ACOP)	514
12.5.4.2 Hypervisor Available Coprocessor Register (HACOP)	515
13. Power Management Methods	517
13.1 Power-savings methods	517
13.2 Power-savings instructions	517
13.2.1 Power-saving instruction sequence	518
14. Register Summary	521
14.1 Register Categories	521
14.2 Reserved Fields	527
14.3 Unimplemented SPRs	527
14.4 Device Control Registers	527
14.5 Alphabetical Register Listing	529
14.5.1 A0ESR - AXU0 Event Select Register	530
14.5.2 ACOP - Available Coprocessor	532
14.5.3 CCR0 - Core Configuration Register 0	533
14.5.4 CCR1 - Core Configuration Register 1	534
14.5.5 CCR2 - Core Configuration Register 2	535
14.5.6 CCR3 - Core Configuration Register 3	537
14.5.7 CCR4 - Core Configuration Register 4	538
14.5.8 CESR1 - Core Event Select Register1	539
14.5.9 CESR2 - Core Event Select Register2	541
14.5.10 CPCRO - Completion Configuration Register 0	543
14.5.11 PCR1 - Completion Configuration Register 1	544
14.5.12 PCR2 - Completion Configuration Register 2	545
14.5.13 CR - Condition Register	546
14.5.14 CSRR0 - Critical Save/Restore Register 0	547
14.5.15 CSRR1 - Critical Save/Restore Register 1	548
14.5.16 CTR - Count Register	550
14.5.17 DAC1 - Data Address Compare 1	551
14.5.18 DAC2 - Data Address Compare 2	552
14.5.19 DAC3 - Data Address Compare 3	553
14.5.20 DAC4 - Data Address Compare 4	554
14.5.21 DBCR0 - Debug Control Register 0	555
14.5.22 DBCR1 - Debug Control Register 1	557
14.5.23 DBCR2 - Debug Control Register 2	559
14.5.24 DBCR3 - Debug Control Register 3	561
14.5.25 DBSR - Debug Status Register	562
14.5.26 DBSRWR - Debug Status Register Write Register	564
14.5.27 DEAR - Data Exception Address Register	566
14.5.28 DEC - Decrementer	567
14.5.29 DECAR - Decrementer Auto-Reload	568
14.5.30 DNHDR - Debug Notify Halt Data Register	569
14.5.31 DSCR - Data Stream Control Register	570
14.5.32 DVC1 - Data Value Compare 1	571
14.5.33 DVC2 - Data Value Compare 2	572

14.5.34 EPCR - Embedded Processor Control Register	573
14.5.35 EPLC - External Process ID Load Context	575
14.5.36 EPSC - External Process ID Store Context	576
14.5.37 EPTCFG - Embedded Page Table Configuration Register	577
14.5.38 ESR - Exception Syndrome Register	578
14.5.39 GDEAR - Guest Data Exception Address Register	580
14.5.40 GESR - Guest Exception Syndrome Register	581
14.5.41 GIVPR - Guest Interrupt Vector Prefix Register	583
14.5.42 GPIR - Guest Processor ID Register	584
14.5.43 GSPRG0 - Guest Software Special Purpose Register 0	585
14.5.44 GSPRG1 - Guest Software Special Purpose Register 1	586
14.5.45 GSPRG2 - Guest Software Special Purpose Register 2	587
14.5.46 GSPRG3 - Guest Software Special Purpose Register 3	588
14.5.47 GSRR0 - Guest Save/Restore Register 0	589
14.5.48 GSRR1 - Guest Save/Restore Register 1	590
14.5.49 HACOP - Hypervisor Available Coprocessor	592
14.5.50 IAC1 - Instruction Address Compare 1	593
14.5.51 IAC2 - Instruction Address Compare 2	594
14.5.52 IAC3 - Instruction Address Compare 3	595
14.5.53 IAC4 - Instruction Address Compare 4	596
14.5.54 IAR - Instruction Address Register	597
14.5.55 IESR1 - IU Event Select Register 1	598
14.5.56 IESR2 - IU Event Select Register 2	599
14.5.57 IMMR - Instruction Match Mask Register	600
14.5.58 IMPDEP0 - Implementation Dependant Region 0	601
14.5.59 IMPDEP1 - Implementation Dependant Region 1	602
14.5.60 IMR - Instruction Match Register	603
14.5.61 IUCR0 - Instruction Unit Configuration Register 0	604
14.5.62 IUCR1 - Instruction Unit Configuration Register 1	605
14.5.63 IUCR2 - Instruction Unit Configuration Register 2	606
14.5.64 IUDBG0 - Instruction Unit Debug Register 0	607
14.5.65 IUDBG1 - Instruction Unit Debug Register 1	608
14.5.66 IUDBG2 - Instruction Unit Debug Register 2	609
14.5.67 IULFSR - Instruction Unit LFSR	610
14.5.68 IULLCR - Instruction Unit Live Lock Control Register	611
14.5.69 IVPR - Interrupt Vector Prefix Register	612
14.5.70 LESR1 - LQ Event Select Register 1	613
14.5.71 LESR2 - LQ Event Select Register 2	614
14.5.72 LPER - Logical Page Exception Register	615
14.5.73 LPERU - Logical Page Exception Register (Upper)	616
14.5.74 LPIDR - Logical Partition ID Register	617
14.5.75 LR - Link Register	618
14.5.76 LRATCFG - LRAT Configuration Register	619
14.5.77 LRATPS - LRAT Page Size Register	620
14.5.78 LSUCR0 - Load Store Configuration Register 0	621
14.5.79 MAS0 - MMU Assist Register 0	622
14.5.80 MAS0_MAS1 - MMU Assist Registers 0 & 1	623
14.5.81 MAS1 - MMU Assist Register 1	624
14.5.82 MAS2 - MMU Assist Register 2	626
14.5.83 MAS2U - MMU Assist Register 2 (Upper)	627

14.5.84 MAS3 - MMU Assist Register 3	628
14.5.85 MAS4 - MMU Assist Register 4	630
14.5.86 MAS5 - MMU Assist Register 5	631
14.5.87 MAS5_MAS6 - MMU Assist Registers 5 & 6	632
14.5.88 MAS6 - MMU Assist Register 6	633
14.5.89 MAS7 - MMU Assist Register 7	634
14.5.90 MAS7_MAS3 - MMU Assist Registers 7 & 3	635
14.5.91 MAS8 - MMU Assist Register 8	636
14.5.92 MAS8_MAS1 - MMU Assist Registers 8 & 1	637
14.5.93 MCSR - Machine Check Syndrome Register	638
14.5.94 MCSRR0 - Machine Check Save/Restore Register 0	639
14.5.95 MCSRR1 - Machine Check Save/Restore Register 1	640
14.5.96 MESR1 - MMU Event Select Register 1	642
14.5.97 MESR2 - MMU Event Select Register 2	643
14.5.98 MMUCFG - MMU Configuration Register	644
14.5.99 MMUCR0 - Memory Management Unit Control Register 0	645
14.5.100 MMUCR1 - Memory Management Unit Control Register 1	646
14.5.101 MMUCR2 - Memory Management Unit Control Register 2	648
14.5.102 MMUCR3 - Memory Management Unit Control Register 3	650
14.5.103 MMUCSR0 - MMU Control and Status Register 0	651
14.5.104 MSR - Machine State Register	652
14.5.105 MSRP - Machine State Register Protect	654
14.5.106 PID - Process ID	655
14.5.107 PIR - Processor ID Register	656
14.5.108 PPR32 - Program Priority Register	657
14.5.109 PVR - Processor Version Register	658
14.5.110 RESR1 - RV Event Select Register 1	659
14.5.111 RESR2 - RV Event Select Register 2	660
14.5.112 SIAR - Sampled Instruction Address Register	661
14.5.113 SPRG0 - Software Special Purpose Register 0	662
14.5.114 SPRG1 - Software Special Purpose Register 1	663
14.5.115 SPRG2 - Software Special Purpose Register 2	664
14.5.116 SPRG3 - Software Special Purpose Register 3	665
14.5.117 SPRG4 - Software Special Purpose Register 4	666
14.5.118 SPRG5 - Software Special Purpose Register 5	667
14.5.119 SPRG6 - Software Special Purpose Register 6	668
14.5.120 SPRG7 - Software Special Purpose Register 7	669
14.5.121 SPRG8 - Software Special Purpose Register 8	670
14.5.122 SRAMD - Shadowed RAMD Register	671
14.5.123 SRR0 - Save/Restore Register 0	672
14.5.124 SRR1 - Save/Restore Register 1	673
14.5.125 TB - Timebase	675
14.5.126 TBL - Timebase Lower	676
14.5.127 TBU - Timebase Upper	677
14.5.128 TCR - Timer Control Register	678
14.5.129 TENC - Thread Enable Clear Register	680
14.5.130 TENS - Thread Enable Set Register	681
14.5.131 TENSr - Thread Enable Status Register	682
14.5.132 TIR - Thread Identification Register	683
14.5.133 TLB0CFG - TLB 0 Configuration Register	684

14.5.134 TLB0PS - TLB 0 Page Size Register	685
14.5.135 TRACE - Hardware Trace Macro Control Register	686
14.5.136 TSR - Timer Status Register	687
14.5.137 UDEC - User Decrementer	688
14.5.138 VRSAVE - Vector Register Save	689
14.5.139 XER - Fixed Point Exception Register	690
14.5.140 XESR1 - XU Event Select Register 1	691
14.5.141 XESR2 - XU Event Select Register 2	692
14.5.142 XUCR0 - Execution Unit Configuration Register 0	693
14.5.143 XUCR1 - Execution Unit Configuration Register 1	696
14.5.144 XUCR2 - Execution Unit Configuration Register 2	697
14.5.145 XUCR4 - Execution Unit Configuration Register 4	698
14.5.146 XUDBG0 - Execution Unit Debug Register 0	699
14.5.147 XUDBG1 - Execution Unit Debug Register 1	700
14.5.148 XUDBG2 - Execution Unit Debug Register 2	701
15. SCOM Accessible Registers	703
15.1 Serial Communications (SCOM) Description	703
15.2 SCOM Register Summary	705
15.2.1 SCOM register access methods	705
15.2.1.1 Register access terminology	705
15.2.1.2 Reset with AND mask (WOAND)	705
15.2.1.3 Set with OR mask (WOOR)	705
15.2.2 SCOM register summary table	705
15.2.3 SCOM register descriptions (alphabetical order)	707
15.2.3.1 AXU/RV Debug Select Register (ARDSR)	707
15.2.3.2 Error Injection Register (ERRINJ)	709
15.2.3.3 Fault Isolation Registers and their associated Action and Mask registers	710
15.2.3.4 IU Debug Select Register (IDSR)	718
15.2.3.5 LQ Debug Select Register (LDSR)	720
15.2.3.6 MMU/PC Debug Select Register (MPDSR)	722
15.2.3.7 PC Configuration Register 0 (PCCR0)	724
15.2.3.8 Ram Data Registers (RAMD, RAMDH, RAMDL)	725
15.2.3.9 Ram Instruction and Command Registers (RAMC, RAMI, RAMIC)	726
15.2.3.10 Special Attention Register (SPATTN)	729
15.2.3.11 Thread Control and Status Register (THRCTL)	730
15.2.3.12 XU Debug Select Register (XDSR)	732
Appendix A. Processor Instruction Summary	735
A.1 Instruction Formats	735
A.2 Implemented Instructions Sorted by Mnemonic	735
Appendix B. FU Instruction Summary	754
B.1 FU Instructions Sorted by Opcode	754
Appendix C. Debug and Trigger Groups	773
C.1 Unit Debug Mux Component	773
C.2 Debug Mux Component Ordering on the Ramp Bus	773
C.3 Example Debug Mux Configuration Settings	773



C.4 Debug Select Registers and Debug Group Tables 773

List of Figures

Figure 1-1.	A2 Core Organization	49
Figure 1-2.	A2O Processor Block Diagram	54
Figure 2-1.	A2 Core Instruction Unit	77
Figure 2-2.	User Programming Model Registers	80
Figure 3-1.	Approximation to Real Numbers	131
Figure 3-2.	Selection of z1 and z2	136
Figure 4-1.	Software-Initiated Reset Request Overview	151
Figure 6-1.	Virtual Address to TLB Entry Match Process	174
Figure 6-2.	Effective-to-Real Address Translation Flow	176
Figure 6-3.	ERAT Entry Word Definitions	204
Figure 6-4.	ERAT Entry Word Definitions for 32-Bit Mode	211
Figure 6-5.	Indirect Entry to Page Table Size Calculation	222
Figure 6-6.	Page Table Entry Format	223
Figure 9-1.	Relationship of Timer Facilities to the Time Base	372
Figure 9-2.	Guest Watchdog State Machine	383
Figure 10-1.	Pass-Through Trace and Trigger Bus Overview	437
Figure 10-2.	Unit Trace and Trigger Bus, and debug_mux Component Description	438
Figure 11-1.	Performance event selection overview	441
Figure 11-2.	A2 common unit event multiplexer component	447
Figure 12-1.	ICSWX (RS _{32:63}) Coprocessor-Command Word	511
Figure 12-2.	Coprocessor Command Word (CCW)	512
Figure 12-3.	Generic Coprocessor-Request Block	514
Figure 15-1.	Chip Level Infrastructure Example to Access SCOM Registers in the A2 Core	704
Figure 15-2.	Principle Timing of Information Carried on CCH and DCH	704



List of Tables

Table 2-1.	Data Operand Definitions	61
Table 2-2.	Alignment Effects for Storage Access Instructions	61
Table 2-3.	Priority Levels	73
Table 2-4.	Other “or” Instruction Hints	74
Table 2-5.	Program Priority Register (PPR32)	74
Table 2-6.	Register Mapping	81
Table 2-7.	Category Listing	82
Table 2-8.	Instruction Categories	86
Table 2-9.	Integer Storage Access Instructions	87
Table 2-10.	Integer Storage Access Instructions by External Process ID	87
Table 2-11.	Operand Handling Dependent on Alignment	87
Table 2-12.	Integer Arithmetic Instructions	89
Table 2-13.	Integer Logical Instructions	89
Table 2-14.	Integer Compare Instructions	89
Table 2-15.	Integer Trap Instructions	89
Table 2-16.	Integer Rotate Instructions	90
Table 2-17.	Integer Shift Instructions	90
Table 2-18.	Integer Population Count Instructions	90
Table 2-19.	Integer Select Instruction	90
Table 2-20.	Binary Coded Decimal Assist Instructions	91
Table 2-21.	Branch Instructions	91
Table 2-22.	Condition Register Logical Instructions	92
Table 2-23.	Register Management Instructions	92
Table 2-24.	System Linkage Instructions	92
Table 2-25.	Processor Control Instruction	93
Table 2-26.	Cache Management Instructions	93
Table 2-27.	Cache Management Instructions by External Process ID	93
Table 2-28.	Storage Visibility Instructions	93
Table 2-29.	TLB Management Instructions	94
Table 2-30.	Processor Synchronization Instruction	94
Table 2-31.	Load and Reserve and Store Conditional Instructions	94
Table 2-32.	Storage Synchronization Instructions	95
Table 2-33.	Wait Instruction	95
Table 2-34.	Initiate Coprocessor Instructions	95
Table 2-35.	Cache Initialization Instructions	95
Table 2-36.	Debug Instructions	96
Table 2-37.	BO Field Encodings	97
Table 2-38.	‘at’ Bit Encodings	98

Table 2-39.	98
Table 2-40.	99
Table 2-41.	99
Table 2-42.	CR Updating Instructions	104
Table 2-43.	GPR Registers	106
Table 2-44.	XER[SO,OV] Updating Instructions	107
Table 2-45.	XER[CA] Updating Instructions	108
Table 2-46.	SPRG0 Register	110
Table 2-47.	SPRG1 Register	110
Table 2-48.	SPRG2 Register	111
Table 2-49.	SPRG3 Register	111
Table 2-50.	SPRG4 Register	111
Table 2-51.	SPRG5 Register	111
Table 2-52.	SPRG6 Register	112
Table 2-53.	SPRG7 Register	112
Table 2-54.	SPRG8 Register	112
Table 2-55.	GSPRG0 Register	113
Table 2-56.	GSPRG1 Register	113
Table 2-57.	GSPRG2 Register	113
Table 2-58.	GSPRG3 Register	113
Table 2-59.	Privileged Instructions	117
Table 3-1.	Data Operand Definitions	124
Table 3-2.	Invalid Operation Exception Categories	125
Table 3-3.	Floating-Point Registers (FPR0–FPR31)	126
Table 3-4.	Floating-Point Status and Control Register (FPSCR)	127
Table 3-5.	Floating-Point Single Format	130
Table 3-6.	Floating-Point Double Format	130
Table 3-7.	Format Fields	130
Table 3-8.	IEEE 754 Floating-Point Fields	130
Table 3-9.	Rounding Modes	136
Table 3-10.	IEEE 64-Bit Execution Model	137
Table 3-11.	Interpretation of the G, R, and X Bits	137
Table 3-12.	Location of the Guard, Round, and Sticky Bits in the IEEE Execution Model	138
Table 3-13.	Multiply-Add 64-Bit Execution Model	139
Table 3-14.	Location of Guard, Round, and Sticky Bits in the Multiply-Add Execution Model	139
Table 3-15.	Floating-Point Load Instructions	142
Table 3-16.	Floating-Point Store Instructions	143
Table 3-17.	Floating-Point Move Instructions	144
Table 3-18.	Floating-Point Elementary Arithmetic Instructions	144

Table 3-19.	Floating-Point Multiply-Add Instructions	145
Table 3-20.	Floating-Point Rounding and Conversion Instructions	146
Table 3-21.	Comparison Sets	146
Table 3-22.	Floating-Point Compare and Select Instructions	147
Table 3-23.	Floating-Point Status and Control Register Instructions	147
Table 5-1.	Data Cache Array Organization	153
Table 5-2.	Cache Size and Parameters	153
Table 5-3.	Instruction Cache Array Organization	154
Table 5-4.	Cache Size and Parameters	154
Table 5-5.	XUCR Bits	167
Table 6-1.	Page Size and Effective Address to EPN Comparison	175
Table 6-2.	Page Size and Real Address Formation	176
Table 6-3.	Access Control Applied to Cache Management Instructions	178
Table 6-4.	TLB Entry Fields	183
Table 6-5.	ERAT Class Field Reload Value For UTLB Hits	192
Table 6-6.	LRAT Entry Fields	195
Table 6-7.	TLB Management Instruction Privilege Levels	196
Table 6-8.	TLB Congruence Class Hashing Function (of EPN Address Bits)	198
Table 6-9.	Supported EPN[27:51] Field Values in Downbound TLBIVAX Request	202
Table 6-10.	ERAT Management Instruction Privilege Levels	203
Table 6-11.	Summary of Supported IS Field Values in ERATIVAX	206
Table 6-12.	Supported EPN[27:51] Field Values in Downbound erativax Request	208
Table 6-13.	TLB Reservation Fields	217
Table 6-14.	TLB Update After Page Table Translation	226
Table 6-15.	MAS Register Update Summary	259
Table 7-1.	Register Mapping in Guest State	285
Table 7-2.	Interrupt Types and Associated Offsets	300
Table 7-3.	Interrupt and Exception Types	306
Table 8-1.	Invalid Operation Exception Categories	356
Table 8-2.	MSR[FE0, FE1] Modes	358
Table 8-3.	Invalid Operation Exceptions	360
Table 8-4.	QNaN Result	365
Table 8-5.	FPSCR[FPRF] Result Flags	366
Table 8-6.	Floating-Point Status and Control Register (FPSCR)	367
Table 8-7.	Bit Encodings for a CR Field	370
Table 9-1.	Timebase Register (TB)	373
Table 9-2.	Timebase Lower Register (TBL)	373
Table 9-3.	Timebase Upper Register (TBU)	374
Table 9-4.	Decrementer Register (DEC)	375



Table 9-5.	Decrementer Auto-Reload Register (DECAR)	375
Table 9-6.	Fixed Interval Timer Period Selection	378
Table 9-7.	Fixed Interval Timer Period Selection	379
Table 9-8.	Watchdog Timer Period Selection	380
Table 9-9.	Watchdog Timer Exception Behavior	380
Table 9-10.	Guest Watchdog Timer Period Selection	381
Table 9-11.	Guest Watchdog Timer Exception Behavior	382
Table 10-1.	PCCR0[DBA] (Debug Action) Definition per Thread	390
Table 10-2.	Debug Events	392
Table 10-3.	Debug Event Summary	405
Table 10-4.	PC Configuration Register 0	421
Table 10-5.	Thread Control and Status Register	424
Table 10-6.	Example Ram procedure	431
Table 11-1.	Core event multiplexer to external event bus	443
Table 11-2.	Performance monitor event tags	448
Table 11-3.	AXU0 Performance Events Table	449
Table 11-4.	IU Performance Events Table	450
Table 11-5.	LQ Performance Events Table	453
Table 11-6.	MMU Performance Events Table	458
Table 11-7.	RV Performance Events Table	460
Table 11-8.	FX0 Performance Events Table	461
Table 14-1.	Register Summary	522
Table 15-1.	PC Unit SCOM Register Summary	705
Table 15-2.	AXU and RV Debug Select Register	707
Table 15-3.	Error Injection Register	709
Table 15-4.	Fault Isolation Register 0	711
Table 15-5.	FIR0 Action and Mask Registers	713
Table 15-6.	FIR0 and FIR1 Registers (Read Only)	714
Table 15-7.	Fault Isolation Register 1	714
Table 15-8.	FIR1 Action and Mask Registers	715
Table 15-9.	Fault Isolation Register 2	716
Table 15-10.	FIR2 Action and Mask Registers	717
Table 15-11.	IU Debug Select Register	718
Table 15-12.	LQ Debug Select Register	720
Table 15-13.	MMU and PC Debug Select Register	722
Table 15-14.	PC Configuration Register 0	724
Table 15-15.	Ram Data Register	725
Table 15-16.	Ram Data Register High	725
Table 15-17.	Ram Data Register Low	726

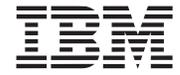
Table 15-18. Ram Command Register	726
Table 15-19. Ram Instruction Register	728
Table 15-20. Ram Instruction and Command Registers	728
Table 15-21. Special Attention Register	729
Table 15-22. Thread Control and Status Register	730
Table 15-23. XU Debug Select Register	732
Table A-1. A2 Core Instructions by Mnemonic	736
Table B-1. FU Instructions by Opcode	754
Table B-2.	758



Revision Log

Each release of this document supersedes all previously released versions. The revision log lists all significant changes made to the document since its initial release. In the rest of the document, change bars in the margin indicate that the adjacent text was modified from the previous release of this document.

Revision Date	Pages	Description
December 15, 2010	—	Version 1.0. Initial release.
June 21, 2011	339	Clarified LRAT miss exception due to tlbwe dependency on MAS0.WQ and TLB-reservation in <i>Section 7.6.28 LRAT Error Interrupt</i> .
November 9, 2011	216	Added <i>Section 6.14 TLB and ERAT Multi-hit Operations</i> to clarify multi-hit error scenarios. Also miscellaneous clarifications in <i>Section 6.13 TLB and ERAT Parity Operations</i> .
December 1, 2011	215, 265	Added clarifications for parity error injection to <i>Section 6.13.2 Simulating TLB and ERAT Parity Errors for Software Testing</i> and <i>Section 6.19.2 Memory Management Unit Control Register 1 (MMUCR1)</i> .
February 2, 2012	265	Added engineering note about multi-hit and parity error recording to the end of <i>Section 6.19.2 Memory Management Unit Control Register 1 (MMUCR1)</i> .
September 15, 2020		Start updates for open release.



About This Book

This user’s manual provides the architectural overview, programming model, and detailed information about the instruction set, registers, and other facilities of the IBM® Power ISA A2O 64-bit embedded processor core.

The A2O embedded controller core features:

- Power ISA Architecture
- Concurrent-in pipeline with dynamic branch prediction
- Separate 16 KB each instruction and data caches
- Memory management unit (MMU) with a 512-entry translation lookaside buffer (TLB)
- 4 TB (42-bit) physical address capability
- 16-bit load interface and 128-bit store interface
- ANSI/IEEE 754-1985 compliant floating-point¹
- Single-precision and double-precision operation in hardware
- Auxiliary execution unit (AXU) that executes the Power ISA floating-point instruction set
- Super-pipelined: Single cycle throughput for most instructions
- In-order execution and completion

Who Should Use This Book

This book is for system hardware and software developers and for application developers who need to understand the A2O core. The audience should understand embedded system design, operating systems, RISC microprocessing, and computer organization and architecture.

How to Use This Book

This book describes the A2O Processor device architecture, programming model, registers, and instruction set. This book contains the following chapters:

- *Overview* on page 45
- *CPU Programming Model* on page 59
- *FU Programming Model* on page 123
- *Initialization* on page 149
- *Instruction and Data Caches* on page 153
- *Memory Management* on page 169
- *CPU Interrupts and Exceptions* on page 277
- *FU Interrupts and Exceptions* on page 355
- *Timer Facilities* on page 371

1. Power ISA FUs require software support for IEEE compliance.

- *Debug Facilities* on page 389
- *Performance Events and Event Selection* on page 441
- *Implementation Dependent Instructions* on page 475
- *Power Management Methods* on page 517
- *Register Summary* on page 521
- *SCOM Accessible Registers* on page 703

This book contains the following appendixes:

- *Processor Instruction Summary* on page 735
- *FU Instruction Summary* on page 754
- on page 773
- *Instruction Execution Performance and Code Optimizations* on page 833
- *Programming Examples* on page 861

Notation

The manual uses the following notational conventions:

- Active low signals are shown with an overbar ($\overline{\text{Active_Low}}$).
- All numbers are decimal unless specified in some special way.
 - 0bnnnn means a number expressed in binary format.
 - 0xn timer means a number expressed in hexadecimal format.

Underscores might be used between digits.

- RA refers to General Purpose Register (GPR) RA.
- (RA) refers to the contents of GPR RA.
- (RA|0) refers to the contents of GPR RA or to the value 0 if the RA field is 0.
- Bits in registers, instructions, and fields are specified as follows.
 - Bits are numbered most-significant bit to least-significant bit, starting with bit 0.
 - X_p means bit p of register, instruction, or field X.
 - $X_{p:q}$ means bits p through q of a register, instruction, or field X.
 - $X_{p,q,\dots}$ means bits p, q,... of a register, instruction, or field X.
 - $X[p]$ means a named field p of register X.
 - $X[p:q]$ means named fields p through q of register X.
 - $X[p,q,\dots]$... means named fields p, q,... of register X.
- $\neg X$ means the ones complement of the contents of X.
- A period (.) as the last character of an instruction mnemonic means that the instruction records status information in certain fields of the Condition Register as a side effect of execution, as described in *Section 12 Implementation Dependent Instructions* on page 475.

- The symbol `||` is used to describe the concatenation of two values. For example, `0b010 || 0b111` is the same as `0b010111`.
- x^n means x raised to the n power.
- ${}^n x$ means the replication of x , n times (that is, 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 `0b00000`.
 - ${}^n 1$ means a field of n bits with each bit equal to 1. Thus ${}^5 1$ is equivalent to `0b11111`.
- `/, //, ///, ...` denotes a reserved field in an instruction or in a register.
- `?` denotes an allocated bit in a register.
- A shaded field denotes a field that is reserved or allocated in an instruction or in a register.

Related Publications

- *Power ISA User Set Architecture* (Book I, Version 2.06)
- *Power ISA Virtual Environment Architecture* (Book II, Version 2.06)
- *Power ISA Operating Environment Architecture* (Book III-E, Version 2.06)

The Power ISA specifications are available at www.openpowerfoundation.org.



List of Acronyms and Abbreviations

ABIST	automatic built-in self test
ALU	arithmetic logic unit
ANSI	American National Standards Institute
ARE	auto-reload enable
AS	address space
ATB	alternate time base category
attn	attention
AXU	auxiliary execution unit
B	base category
BCLR	branch conditional to Link Register
BE	big endian
BHT	branch history table
BP	branch prediction
BRDCAST	broadcast
BRT	branch taken
BTA	branch target address
CA	carry
CAM	content addressable memory
CC	congruence class
CCH	control channel
CCW	coprocessor command word
CD	coprocessor directive
CEE	change exception enable
CI	coprocessor instance
CIA	current instruction address
CPU	central processing unit
CRB	coprocessor-request block
CS	cache specification category

CSB	control status block
CSI	context synchronizing instruction
DAC	data address compare
DBA	debug action
DBELL	doorbell interrupt
DCC	data cache controller
DCH	data channel
DCI	data cache invalidate instruction
DCR	device control register
DEA	data effective address
DEC	decrementer
D-ERAT	data ERAT
DERRDET	D-ERAT error detect
DFP	decimal floating-point category
DL	downlink
DRA	data real address
DSI	data storage interrupt
DSP	digital signal processor
DVC	data value compare
E	endian or embedded category
E.CD	embedded.cache debug category
E.CI	embedded.cache initialization category
E.DC	embedded.device control category
E.ED	embedded.enhanced debug category
E.HV	embedded.hypervisor category
E.LE	embedded.little-endian category
E.PC	embedded.processor control category
E.PD	embedded.external PID category
E.PM	embedded.performance monitor category

E.PT	embedded.page table category
E.TWC	embedded.tlb write conditional category
EA	effective address
ECC	error-correcting code
ECL	embedded cache locking category
EDM	external debug mode
EEN	error entry number
EH	exclusive access hint
EM	embedded multithreading category
EM.TM	embedded multithreading.thread management category
EPID	external PID
EPLC	external process ID load context
EPN	effective page number
EPR	external problem state bit
EPSC	external process ID store context
ERAT	effective to real address translation
ESID	effective segment ID
EVPR	Exception Vector Prefix Register
EXC	external control category
EXP	external proxy category
FE	floating-point equal
FG	floating-point greater than
FIFO	first-in, first out
FIR	fault isolation register
FIT	fixed interval timer
FL	floating-point less than
FP	floating-point category
FP.R	floating-point.record category
FPR	floating-point register

FU	floating-point unit
FXU	fixed-point unit
G	guarded
GB	gigabyte
GB/sec	gigabytes per second
GHz	gigahertz
GPR	general purpose register
GS	guest state
HTM	hardware trace macro
HWT	hardware table walker
I	caching inhibited
I/O	input/output
IAC	instruction address compare
IEA	instruction effective address
IBUFF	instruction buffer
ICC	instruction cache controller
ICI	instruction cache immediate instruction
ICMP	instruction complete
IDE	imprecise debug event
IEA	instruction effective address
IEEE	Institute of Electrical and Electronics Engineers
I-ERAT	instruction ERAT
IERRDET	I-ERAT error detect
IFAR	Instruction Fetch Address Register
IND	indirect
INSTTRACE	instruction trace mode
I/O	input/output
IR	intermediate result
IRPT	interrupt

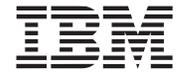
IS	instruction fetch address space OR invalidation select
ISA	instruction set architecture
ISI	instruction storage interrupt
IU	instruction unit
IU0 - IU6	instruction unit pipeline stage
IVC	instruction value compare
JTAG	Joint Test Action Group
KB	kilobyte
LA	logical address
L1	level 1
L2	level 2
LA	logical address
LBIST	logic built-in self-test
LE	little endian
LIFO	last-in, first-out
LMA	legacy integer multiply-accumulate1 category
LMQ	load miss queue
LMV	legacy move assist category
LPID	logical partition identifier
LPIDTAG	LPID tag
LPN	logical page number
LRAT	logical to real address translation
LRU	least recently used
LSb	least significant bit
LSB	least significant byte
LSQ	load/store quadword category
LSU	load/store unit
M	memory coherence required
MA	move assist category

MAS	MMU assist
MAV	MMU Architecture version
MB	megabyte
MESI	modified, exclusive, shared, invalid
MHz	megahertz
MMC	memory coherence category
MMU	memory management unit
MSB	most significant byte
MSRP	Machine State Register protect
MT	multithread
NaN	Not a Number
NAND	not AND
NH	next higher in magnitude
NIA	next instruction address
NL	next lower in magnitude
NOR	not OR
OV	overflow
OX	overflow exception
PC	processor control
PCB	pervasive control bus
PCR	processor compatibility category
PIB	pervasive interconnect bus
PID	processor ID
PIRTAG	PIR tag
PME	power-management
PMU	performance monitor unit
POR	power-on reset
PS	page size specified by PTE
PTE	page table entry

QNaN	quiet NaN
RA	real address
RAW	read-after-write
REE	reference exception enable
RET	return
RISC	reduced instruction set computing
RMT	replacement management table
RO	read only
ROM	read-only memory
RPN	real page number
S	server category
S.PM	server.performance monitor category
S.RPTA	server.relaxed page table alignment category
SAO	strong access order category
SCOM	serial communications
SCPM	store conditional page mobility category
SEM	sequential execution model
SER	soft error rate
SIMD	single instruction, multiple data
SLB	segment lookaside buffer
SNaN	signalling NaN
SO	summary overflow
SOC	system-on-a-chip
SP	signal processing engine category
SPE	signal processing engine
SP.FD	SPE.embedded float scalar double category
SP.FS	SPE.embedded float scalar single category
SP.FV	SPE.embedded float vector category
SPR	Special Purpose Register

SPRN	special purpose register number
SPRG	Special Purpose Registers General
SR	supervisor mode read access
SRAM	static random access memory
STM	stream category
SW	supervisor mode write access
SX	supervisor mode execution access
TB	terabyte
TBC	transfer byte count
TBL	time base lower
TBU	time base upper
TERRDET	TLB error detect
TGS	translation guest space identifier
TID	translation ID
TLB	translation lookaside buffer
TLPID	translation logical partition identifier
TRC	trace category
TS	translation space identifier
UC	microcode unit or uncorrectable error
uCode	microcode
UCT	unavailable coprocessor type
UDE	unconditional debug event
UDEC	user decrementer
UE	underflow exception
UL	uplink
UND	undefined
UR	user mode read access
UTLB	unified translation lookaside buffer
UW	user mode write access

UX	underflow exception or user mode execution access
V	vector category
V.LE	little-endian category
VA	virtual addresses
VF	virtualization fault
VHDL	very-high-speed integrated circuit (VHSIC) hardware description language
VLE	variable length encoding category
VLPT	virtual linear page table
VPN	virtual page number
VSID	virtual segment ID
VSX	vector-scalar extension category
VX	invalid operation exception
W	write-through
WAW	write-after-write
WC	wake control or write to clear
WDT	watchdog timer
WIMGE	write-through, caching-inhibited, memory coherency required, guarded, and endianness attributes
WP	watchdog timer period
WS	write to set
WT	wait category
XOR	exclusive OR
XU	execution unit
ZX	zero divide exception



1. Overview

The IBM™ PowerISA™ A2O 64-bit embedded processor core is an implementation of the scalable and flexible PowerISA architecture. The A2O core implements a super-scalar, out-of-order single thread of execution design.

Full floating point capability is supported by the floating point unit attached to the AXU interface.

1.1 A2O Core Key Design Fundamentals

The key design fundamentals of the A2O core are the following:

- 64-bit implementation of the PowerISA Version 2.07 Book III E - Embedded Platform Environment
 - A2O core provides binary compatibility for PowerPC application level code (problem state)
 - A2O core implements the Embedded Hypervisor architecture to provide secure compute domains and operating system virtualization
- The A2O core is optimized for single thread performance
 - Single thread, Super-scalar, out-of-order execution design
 - 2 Instruction dispatch, 4 Instruction issue
 - 27 FO4 design
- The A2O core is a modular design to support reuse
 - The A2O core provides a general purpose co-processor (AXU) port to attached unique AXUs
 - AXUs have full ISA flexibility
 - AXUs currently include
 - FPU - PowerISA V2.07 Scalar Double Precision Floating Point Unit
 - The AXU is an optional unit
 - The A2O core provides for an optional MMU unit
 - The MMU unit supports PowerISA V2.07 Book III E Memory Management (MAV 2.0)
 - Without the MMU the A2O core supports software-managed ERATs defined in this document
 - The A2O core provides for an optional Microcode Engine and ROM
 - PowerISA V2.07 Book I and II instructions are supported with a combination of Microcoded instructions and hardware implemented instructions

1.2 A2O Core Features

The A2O Core is a high-performance, low-power engine that implements the flexible and powerful 64-bit Power ISA Architecture.

A2O Core features include:

- Instruction Fetch
 - 32K, 4 way, I-cache
 - 64-byte cacheline

- 16 entry fully associative ERAT
- Fetch 4 instructions per cycle
- 16-entry instruction buffer
- Rename, dispatch and retire two instructions per cycle
- 32-entry completion queue
- High-accuracy dynamic branch prediction
 - Static hardware, dynamic hardware, and software branch prediction
 - Resolves one branch per cycle
 - CR logicals
 - 4 simultaneous predictions per cycle
 - 3 branch history tables (BHT)
 - Two 2-bits x 4 instructions x 1k entries = 8Kb per table
 - One 1-bit x 4 instructions x 512 entries = 2k table
 - BTB/BTAC for early prediction of target addresses, size TBD
 - 8-entry link stack for target address prediction of subroutine-return indirect branches
- Reservation Station (RV)
 - Unified instruction target in flight scorecard
 - Separate reservation station per execution unit
 - Each can accept up to two instructions per cycle from dispatch (i.e. two FXU0)
 - Oldest instruction issues if source operands available
 - Instruction restart support
 - Up to 16 entries per queue
 - Data forwarding between pipelines adds one cycle of latency for all operations
 - Full bypass of GPR, XER, CR, LR, and CTR
- Load Store Unit (LSU)
 - 32K, 8way, D-cache
 - 64 byte cacheline
 - 32 entry fully associative ERAT
 - Store-through cache
 - Back-invalidate support for snooping
 - Non-blocking (hit under miss)
 - L1 hit has 4-cycle load-to-use
 - 16-byte load/store support between AXU and D-Cache
 - Line locking, way locking
 - Full little-endian support
 - Misaligned support within a 16-byte granule, accesses across 16-byte granule is micro-op'ed
 - 8-entry load miss queue
 - 8-entry store queue
 - 16-entry load/store out-of-order queue
 - Separate load/store address and store data pipelines, each with a reservation station
- Instruction Execution (FXU0, FXU1)
 - One-cycle latency for simple adds and subtracts
 - All other adds, subtracts, logicals, shifts and rotates are two cycles
 - Multiplies and divides use FXU0
 - Multiply latency of 3-6 cycles
 - 2-bit per cycle divider with early exit
- Double Precision Float (FU)
 - Fully renamed Power double-precision floating point.

- 6 cycle floating point latency
- Highly-pipelined micro-architecture
 - Full GPR Bypass
 - Full CR Bypass
 - Link Register Bypass
- Primary caches
 - Separate instruction and data cache arrays
 - Array size offerings: 32KB
 - Single-cycle access
 - 64-byte line size
 - 8-way set-associative D-Cache, 4-way set-associative I-Cache
 - Write-through operation
 - Unified (for all threads) non-blocking, with up to 8 outstanding load misses
 - Cache line locking supported
 - Caches can be partitioned to provide separate regions for “transient” data
 - Critical-word-first data access and forwarding
 - Pseudo-LRU replacement policy
 - Cache tags and data are parity protected. Errors are recoverable.
- Memory Management Unit (MMU)
 - Support for PowerISA Categories Embedded.Hypervisor (E.HV), Embedded.Hypervisor.LRAT (E.HV.LRAT), and Embedded.TLB Write Conditional (E.TWC), and Embedded.Page Table (E.PT).
 - Support for PowerISA Book III-E MMU Architecture Version 2.0 (MAV 2.0).
 - Separate instruction and data ERATs
 - Fully-associative 16-entry I-ERAT shared by all threads
 - Fully-associative 32-entry D-ERAT shared by all threads
 - Exclusion range function to allow address “holes” at base of page entries
 - ERATs operate in one of two modes: MMU mode or ERAT-only mode
 1. MMU Mode; ERAT w/ backing MMU
 - Software-managed page tables and indirect (IND=1) TLB entries
 - Hardware handles ERAT miss with TLB hit
 - Hardware handles direct (IND=0) TLB miss via hardware page table walking
 - Software handles indirect (IND=1) TLB miss via I/D TLB miss exceptions
 - Software may also install direct (IND=0) TLB entries as required
 2. ERAT-only Mode; Effective-to-Real Address Translation w/ ERATs Only
 - MMU removed, No backing TLB
 - Software managed ERAT entries – I/D TLB miss exceptions
 - 512-entry, 4-way set-associative unified TLB array
 - Variable page sizes for direct (IND=0) entries (4KB, 64KB, 1MB, 16MB, 1GB), simultaneously resident in TLB and/or ERAT, and indirect (IND=1) entries (1 MB and 256 MB) in TLB
 - 88-bit virtual address (contains 64-bit effective address)
 - 42-bit (4 TB) real addressability
 - Flexible TLB management via software management, or via hardware page table search
 - Flexible storage attribute controls for write-through, caching inhibited, coherent, guarded, and byte order (endianness)
 - Four user-definable storage attribute controls
 - TLB tags and data are parity protected against soft errors.
- Intelligent Data Prefetching

- Learns strides for specific load instructions
- High prefetching accuracy
- Debug facilities
 - Extensive hardware debug facilities
 - Multiple instruction and data address breakpoints
 - Data value compare
 - Instruction value compare
 - Single-step, branch, trap, and other debug events
 - Non-invasive real-time software trace interface
- Timer facilities
 - 64-bit time base
 - Decrementer with auto-reload capability
 - Fixed Interval Timer (FIT)
 - Watchdog Timer with critical interrupt and/or auto-reset
- Multiple core Interfaces
 - System interface
 - A command interface for instruction reads, data reads, and data writes
 - A 128-bit interface for data writes
 - A 128-bit interface for instruction reads and data reads
 - An invalidate interface to the core for the system to maintain L1 cache coherency
 - Auxiliary Processor Unit (AXU) Port
 - Allows full ISA flexibility
 - AXU includes support for separate decode and dependency
 - Full support to stall and flush the processor
 - A20 core pipeline exposed to allow high performance tightly coupled Co-processors
 - Provides functional extensions to the processor pipelines
 - 128-bit load/store interface (direct access between AXU and the primary data cache)
 - Interface can support AXU execution of all PowerISA floating point instructions
 - Attachment capability for DSP co-processing such as accumulators and SIMD computation
 - Enables customer-specific instruction enhancements for unique applications
 - Clock and power management interface
 - Debug interface
 - Performance Monitor Event interface
- Floating Point Unit Features
 - IEEE 754-1985 compliance¹
 - Single-precision and double-precision operation in hardware
 - Executes PowerISA Floating point instruction set
 - Masked exceptions handled in hardware
 - Superpipelined; single-cycle throughput for most instructions
 - Out-of-order execution
 - Single instruction decode and issue

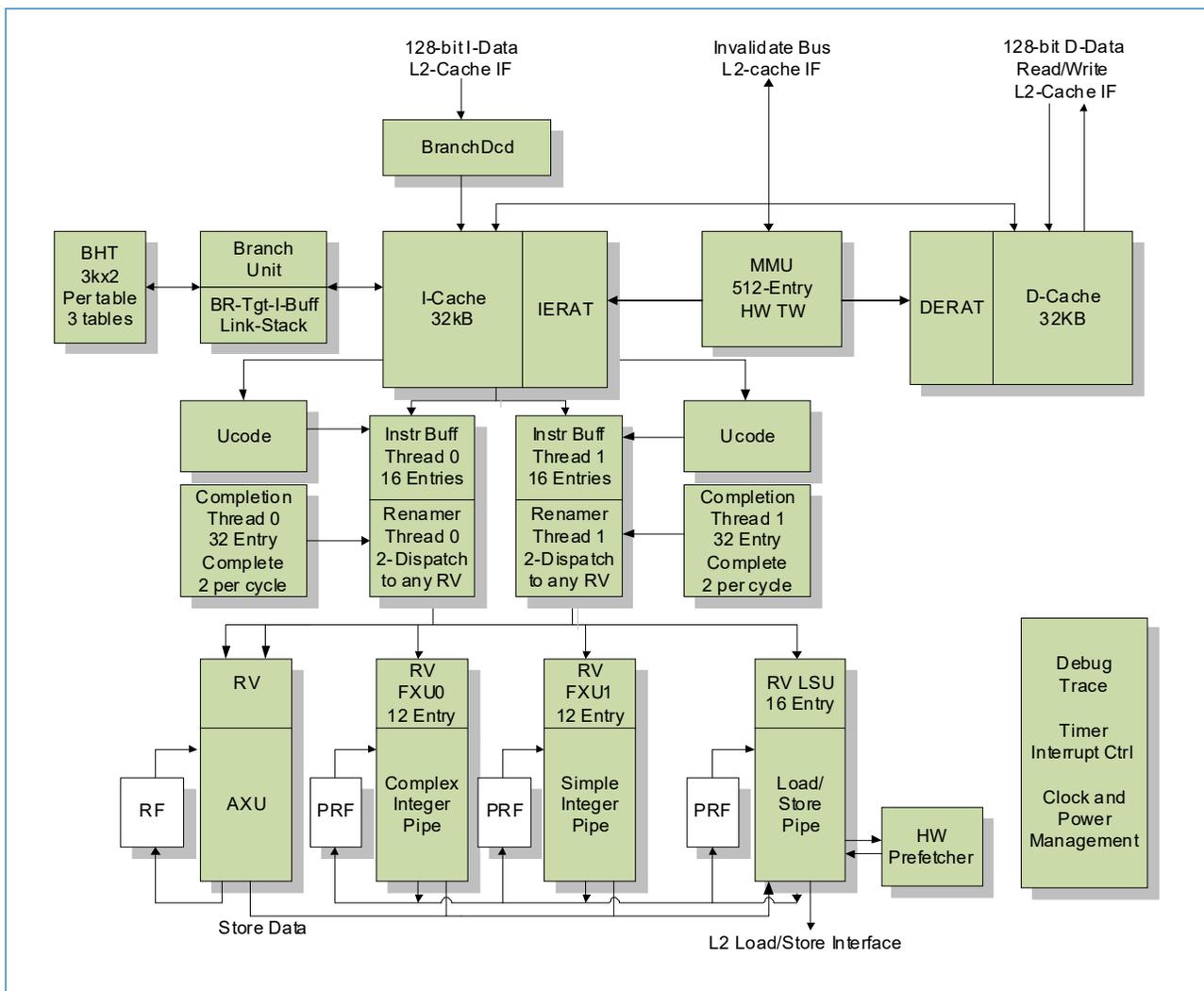
1. The A20 FPU requires software support for IEEE 754 compliance. See *IEEE 754 and Architectural Compliance* on page 54 for details.

- Thirty-two 64-bit Floating Point Registers (FPRs)
- 64-bit load/store interface

1.3 A20 Core Organization

The A20 Core contains an instruction unit and four execution pipes, and other functional elements required by embedded product specifications. These other functions include memory management, cache control, timers, and debug facilities. Interfaces for custom co-processors and floating point functions are provided. The processor L2 interface provides the framework to efficiently support system-on-a-chip (SOC) designs.

Figure 1. A2 Core Organization



1.3.1 Instruction Unit

The instruction unit of the A20 Core includes instruction fetch, instruction decode and dispatch and branch prediction. The instruction unit fetches, decodes, and issues two instructions per cycle to any combination of execution pipelines. Dynamic branch prediction is provided by using a branch history table (BHT). This mechanism greatly improves the branch prediction accuracy and reduces the latency of taken branches, such that the target of a branch can usually be executed immediately after the branch itself, with no penalty.

1.3.2 Execution Unit

The A20 Core contains four execution pipelines - two fixed-point pipes, a load/store pipe and a floating-point pipe.

The fixed-point pipelines handle all arithmetic, logical, branch, and system management instructions, such as interrupt and TLB management, move to/from system registers, etc. The width of the divider is 64 bits. Divide instructions dealing with 64-bit operands recirculate for 65 cycles, and operations with 32 bit-operands recirculate for 32 cycles. No divide instructions are pipelined; they all require some recirculation.

The load/store pipe handles all loads, stores and cache management operations. The pipelined multiply unit can perform 32-bit \times 32-bit multiply operations with single-cycle throughput and single-cycle latency. All misaligned operations are handled in hardware, with no penalty on any operation which is contained within an aligned 32-byte region. The load/store pipeline supports all operations to both big-endian and little-endian data regions.

Appendix D Instruction Execution Performance and Code Optimizations on page 787 provides detailed information on instruction timings and performance implications in the A20 Core.

1.3.3 Instruction and Data Caches

The A20 Core provides separate instruction and data caches, which allows concurrent access and minimizes pipeline stalls. The storage capacity of the cache arrays is 32KB each. Both caches have 64-byte lines, with 4-way set-associativity in the I-cache and 8-way set-associativity in the D-cache. Both caches support parity checking on the tags and data in the memory arrays, to protect against soft errors. If a parity error is detected, the CPU forces an L1 miss and reload from the system bus. or may be configured to cause a machine check exception.

The PowerISA instruction set provides a rich set of cache management instructions for software-enforced coherency. See *Instruction and Data Caches* on page 153 for detailed information about the instruction and data cache controllers.

1.3.3.1 Instruction Cache

The instruction cache delivers up to four instructions per cycle to the instruction unit of the A20 Core. The instruction cache also handles the execution of the PowerISA instruction cache management instructions for coherency.

1.3.3.2 Data Cache

The data cache handles all load and store data accesses, as well as the PowerISA data cache management instructions. All misaligned accesses are handled in hardware, with cacheable load accesses that are contained within a double quadword (32 bytes) handled as a single request, and cacheable store or cache-inhib-

ited load or store accesses that are contained within a quadword (16 bytes) handled as a single request. Load and store accesses which cross these boundaries are broken into separate byte accesses in hardware using the microcode engine.

The data cache interfaces to the AXU port to provide direct load/store access to the data cache for AXU load and store operations. Such AXU load and store instructions can access up to 16 bytes in a single cycle for cacheable or caching inhibited accesses.

The data cache always operates in a write-through manner. It supports cache line locking and “transient” data via way locking.

The data cache provides for up to eight outstanding load misses, and can continue servicing subsequent load and store hits in an out-of-order fashion. Store-gathering is not performed within the A2O core.

1.3.4 Memory Management Unit (MMU)

The A2O Core supports a flat, 42-bit (4TB) real (physical) address space. This 42-bit real address is generated by the MMU, as part of the translation process from the 64-bit effective address, which is calculated by the processor core as an instruction fetch or load/store address. Note: In 32-bit mode, the A2O core forces bits 0:31 of the calculated 64-bit effective address to zeroes. Therefore, to have a translation hit in 32-bit mode, software needs to set the effective address upper bits to zero in the ERATs and TLB.

The MMU provides address translation, access protection, and storage attribute control for embedded applications. The MMU supports demand paged virtual memory and other management schemes that require precise control of logical to physical address mapping and flexible memory protection. Working with appropriate system level software, the MMU provides the following functions:

- Translation of the 88-bit virtual address 1-bit “guest state” (GS), 8-bit logical partition ID (LPID), 1-bit “address space” identifier (AS), 14-bit Process ID (PID), and 64-bit effective address) into the 42-bit real address (note the 1-bit “indirect entry” IND bit is not considered part of the virtual address)
- Page level read, write, and execute access control
- Storage attributes for cache policy, byte order (endianness), and speculative memory access
- Software control of page replacement strategy

The translation lookaside buffer (TLB) is the primary hardware resource involved in the control of translation, protection, and storage attributes. It consists of 512 entries, each specifying the various attributes of a given page of the address space. The TLB is 4-way set associative. The TLB entries may be of type direct (IND=0), in which case the virtual address is translated immediately by a matching entry, or of type indirect (IND=1), in which case the hardware page table walker is invoked to fetch and install an entry from the hardware page table.

The TLB tag and data memory arrays are parity protected against soft errors; if a parity error is detected during an address translation, the TLB and ERAT caches treat the parity error like a miss and proceed to either reload the entry with correct parity in the case of an ERAT miss/TLB hit and set the parity error bit in the appropriate FIR register, or generate a TLB exception ID where software can take appropriate action in the case of a TLB miss.

An operating system may choose to implement hardware page tables in memory that contain virtual to logical translation page table entries (PTEs) per Category E.PT. These PTEs are loaded into the TLB by the hardware page table walker logic after the logical address is converted to a real address via the LRAT per Category E.HV.LRAT. Software must install indirect (IND=1) type TLB entries for each page table that is to be traversed by the hardware walker. Alternately, software can manage the establishment and replacement of

TLB entries by simply not using indirect entries. This gives system software significant flexibility in implementing a custom page replacement strategy. The instruction set provides several instructions for managing TLB entries. These instructions are privileged and the processor must be in supervisor state in order for them to be executed.

The MMU divides the address space into pages. Five direct (IND=0) page sizes (4KB, 64KB, 1MB, 16MB, 1GB) are simultaneously supported, such that at any given time the TLB can contain entries for any combination of page sizes. The MMU also supports two indirect (IND=1) page sizes (1 MB and 256 MB) with associated sub-page sizes (refer to *Section 6.17 Hardware Page Table Walking (Category E.PT)*).

To improve performance, both the instruction cache and the data cache maintain separate “shadow” TLBs called ERATs. The ERATs contain only direct (IND=0) type entries. The instruction I-ERAT contains 16 entries, while the data D-ERAT contains 32 entries. These ERAT arrays minimize TLB contention between instruction fetch and data load/store operations. The instruction fetch and data access mechanisms only access the main unified TLB when a miss occurs in the respective ERAT. Hardware manages the replacement and invalidation of both the I-ERAT and D-ERAT; no system software action is required in MMU mode. In ERAT-only mode, an attempt to access an address for which no ERAT entry exists causes an Instruction or Data TLB Miss exception.

Section 6 Memory Management describes the A2O Core MMU functions in greater detail.

1.3.5 Timers

The A2O Core contains a Time Base and five timers: a Decrementer (DEC), a Guest Decrementer (GDEC), a User Decrementer (UDEC), a Fixed Interval Timer (FIT), and a Watchdog Timer. The Time Base is a 64-bit counter which gets incremented at a frequency either equal to the processor core clock rate or as controlled by a separate asynchronous timer clock input to the core. No interrupt is generated as a result of the Time Base wrapping back to zero.

The DEC is a 32-bit register that is decremented at the same rate at which the Time Base is incremented. The DEC is a hypervisor resource. The DEC register is loaded with a value to create the desired interval. When the register is decremented to zero, a number of actions occur: the DEC stops decrementing, a status bit is set in the Timer Status Register (TSR), and a Decrementer exception is reported to the interrupt mechanism of the A2O Core. Optionally, the DEC can be programmed to reload automatically the value contained in the Decrementer Auto-Reload register (DECAR), after which the DEC resumes decrementing. The Timer Control Register (TCR) contains the interrupt enable for the Decrementer interrupt.

The GDEC and UDEC operate the same as DEC but are accessible to the Guest and User, respectively.

The FIT generates periodic interrupts based on the transition of a selected bit from the Time Base. Users can select one of four intervals for the FIT period by setting a control field in the TCR to select the appropriate bit from the Time Base. When the selected Time Base bit transitions from 0 to 1, a status bit is set in the TSR and a Fixed Interval Timer exception is reported to the interrupt mechanism of the A2O Core. The FIT interrupt enable is contained in the TCR.

Similar to the FIT, the Watchdog Timer also generates a periodic interrupt based on the transition of a selected bit from the Time Base. Users can select one of four intervals for the watchdog period, again by setting a control field in the TCR to select the appropriate bit from the Time Base. Upon the first transition from 0 to 1 of the selected Time Base bit, a status bit is set in the TSR and a Watchdog Timer exception is reported to the interrupt mechanism of the A2O Core. The Watchdog Timer can also be configured to initiate

a hardware reset if a second transition of the selected Time Base bit occurs prior to the first Watchdog exception being serviced. This capability provides an extra measure of recoverability from potential system lock-ups.

The timer functions of the A2O Core are more fully described in *Timer Facilities* on page 371

1.3.6 Debug Facilities

The A2O Core debug facilities include multiple modes for debugging during hardware and software development. Also included are debug events that allow developers to control the debug process. Debug modes and debug events are controlled using debug registers in the chip. The debug registers are accessed either through software running on the processor, or through the SCOM port.

The debug modes, events, controls, and interfaces provide a powerful combination of debug facilities for hardware development tools, such as the RISCWatch™ debugger from IBM.

A brief overview of the debug modes and development tool support are provided below. *Debug Facilities* on page 389 provides detailed information about each debug mode and other debug resources.

1.3.6.1 Debug Modes

The A2O Core supports two debug modes: internal and external. Each mode supports a different type of debug tool used in embedded systems development. Internal debug mode supports software-based ROM monitors, and external debug mode supports a hardware emulator type of debug. The debug modes are controlled by Debug Control Register 0 (DBCRO) and the setting of bits in the Machine State Register (MSR).

Internal debug mode supports accessing architected processor resources, setting hardware and software breakpoints, and monitoring processor status. In internal debug mode, debug events can generate debug exceptions, which can interrupt normal program flow so that monitor software can collect processor status and alter processor resources.

Internal debug mode relies on exception-handling software—running on the processor—along with an external communications path to debug software problems. This mode is used while the processor continues executing instructions and enables debugging of problems in application or operating system code. Access to debugger software executing in the processor while in internal debug mode is through a communications port on the processor board, such as a serial port or ethernet connection.

External debug mode supports stopping, starting, and single-stepping the processor, accessing architected processor resources, setting hardware and software breakpoints, and monitoring processor status. In external debug mode, debug events can architecturally “freeze” the processor. While the processor is frozen, normal instruction execution stops, and the architected processor resources can be accessed and altered using a debug tool (such as RISCWatch) attached through the SCOM port. This mode is useful for debugging hardware and low-level control software problems.

1.3.6.2 Development Tool Support

The A2O Core provides powerful debug support for a wide range of hardware and software development tools.

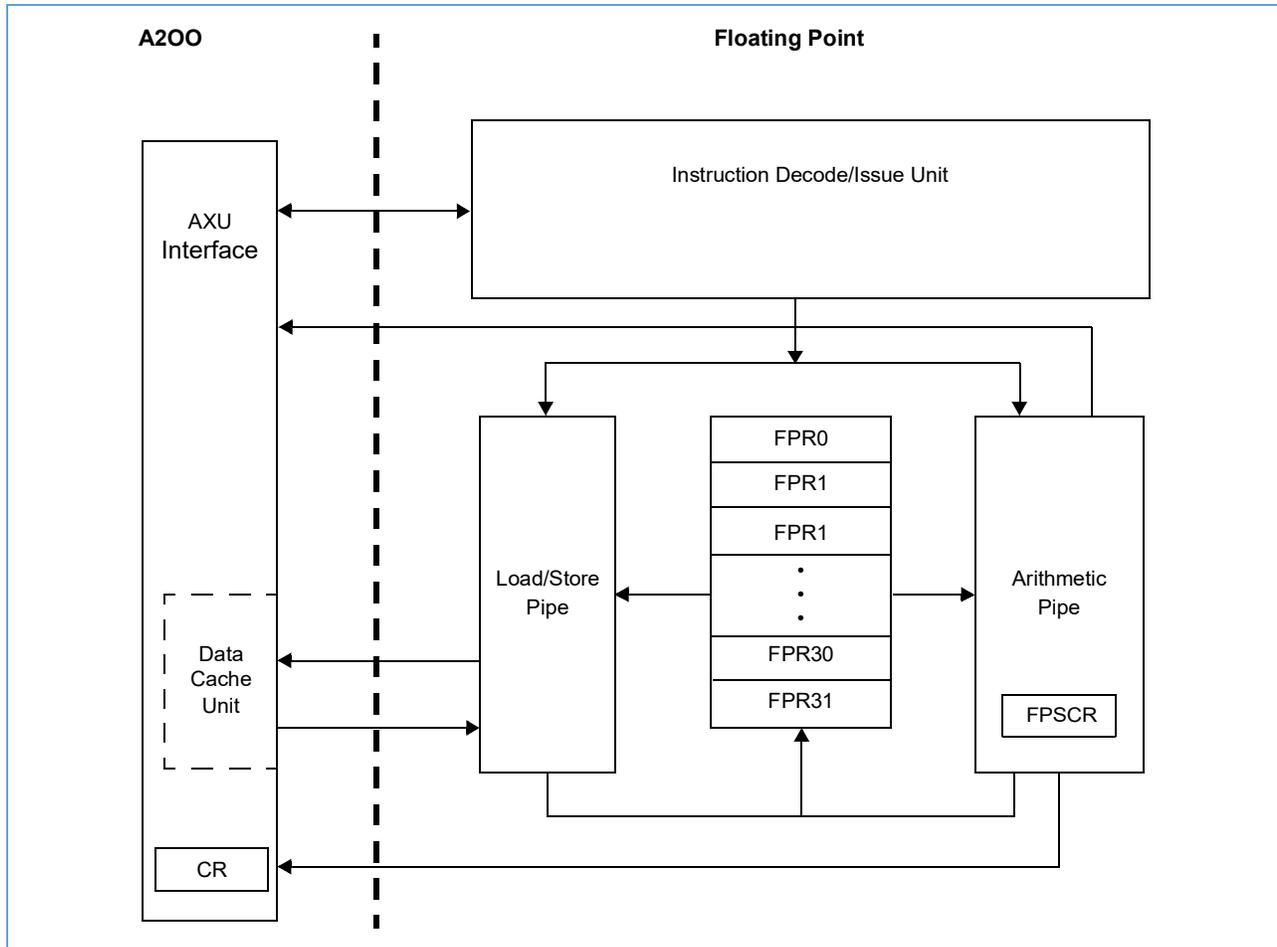
RISCWatch is an example of a development tool that uses the external debug mode, debug events, and the SCOM port to support hardware and software development and debugging.

1.3.7 Floating-Point Unit Organization

The floating-point unit incorporates a single-issue instruction decode and issue unit and a 6-stage arithmetic pipeline working in parallel with a 4-stage load/store pipeline. The floating-point unit contains a Floating-Point Register (FPR) file that interfaces to both pipelines. There are thirty-two 64-bit FPRs.

Figure 1-2 illustrates the logical organization of the A20 Core and its relationship to the A20 processor core.

Figure 2. A20 Processor Block Diagram



1.3.7.1 Arithmetic and Load/Store Pipelines

The A20 Core has a single execution pipeline. The pipeline handles all computational instructions and reads from and writes to the FPRs, Floating-Point Status and Control Register (FPSCR), and the Condition Register (CR).

1.3.8 IEEE 754 and Architectural Compliance

The A20 Core is IEEE 754 and Power ISA compliant and implements single-precision and double-precision instructions.

1.3.8.1 IEEE 754 Compliance

IEEE 754 requires a certain set of operations to be included in any implementation that claims to be compliant. Such operations can be implemented in hardware, software, or a combination of the two. The Power ISA floating-point architecture includes most of the required operations but some are missing. The missing operations are: floating-point remainder, format conversion between binary and decimal, and format conversion from integer to floating-point. It is necessary to provide a software library to support these missing functions. In other words, the Power ISA Architecture requires software support to be fully compliant with the IEEE standard.

1.3.9 Floating-Point Unit Implementation

Certain aspects of the behavior of the floating-point unit are implementation-specific.

1.3.9.1 Reciprocal Estimates

While the Power ISA Architecture defines single-precision reciprocal estimates and reciprocal square root estimates to have relative errors of 2^{-5} and 2^{-8} respectively, both are implemented in the A2O Core to have a relative error of 2^{-14} .

Programmers are encouraged to take advantage of this increased accuracy, but must be aware that code that relies on this increased accuracy might not work on any other Power ISA FU.

1.3.9.2 Denormalized B Operands

The floating-point unit supports all denormal numbers in the dataflow with no additional latency *except* the following cases:

1. B is a double-precision denorm AND NOT (move{fabs/fnabs/fneg} OR fsel OR fcfid OR mv_to_fpscr).
2. B is a single-precision denorm AND NOT (move{fabs/fnabs/fneg} OR fsel)

If any of the above cases are detected, the A2 core flushes to the microcode engine, which in turn issues a prenormalization instruction, followed by the original instruction. The latency for these operations increases by 20 cycles when this occurs.

1.3.9.3 Non-IEEE mode

Non-IEEE mode, controlled by the NI bit in the FPSCR, is intended to eliminate data-dependent overhead cycles caused by exceptional operands or results. The result is faster, deterministic performance with reasonable results. This mode is not supported by the A2 core. The value of the NI bit is ignored.

1.3.10 Floating-Point Unit Interfaces

The floating-point unit interfaces to the A2O processor core.

1.3.10.1 A2O Processor Core Interface

This interface enables the A2O Core to interact with the A2O processor core. Interactions include resets and updating the CR.

1.3.10.2 Clock and Power Management Interface

The CPM interface supports clock distribution and power management to reduce power consumption below the normal operational level. External logic is necessary for the sleep mode to function.

1.4 Core Interfaces

The core includes the following interfaces:

- System interface
- Auxiliary execution unit (AXU) port
- SCOM, debug, trace, and performance monitor event ports
- Interrupt interface
- Clock and power management interface

Several of these interfaces are described briefly in the sections below.

1.4.1 System Interface

The A2O Core Interface has one command interface for instruction reads, data reads, and data writes, and uses a 42-bit address bus. A full 64 byte cacheline is implied for cacheable data reads and cacheable instruction fetches. The Transfer length is used to indicate 1 byte, 2 byte, 4 byte, 8 byte, 16 byte and 32 byte for non-cacheable reads and 16 bytes for non-cacheable instruction fetches. Data writes can be 1 byte, 2 byte, 4 byte, 8 byte, or 16 byte for non-cacheable or cacheable writes. There is a 128-bit data reload interface for instruction reads and data reads. When the reload data is less than 16 bytes the data should be aligned within the 16 byte reload bus based on the associated command interface address. There is a back invalidate interface for systems with an entity outside the A2O core (such as an L2 cache controller) that provide hardware cache coherency.

The command interface is a credit-based interface. The A2O core can handle up to 8 load-type credits. The actual number of load-type credits (L) it will handle is initialized in the A2O Core Config Ring. The 8 load type credits are shared between A2O a 8 Entry Load Miss Queue, 2 instruction fetches and 1 MMU load. An entity outside the A2O core is expected to be have a near queue of L entries for load-type operations. The specific command is indicated in the Transaction Type.

Examples of Transaction Types that expect data to be returned on the reload bus are instruction fetch, load, and dcbt. Examples of Transaction Types that do not expect data to be returned on the reload bus are store, dcbz, and dcbf. The A2O core can handle up to 32 store-type credits. The actual number of credits (S) it will handle is initialized in the A2O Core Config Ring.

An entity outside the A2O core is expected to be able to queue the S store-type operations and give a pop indication to the A2O Core for each as it is processed and the queue entry is available. For an entity outside the A2O core that also supports store gathering, it should give a gather indication to the A2O Core when the store is gathered with an existing queue entry to let the A2O Core know that an additional queue entry is available.

1.4.2 Auxiliary Execution Unit (AXU) Port

This interface provides the A2O Core with the flexibility to attach a tightly-coupled coprocessor macro incorporating instructions that extend those provided within the processor core itself. The AXU port provides sufficient functionality for attachment of various coprocessor functions such as a fully-compliant Power ISA floating-point unit (single- or double-precision), multimedia engine, DSP, or other custom function implementing algorithms appropriate for specific system applications. The AXU interface supports can be used with macros that contain their own register files. AXU load and store instructions can directly access the A2O Core data cache, with operands of up to a double quadword (16 bytes) in length.

The AXU interface provides the capability for a coprocessor to execute instructions that are not part of the Power ISA instruction set at the same time that the A2 core is executing PowerISA instructions. Areas within the architected instruction space allow for these customer-specific or application-specific AXU instruction set extensions. Further description is beyond the scope of this document.

1.4.3 JTAG Port

The A2O Core SCOM port supports the indirect attachment of a debug tool such as the RISCWatch product from IBM. A logic block outside the A2 core must provide JTAG to SCOM port translation. Through the SCOM port, and using the debug facilities designed into the A2O Core, a debug workstation can single-step the processor and interrogate the internal processor state to facilitate hardware and software debugging.



2. CPU Programming Model

The programming model of the A2O Core describes how the following features and operations of the core appear to programmers:

- *Logical Partitioning* on page 59
- *Storage Addressing* on page 60
- *Multithreading* on page 68
- *Registers* on page 79
- *32-Bit Mode* on page 82
- *Instruction Categories* on page 82
- *Instruction Classes* on page 84
- *Implemented Instruction Set Summary* on page 85
 - *Wait Instruction* on page 95
- *Branch Processing* on page 96
- *Integer Processing* on page 106
- *Processor Control* on page 109
- *Privileged Modes* on page 116
- *Speculative Accesses* on page 118
- *Synchronization* on page 118
- *Software Transactional Memory Acceleration* on page 121

2.1 Logical Partitioning

2.1.1 Overview

Logical partitioning defines instructions, resources, and methods for establishing an additional attribute of processor privilege called a guest state.

The Embedded.Hypervisor category permits processors and portions of real storage to be assigned to local collections called partitions such that a program executing on a processor in one partition cannot interfere with any program executing on a processor in a different partition. This isolation can be provided for both problem state and privileged state programs by using a layer of trusted software called a hypervisor program (or simply a “hypervisor”) and the resources provided by this category to manage system resources. The collection of software that runs in a given partition and its associated resources is called a guest. The guest normally includes an operating system (or other system software) running in privileged state and its associated processes running in the problem state under the management of the hypervisor. The processor is in the guest state when a guest is executing, and it is in the hypervisor state when the hypervisor is executing. The processor is executing in the guest state when $MSR[GS] = 1$.

A2 implements 2^8 partitions. See *Section 6.18.2 Logical Partition ID Register (LPIDR)* on page 230. All threads of a single A2 core must be assigned to the same logical partition.

A processor is assigned to one partition at any given time. A processor can be assigned to any given partition without consideration of the physical configuration of the system (for example, shared registers, caches, organization of the storage hierarchy), except that processors that share certain hypervisor resources might need to be assigned to the same partition. Additionally, certain resources can be used by the guest at the discretion of the hypervisor. Such usage might cause interference between partitions, and the hypervisor should allocate those resources accordingly. The primary registers and facilities used to control logical partitioning are described in the following subsections. Other facilities associated with logical partitioning are described within the appropriate sections within this book.

Category Embedded. Hypervisor changes the operating system programming model to allow for easier virtualization, while retaining a default backwards compatible mode where an operating system written for processors not containing this category will still operate as before without using the logical partitioning facilities.

2.2 Storage Addressing

As a 64-bit implementation of the Power ISA Architecture, the A2O Core implements a uniform 64-bit effective address (EA) space. Effective addresses are expanded into virtual addresses and then translated to 42-bit (4 TB) real addresses by the memory management unit (see *Memory Management* on page 169 for more information about the translation process). The organization of the real address space into a physical address space is system-dependent, and is described in the user's manuals for chip-level products that incorporate an A2O Core.

The A2O Core generates an effective address whenever it executes a storage access, branch, cache management, or translation look aside buffer (TLB) management instruction, or when it fetches the next sequential instruction.

2.2.1 Storage Operands

Bytes in storage are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Data storage operands accessed by the integer load/store instructions can be bytes, halfwords, words, doublewords or—for load/store multiple and string instructions—a sequence of words or bytes, respectively. Data storage operands accessed by auxiliary execution unit (AXU) load/store instructions can be bytes, halfwords, words, doublewords, quadwords or double quadwords. The address of a storage operand is the address of its first byte (that is, of its lowest-numbered byte). Byte ordering can be either big endian or little endian, as controlled by the endian storage attribute (see *Byte Ordering* on page 64; also see *Endian (E)* on page 181 for more information about the endian storage attribute).

Operand length is implicit for each scalar storage access instruction type (that is, each storage access instruction type other than the load/store multiple and string instructions). The operand of such a scalar storage 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 storage operand is said to be *aligned* if it is aligned at its natural boundary; otherwise, it is said to be *unaligned*.

Data storage operands for storage access instructions have the characteristics shown in *Table 2-1* on page 61.

Table 1. Data Operand Definitions

Storage Access Instruction Type	Operand Length	Addr[59:63] if Aligned
Byte (or String)	8 bits	0bxxxxx
Halfword	2 bytes	0bxxxx0
Word (or Multiple)	4 bytes	0bxxx00
Doubleword	8 bytes	0bxx000
Quadword (AXU only)	16 bytes	0bx0000

Note: 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.

The alignment of the operand effective address of some storage access instructions might affect performance; in some cases, it might cause an alignment exception to occur. For such storage access instructions, the best performance is obtained when the storage operands are aligned. *Table 2-2* summarizes the effects of alignment on those storage access instruction types for which such effects exist. If an instruction type is not shown in the table, there are no alignment effects for that instruction type.

Table 2. Alignment Effects for Storage Access Instructions (Sheet 1 of 2)

Storage Access Instruction Type	Alignment Effects
Integer cacheable load halfword	Broken into byte accesses if crosses 32-byte boundary (EA[59:63] = 0b11111); otherwise no effect. (See notes.)
Integer cacheable store or caching inhibited load/store halfword	Broken into byte accesses if crosses 16-byte boundary (EA[60:63] = 0b1111); otherwise no effect. (See notes.)
Integer cacheable load word	Broken into byte accesses if crosses 32-byte boundary (EA[59:63] > 0b11100); otherwise no effect. (See notes.)
Integer cacheable store or caching inhibited load/store word	Broken into byte accesses if crosses 16-byte boundary (EA[60:63] > 0b1100); otherwise no effect. (See notes.)
Integer cacheable load doubleword	Broken into byte accesses if crosses 32-byte boundary (EA[59:63] > 0b11000); otherwise no effect. (See notes.)
Integer cacheable store or caching inhibited load/store doubleword	Broken into byte accesses if crosses 16-byte boundary (EA[60:63] > 0b1000); otherwise no effect. (See notes.)
Integer load/store multiple	Broken into a series of word (4-byte) accesses until the last word is accessed. The load/store multiple address must be word aligned. (See notes.)
Integer load/store string	Broken into a series of byte accesses until the last byte is accessed. (See notes.)
AXU cacheable load halfword	Broken into byte accesses if crosses 32-byte boundary (EA[59:63] = 0b11111); otherwise no effect. (See notes.)
AXU cacheable store or caching inhibited load/store halfword	Broken into byte accesses if crosses 16-byte boundary (EA[60:63] = 0b1111); otherwise no effect. (See notes.)
AXU cacheable load word	Broken into byte accesses if crosses 32-byte boundary (EA[59:63] > 0b11100); otherwise no effect. (See notes.)
AXU cacheable store or caching inhibited load/store word	Broken into byte accesses if crosses 16-byte boundary (EA[60:63] > 0b1100); otherwise no effect. (See notes.)
AXU cacheable load doubleword	Broken into byte accesses if crosses 32-byte boundary (EA[59:63] > 0b11000); otherwise no effect. (See notes.)
AXU cacheable store or caching inhibited load/store doubleword	Broken into byte accesses if crosses 16-byte boundary (EA[60:63] > 0b1000); otherwise no effect. (See notes.)

Table 2. Alignment Effects for Storage Access Instructions (Sheet 2 of 2)

Storage Access Instruction Type	Alignment Effects
AXU cacheable load quadword	Broken into byte accesses if crosses 32-byte boundary (EA[59:63] > 0b10000); otherwise no effect. (See notes.)
AXU cacheable store or caching inhibited load/store quadword	Broken into byte accesses if crosses 16-byte boundary (EA[60:63] > 0b0000); otherwise no effect. (See notes.)
Notes:	
<ul style="list-style-type: none"> Any unaligned access that also crosses a 4 K page boundary causes an alignment exception. An auxiliary processor can specify that the EA for a given AXU load/store instruction must be aligned at the operand-size boundary or, alternatively, at a word boundary. If the AXU so indicates this requirement and the calculated EA fails to meet it, the A2O Core generates an alignment exception. Alternatively, an auxiliary processor can specify that the EA for a given AXU load/store instruction should be "forced" to be aligned by ignoring the appropriate number of low-order EA bits and processing the AXU load/store as if those bits were 0. Byte, halfword, word, doubleword, and quadword AXU load/store instructions ignore 0, 1, 2, 3, and 4 low-order EA bits, respectively. 	

Cache management instructions access *cache block* operands; for the A2O Core, the cache block size is 64 bytes. However, the effective addresses calculated by cache management instructions are not required to be aligned on cache block boundaries. Instead, the architecture specifies that the associated low-order effective address bits (bits 58:63 for the A2O Core) are ignored during the execution of these instructions.

Similarly, the TLB management instructions access *page* operands, and—as determined by the page size—the associated low-order effective address bits are ignored during the execution of these instructions.

Instruction storage operands, on the other hand, are always 4 bytes long, and the effective addresses calculated by branch instructions are therefore always word-aligned.

2.2.2 Effective Address Calculation

For a storage access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address of $2^{64}-1$ for 64-bit mode or $2^{32}-1$ in 32-bit mode (that is, the storage operand itself crosses the maximum address boundary), the result of the operation is undefined, as specified by the architecture. The A2O Core performs the operation as if the storage operand wrapped around from the maximum effective address to effective address 0. Software, however, should not depend upon this behavior, so that it can be ported to other implementations that do not handle this scenario in the same fashion. Accordingly, software should ensure that no data storage operands cross the maximum address boundary.

Note: Because instructions are words and because the effective addresses of instructions are always implicitly on word boundaries, it is not possible for an instruction storage operand to cross any word boundary, including the maximum address boundary.

Effective address arithmetic, which calculates the starting address for storage operands, wraps around from the maximum address to address 0 for all effective address computations except next sequential instruction fetching. See *Instruction Storage Addressing Modes* on page 63 for more information about next sequential instruction fetching at the maximum address boundary.

2.2.2.1 Data Storage Addressing Modes

There are two data storage addressing modes supported by the A2O Core:

- Base + displacement (D-mode) addressing mode:



The 16-bit D field is sign-extended and added to the contents of the GPR designated by RA or to zero if RA = 0.

- Base + index (X-mode) addressing mode:

The contents of the GPR designated by RB (or the value 0 for **lswi** and **stswi**) are added to the contents of the GPR designated by RA or to 0 if RA = 0.

2.2.2.2 Instruction Storage Addressing Modes

There are four instruction storage addressing modes supported by the A2O Core:

- I-form branch instructions (unconditional):

The 24-bit LI field is concatenated on the right with 0b00, sign-extended, and then added to either 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 then added to either the address of the branch instruction if AA = 0 or to 0 if AA = 1.

- Taken XL-form branch instructions:

The contents of bits 0:61 of the Link Register (LR) or the Count Register (CTR) are concatenated on the right with 0b00 to form the 64-bit effective address of the next instruction.

Note: In 32-bit mode, the A2 core forces bits 0:31 of the calculated 64-bit effective address to zeros.

- Next sequential instruction fetching (including nontaken branch instructions):

The value 4 is added to the address of the current instruction to form the 64-bit effective address of the next instruction. If the address of the current instruction is 0xFFFF_FFFF_FFFF_FFFC in 64-bit mode or 0x0000_0000_FFFF_FFFC in 32-bit mode, the A2O Core wraps the next sequential instruction address back to address 0. This behavior is not required by the architecture, which specifies that the next sequential instruction address is undefined under these circumstances. Therefore, software should not depend upon this behavior, so that it can be ported to other implementations that do not handle this scenario in the same fashion. Accordingly, if software wants to execute across this maximum address boundary and wrap back to address 0, it should place an unconditional branch at the boundary with a displacement of 4.

In addition to the above four instruction storage addressing modes, the following behavior applies to branch instructions:

- Any branch instruction with LK = 1:

The value 4 is added to the address of the current instruction and the low-order 64 bits of the result are placed into the LR. As for the similar scenario for next sequential instruction fetching, if the address of the branch instruction is 0xFFFF_FFFF_FFFF_FFFC in 64-bit mode or 0x0000_0000_FFFF_FFFC in 32-bit mode, the result placed into the LR is architecturally undefined, although once again the A2O Core wraps the LR update value back to address 0. Again, however, software should not depend on this behavior so that it can be ported to implementations that do not handle this scenario in the same fashion.

2.2.3 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 storage, because nothing can be observed about such order. Only when scalars, which the programmer and processor regard as indivisible quantities, can comprise more than one addressable unit of storage does the question of order arise.

For a machine in which the smallest addressable unit of storage is the 64-bit doubleword, there is no question of the ordering of bytes within doublewords. All transfers of individual scalars between registers and storage are of doublewords, 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.

For the Power ISA Architecture, as for most current computer architectures, the smallest addressable unit of storage is the 8-bit byte. Many scalars are halfwords, words, or doublewords that consist of groups of bytes. When a word-length scalar is moved from a register to storage, the scalar occupies 4 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 8 bits of the scalar, which byte contains the next-highest-order 8 bits, and so on.

Given a scalar that contains multiple bytes, the choice of byte ordering is essentially arbitrary. There are 24 ways to specify the ordering of 4 bytes within a word, but only two of these orderings are sensible:

- The ordering that assigns the lowest address to the highest-order (left-most) 8 bits of the scalar, the next sequential address to the next-highest-order 8 bits, and so on.

This ordering is called *big endian* because the “nd” (most-significant end) of the scalar, considered as a binary number, comes first in storage. IBM RISC System/6000, IBM System/390®, and Motorola 680x0 are examples of computer architectures using this byte ordering.

- The ordering that assigns the lowest address to the lowest-order (“right-most”) 8 bits of the scalar, the next sequential address to the next-lowest-order 8 bits, and so on.

This ordering is called *little endian* because the “little end” (least-significant end) of the scalar, considered as a binary number, comes first in storage. The Intel x86 is an example of a processor architecture using this byte ordering.

Power ISA supports both big-endian and little-endian byte ordering, for both instruction and data storage accesses. Which byte ordering is used is controlled on a memory page basis by the endian (E) storage attribute, which is a field within the TLB entry for the page. The endian storage attribute is set to 0 for a big-endian page and is set to 1 for a little-endian page. See *Memory Management* on page 169 for more information about memory pages, the TLB, and storage attributes, including the endian storage attribute.

2.2.3.1 Structure Mapping Examples

The following C language structure, *s*, contains an assortment of scalars and a character string. The comments show the value assumed to be in each structure element; these values show how the bytes comprising each structure element are mapped into storage.

```
struct {
    int a;           /* 0x1112_1314 word */
    long long b;    /* 0x2122_2324_2526_2728 doubleword */
    int c;          /* 0x3132_3334 word */
    char d[7];      /* 'A','B','C','D','E','F','G' array of bytes */
}
```

```

short e;      /* 0x5152 halfword */
int f;       /* 0x6162_6364 word */
} s;
    
```

C structure mapping rules permit the use of padding (skipped bytes) to align scalars on desirable boundaries. The following structure mapping examples show each scalar aligned at its natural boundary. This alignment introduces padding of 4 bytes between a and b, one byte between d and e, and two bytes between e and f. The same amount of padding is present in both big-endian and little-endian mappings.

Big-Endian Mapping

The big-endian mapping of structure *s* follows (the data is highlighted in the structure mappings). Addresses, in hexadecimal, are below the data stored at the address. The contents of each byte, as defined in structure *s*, is shown as a (hexadecimal) number or character (for the string elements). The shaded cells correspond to padded bytes.

11	12	13	14				
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
21	22	23	24	25	26	27	28
0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
31	32	33	34	'A'	'B'	'C'	'D'
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
'E'	'F'	'G'		51	52		
0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
61	62	63	64				
0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27

Little-Endian Mapping

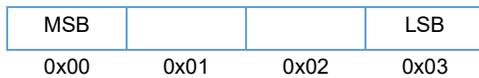
Structure *s* is shown mapped little endian.

14	13	12	11				
0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
28	27	26	25	24	23	22	21
0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
34	33	32	31	'A'	'B'	'C'	'D'
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
'E'	'F'	'G'		52	51		
0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
64	63	62	61				
0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27

2.2.3.2 Instruction Byte Ordering

Power ISA defines instructions as aligned words (4 bytes) in memory. As such, instructions in a big-endian program image are arranged with the most-significant byte (MSB) of the instruction word at the lowest-numbered address.

Consider the big-endian mapping of instruction p at address 0x00, where, for example, $p = \text{add r7, r7, r4}$:



On the other hand, in a little-endian mapping the same instruction is arranged with the least-significant byte (LSB) of the instruction word at the lowest-numbered address:



By the definition of Power ISA bit numbering, the most-significant byte of an instruction is the byte containing bits 0:7 of the instruction. As depicted in the instruction format diagrams (see *Instruction Formats* in the *Power ISA* specification), this most-significant byte is the one that contains the primary opcode field (bits 0:5). Due to this difference in byte orderings, the processor must perform whatever byte reversal is required (depending on the particular byte ordering in use) to correctly deliver the opcode field to the instruction decoder. In the A2O Core, this reversal is performed between the memory interface and the instruction cache, according to the value of the endian storage attribute for each memory page, such that the bytes in the instruction cache are always correctly arranged for delivery directly to the instruction decoder.

If the endian storage attribute for a memory page is reprogrammed from one byte ordering to the other, the contents of the memory page must be reloaded with program and data structures that are in the appropriate byte ordering. Furthermore, anytime the contents of instruction memory change, the instruction cache must be made coherent with the updates by invalidating the instruction cache and refetching the updated memory contents with the new byte ordering.

2.2.3.3 Data Byte Ordering

Unlike instruction fetches, data accesses cannot be byte-reversed between memory and the data cache. Data byte ordering in memory depends upon the data type (byte, halfword, word, and so on) of a specific data item. It is only when moving a data item of a specific type from or to an architected register (as directed by the execution of a particular storage access instruction) that it becomes known what kind of byte reversal might be required due to the byte ordering of the memory page containing the data item. Therefore, byte reversal during load or store accesses is performed between the data cache (or memory, on a data cache miss, for example) and the load register target or store register source, depending on the specific type of load or store instruction (that is, byte, halfword, word, and so on).

Comparing the big-endian and little-endian mappings of structure s , as shown in *Structure Mapping Examples* on page 64, the differences between the byte locations of any data item in the structure depends upon the size of the particular data item. For example (again referring to the big-endian and little-endian mappings of structure s):

- The word a has its 4 bytes reversed within the word spanning addresses 0x00 – 0x03.
- The halfword e has its 2 bytes reversed within the halfword spanning addresses 0x1C – 0x1D.

Note: The array of bytes d , where each data item is a byte, is not reversed when the big-endian and little-endian mappings are compared. For example, the character 'A' is located at address 0x14 in both the big-endian and little-endian mappings.

The size of the data item being loaded or stored must be known before the processor can decide whether, and if so, how, to reorder the bytes when moving them between a register and the data cache (or memory).

- For byte loads and stores, including strings, no reordering of bytes occurs regardless of byte ordering.
- For halfword loads and stores, bytes are reversed within the halfword for one byte order with respect to the other.
- For word loads and stores (including load/store multiple), bytes are reversed within the word for one byte order with respect to the other.
- For doubleword loads and stores, bytes are reversed within the doubleword for one byte order with respect to the other.
- For quadword loads and stores (AXU loads/stores only), bytes are reversed within the quadword for one byte order with respect to the other.

Note: This mechanism applies independent of the alignment of data. In other words, when loading a multi-byte data operand with a scalar load instruction, bytes are accessed from the data cache (or memory) starting with the byte at the calculated effective address and continuing with consecutively higher-numbered bytes until the required number of bytes have been retrieved. Then, the bytes are arranged such that either the byte from the highest-numbered address (for big-endian storage regions) or the lowest-numbered address (for little-endian storage regions) is placed into the least-significant byte of the register. The rest of the register is filled in corresponding order with the rest of the accessed bytes. An analogous procedure is followed for scalar store instructions.

For load/store multiple instructions, each group of 4 bytes is transferred between memory and the register according to the procedure for a scalar load word instruction.

For load/store string instructions, the most-significant byte of the first register is transferred to or from memory at the starting (lowest-numbered) effective address, regardless of byte ordering. Subsequent register bytes (from most-significant to least-significant, and then moving into the next register, starting with the most-significant byte, and so on) are transferred to or from memory at sequentially higher-numbered addresses. This behavior for byte strings ensures that if two strings are loaded into registers and then compared, the first bytes of the strings are treated as most significant with respect to the comparison.

2.2.3.4 Byte-Reverse Instructions

The Power ISA defines load/store byte-reverse instructions, which can access storage that is specified as being of one byte ordering in the same manner that a regular (that is, nonbyte-reverse) load/store instruction would access storage that is specified as being of the opposite byte ordering. In other words, a load/store byte-reverse instruction to a big-endian memory page transfers data between the data cache (or memory) and the register in the same manner that a normal load/store would transfer the data to or from a little-endian memory page. Similarly, a load/store byte-reverse instruction to a little-endian memory page transfers data between the data cache (or memory) and the register in the same manner that a normal load/store would transfer the data to or from a big-endian memory page.

The function of the load/store byte-reverse instructions is useful when a particular memory page contains a combination of data with both big-endian and little-endian byte ordering. In such an environment, the endian storage attribute for the memory page would be set according to the predominant byte ordering for the page,

and the normal load/store instructions would be used to access data operands that used this predominant byte ordering. Conversely, the load/store byte-reverse instructions would be used to access the data operands that were of the other (less prevalent) byte ordering.

Software compilers cannot typically make general use of the load/store byte-reverse instructions, so they are ordinarily used only in special, hand-coded device drivers.

2.3 Multithreading

The A2 core has two threads that allow simultaneous execution within the processor and can be viewed as a 2-way multiprocessor with shared dataflow. This gives the effective appearance of two independent processing units from the view of software. The performance of each thread can be limited due to the sharing of resources between each of the threads.

2.3.1 Thread Identification

2.3.1.1 Thread Identification Register (TIR)

The TIR is a read-only register that can be used to distinguish a thread from other threads on the A2 core. The TIR returns a value n , where n is referred to as “thread n .”

Register Short Name:	TIR	Read Access:	Hypv
Decimal SPR Number:	446	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:61	///	0x0	<u>Reserved</u>
62:63	TID	0b00	<u>Processor Thread ID</u> This field can be used to distinguish the thread from other threads on the processor. Threads are numbered sequentially, with valid values ranging from 0 to 3.

2.3.1.2 Processor Identification Register (PIR)

The PIR is a read-only register that uniquely identifies a specific instance of a processor thread, within a multiprocessor configuration, enabling software to determine exactly which thread it is running on. This capability is important for operating system software within multiprocessor configurations.

Register Short Name:	PIR	Read Access:	Priv
Decimal SPR Number:	286	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GPIR	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:53	///	0x0	<u>Reserved</u>
54:61	CID ¹⁰	0x0	<u>Processor Core ID</u> ¹⁰ Returns the value of the IO pin <code>an_ac_coreid</code> . This can be used to distinguish a processor core from other processor cores in the system
62:63	TID	0b00	<u>Processor Thread ID</u> This field can be used to distinguish the thread from other threads on the processor. Threads are numbered sequentially, with valid values ranging from 0 to 3.

2.3.1.3 Guest Processor Identification Register (GPIR)

The GPIR is a register that identifies a specific instance of a processor thread for the guest operating system. The GPIR is used to filter incoming processor messages. See *Processor Messages* on page 341.

Register Short Name:	GPIR	Read Access:	Priv
Decimal SPR Number:	382	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:49	VPTAG	0x0	<u>Virtual Processor Tag</u> Storage used by the guest OS to identify the virtual processor on which the OS is running
50:63	DBTAG	0x0	<u>Doorbell Tag</u> Used to match guest doorbell messages that are sent to all the processors and virtual processors in a coherence domain. If a sent guest doorbell message tag matches the DBTAG field, a guest doorbell is said to be accepted on the [virtual] processor.

2.3.2 Thread Run State

The A2 core provides several methods for controlling a thread's run state. For a thread to fetch instructions, all methods outlined below must be properly configured. If any one I/O or register is configured to stop a thread, the affected thread will not fetch instructions.

2.3.2.1 Thread Stop I/O Pin

The I/O pin, `an_ac_pm_thread_stop`, can be used to stop the A2 core from fetching instructions. Stopping a thread causes all instructions that have begun executing to be completed and all prefetched instructions to be discarded.

2.3.2.2 Thread Control and Status Register (THRCTL)

The SCOM accessible THRCTL register can control the thread run state to allow an external debugger control of the processor. See *Direct Access to I-Cache and D-Cache Directories* on page 432. Stopping a thread via THRCTRL causes all instructions that have begun executing to be completed and all prefetched instructions to be discarded.

2.3.2.3 Core Configuration Register 0 (CCR0)

The CCR0 is used to disable or enable threads. When a thread is disabled by setting the CCR0 bit corresponding to the thread to 0, all instructions that have begun executing are completed and all prefetched instructions are discarded. Subsequent instructions are not prefetched or initiated. Asynchronous interrupts or other conditions that are unmasked and enabled in CCR1 for the thread will cause the thread to be re-enabled. Executing a **wait** instruction on a thread will cause that thread's CCR0[WE] to be set to 1. CCR0 also contains controls for allowing the processor to enter a power managed state. See *Section 13 Power Management Methods* on page 517 for information about power savings modes.

Note: When using **mtccr0** to put other threads to sleep, using an external interrupt or any asynchronous interrupt as the wake-up method is not reliable. The thread being put to sleep might have just taken an interrupt and MSR(EE) is zero, preventing wake-up. In this case, **mtccr0** should be used to wake up the sleeping threads. A thread can put itself to sleep using **mtccr0** or the **wait** instruction and wake up using an external interrupt or any asynchronous interrupt reliably.

Register Short Name:	CCR0	Read Access:	Hypv
Decimal SPR Number:	1008	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	bcfg

Bit(s):	Field Name:	Init	Description
32:33	PME	0b00	<u>Power Management Enable</u> 00 Disabled: No power savings mode entered. 01 PM_Sleep_enable: PM_Sleep state entered when all threads are stopped. 10 PM_RVW_enable: PM_RVW state entered when all threads are stopped. 11 Disabled2: No power savings mode entered. NOTE: Refer to the A2 User Manual, Power Management Methods section.
34:51	///	0x0	<u>Reserved</u>
52:55	WEM	0b0000	<u>Wait Enable Mask</u> 0 No effect to CCR0[WE] 1 Allows writing of corresponding bit in CCR0[WE] field. These bits are non-persistent. A read always returns zeros.
56:59	///	0b0000	<u>Reserved</u>
60:63	WE	0b0000	<u>Wait Enable</u> For $t < 4$, bit 63-t corresponds to thread t: 0 Indicates the thread is enabled 1 Indicates the thread is disabled Note: This field may also be set by a 'wait' instruction

2.3.2.4 Thread Enable Register (TENS, TENC)

The Thread Enable Register is used to disable or enable threads and is provided as a means to access shared resources (see *Accessing Shared Resources* on page 75). When a thread is disabled by setting the TEN bit corresponding to the thread 0, all instructions that have begun executing are completed and all prefetched instructions are discarded. Subsequent instructions are not prefetched or initiated. All asynchronous interrupts for the thread are delayed until the thread is re-enabled.

The TEN is accessed by using two registers: TENS and TENC. When TENS is written, threads for which the corresponding bit in TENS is 1 are enabled; threads for which the corresponding bit in TENS is 0 are unaffected. When TENC is written, threads for which the corresponding bit in TENC is 1 are disabled; threads for which the corresponding bit in TENC is 0 are unaffected. When either SPR is read, the current value of the TEN is returned.

Register Short Name:	TENS	Read Access:	Hypv
Decimal SPR Number:	438	Write Access:	Hypv
Initial Value:	0x0000000000000001	Duplicated for MT:	N
Slow SPR:	N	Notes:	WS
Guest Supervisor Mapping:		Scan Ring:	bcfg

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:59	///	0x0	<u>Reserved</u>
60:63	TEN	0b0001	<u>Thread Enable Set</u> For $t < 4$, bit 63-t corresponds to thread t. When bit 63-t is set to 1, thread t is enabled, if it is not already. When bit 63-t is set 0, thread t is unaffected. When bit 63-t is read, the current value of the thread enable is returned.

Register Short Name:	TENC	Read Access:	Hypv
Decimal SPR Number:	439	Write Access:	Hypv
Initial Value:	0x0000000000000001	Duplicated for MT:	N
Slow SPR:	N	Notes:	WC
Guest Supervisor Mapping:		Scan Ring:	bcfg

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:59	///	0x0	<u>Reserved</u>
60:63	TEN	0b0001	<u>Thread Enable Clear</u> For $t < 4$, bit 63-t corresponds to thread t. When bit 63-t is set to 1, thread t is disabled, if it is not already. When bit 63-t is set 0, thread t is unaffected. When bit 63-t is read, the current value of the thread enable is returned.

2.3.2.5 Thread Enable Status Register (TENSr)

The TENSr indicates which threads are quiesced.

Note: The TENSr is only valid after a context synchronizing instruction or an event that precisely stops a thread, such as a write to TEN.

Note: When thread T1 disables other threads, Tn, it sets the 10 bits corresponding to Tn to zeros. To ensure that all operations being performed by threads Tn have been performed with respect to all threads on the processor, thread T1 reads the TENSr until all the bits corresponding to the disabled threads, Tn, are zeros.

Register Short Name:	TENSr	Read Access:	Hypv
-----------------------------	-------	---------------------	------

Decimal SPR Number:	437	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:59	///	0x0	<u>Reserved</u>
60:63	TENSR	0b0000	<u>Thread Enable Status Register</u> Bit 63-t of the TENSr corresponds to thread t.

2.3.3 Wake On Interrupt

The A2 core can be configured to wake on interrupts or other conditions, if the thread was disabled by a write to CCR0 or by executing a **wait** instruction.

2.3.3.1 Core Configuration Register 1 (CCR1)

CCR1 provides additional masking on what conditions can cause the processor to resume execution. The conditions or interrupts specified must be appropriately unmasked and must also be enabled in CCR1 to exit the stopped state.

Register Short Name:	CCR1	Read Access:	Hypv
Decimal SPR Number:	1009	Write Access:	Hypv
Initial Value:	0x00000000f0f0f0f	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	///	0b00	<u>Reserved</u>
34:39	WC3	0xf	<u>Thread 3 Wake Control</u> (0) 1 Disables sleep on waitrsv (1) 1 Disables sleep on waitimpl (2) 1 Enables wake on Critical Input, Watchdog, Critical Doorbell, Guest Critical Doorbell, or Guest Machine Check Doorbell Interrupts (3) 1 Enables wake on External Input, Performance Monitor, Doorbell, or Guest Doorbell Interrupts (4) 1 Enables wake on Decrementer or User Decrementer Interrupts (5) 1 Enables wake on Fixed Interval Timer Interrupts
40:41	///	0b00	<u>Reserved</u>

Bit(s):	Field Name:	Init	Description
42:47	WC2	0xf	<u>Thread 2 Wake Control</u> (0) 1 Disables sleep on waitrsv (1) 1 Disables sleep on waitimpl (2) 1 Enables wake on Critical Input, Watchdog, Critical Doorbell, Guest Critical Doorbell, or Guest Machine Check Doorbell Interrupts (3) 1 Enables wake on External Input, Performance Monitor, Doorbell, or Guest Doorbell Interrupts (4) 1 Enables wake on Decrementer or User Decrementer Interrupts (5) 1 Enables wake on Fixed Interval Timer Interrupts
48:49	///	0b00	<u>Reserved</u>
50:55	WC1	0xf	<u>Thread 1 Wake Control</u> (0) 1 Disables sleep on waitrsv (1) 1 Disables sleep on waitimpl (2) 1 Enables wake on Critical Input, Watchdog, Critical Doorbell, Guest Critical Doorbell, or Guest Machine Check Doorbell Interrupts (3) 1 Enables wake on External Input, Performance Monitor, Doorbell, or Guest Doorbell Interrupts (4) 1 Enables wake on Decrementer or User Decrementer Interrupts (5) 1 Enables wake on Fixed Interval Timer Interrupts
56:57	///	0b00	<u>Reserved</u>
58:63	WC0	0xf	<u>Thread 0 Wake Control</u> (0) 1 Disables sleep on waitrsv (1) 1 Disables sleep on waitimpl (2) 1 Enables wake on Critical Input, Watchdog, Critical Doorbell, Guest Critical Doorbell, or Guest Machine Check Doorbell Interrupts (3) 1 Enables wake on External Input, Performance Monitor, Doorbell, or Guest Doorbell Interrupts (4) 1 Enables wake on Decrementer or User Decrementer Interrupts (5) 1 Enables wake on Fixed Interval Timer Interrupts

2.3.4 Thread Priority

Thread priority can be changed by writing the PPR32 register, executing an **or Rx,Rx,Rx** instruction, or by causing an interrupt.

2.3.4.1 Program Priority Register (PPR32)

The program priority register controls thread priority. A2 hardware supports three physical priorities. In A2's lowest hardware priority, the number of cycles between two instruction groups being dispatched is determined by IUCR1[THRES]. See *Instruction Unit Configuration Register 1 (IUCR1)* on page 74.

The mapping of the three hardware priorities to the architected priorities in the PPR32 register is shown in *Table 2-3*. An **or Rx,Rx,Rx** is used to set PPR32[PRI]; these are also shown in *Table 2-3*. Other defined **or Rx,Rx,Rx** hints shown in *Table 2-4* are ignored. PPR32[PRI] remains unchanged if the privilege state of the processor executing the instruction is lower than the privilege indicated in *Table 2-3*. PPR32[PRI] also remains unchanged if "000" is written to the field.

If CCR3[EN_EEPRI]=1 and MSR[EE]=0 and the thread priority is "a2low", the thread priority is increased to "a2medium"; PPR32 is unchanged. This function is provided to reduce delay in the processing of interrupts.

Table 3. Priority Levels

Rx	PPR32[PRI]	ISA Priority	A2 Hardware Priority with IUCR1[HIPRI] Setting				Privileged
			00	01	10	11	
31	001	very low	a2low	a2low	a2low	a2low	yes
1	010	low					no
6	011	medium low	a2medium	a2medium	a2medium	a2medium	no
2	100	medium					no
5	101	medium high	a2high	a2high	a2high	a2high	yes
3	110	high					yes
7	111	very high				a2high	hypv

Table 4. Other "or" Instruction Hints

Rx	Mnemonic	Reserved
27	yield	Yes
29	mdoio	Yes
30	mdoom	Yes

Table 5. Program Priority Register (PPR32)

Register Short Name:	PPR32	Read Access:	Any
Decimal SPR Number:	898	Write Access:	Any
Initial Value:	0x000000000000c0000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:42	///	0x0	<u>Reserved</u>
43:45	PRI	0b011	<u>Thread Priority</u> 001 very low (privileged) 010 low 011 medium low 100 medium 101 medium high (privileged) 110 high (privileged) 111 very high (hypervisor) Access violations or writing a value of zero will result in a nop.
46:63	///	0x0	<u>Reserved</u>

2.3.4.2 Instruction Unit Configuration Register 1 (IUCR1)

Register Short Name:	IUCR1	Read Access:	Hypv
-----------------------------	-------	---------------------	------

Decimal SPR Number:	883	Write Access:	Hypv
Initial Value:	0x0000000000001000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:49	///	0x0	<u>Reserved</u>
50:51	HIPRI	0b01	<u>High Priority privilege Level</u> A2 has three priority values implemented in hardware. This field configures which value in PPR32[PRI] corresponds to the implementations highest priority. 00 medium normal 01 medium high 10 high 11 very high
52:57	///	0x0	<u>Reserved</u>
58:63	THRES	0x0	<u>Low Priority Minimum Issue Count</u> Sets the amount of cycles between low priority issues, which is set by PPR32[PRI]. The number of cycles is equal to THRES*4. This field is not used when a thread is set to high or medium priority.

2.3.5 Resources Shared between Threads

All architected states are duplicated for each thread except for logical partitioning and memory. This allows each thread to look independent from a software standpoint. Some nonarchitected resources are shared between threads to save on the overall area for the core. *Section 2.3.6* provides more information about shared resources. *Section 2.3.7* on page 76 provides more information about duplicated resources.

2.3.6 Shared Resources

- Instruction ERAT array  Entries can be used as shared or thread specific.
- L1 instruction cache array
- Data ERAT array  Entries can be used as shared or thread specific.
- L1 data cache array
- Load miss queue
- Store queue 
- Microcode ROM array
- Branch history table  This is a configurable resource and can be set up to be shared or duplicated.
- SPR registers  Not all SPRs are shared. See *Table 14-1 Register Summary* on page 522 for more information.
- Instruction fetch pipeline
- Instruction issue
- Integer execution pipeline
-  LRAT

2.3.6.1 Accessing Shared Resources

When software executing in thread T_n writes a new value in an SPR (**mtspr**) that is shared with other threads, either of the following sequences of operations can be performed to ensure that the write operation has been performed with respect to other threads.

Sequence 1

- Disable all other threads (see *Thread Enable Register (TENS, TENC)* on page 70).
- Write to the shared SPR (**mtspr**).
- Perform a context synchronizing operation.
- Enable the previously disabled threads.

In the above sequence, the context synchronizing operation ensures that the write operation has been performed with respect to all other threads that share the SPR. The enabling of other threads ensures that subsequent instructions of the enabled threads use the new SPR value because enabling a thread is a context synchronizing operation.

Sequence 2

- All threads are put in hypervisor state and begin polling a storage flag.
- The thread updating the SPR does the following:
 - Writes to the SPR (**mtspr**).
 - Sets a storage flag indicating that the write operation was done.
 - Performs a context synchronizing operation.
- When other threads see the updated storage flag, they perform context synchronizing operations.

In the above sequence, the context synchronizing operation by the thread that writes to the SPR ensures that the write operation has been performed with respect to all other threads that share the SPR; the context synchronizing operation by the other threads ensures that subsequent instructions for these threads use the updated value.

2.3.7 Duplicated Resources

Link stack queue

Instruction buffer

Thread dependency

GPR register file

This includes extra registers for microcode instruction use.

SPR registers

Not all SPRs are duplicated. See *Table 14-1 Register Summary* on page 522 for more information.

Branch history table

This is a configurable resource and can be setup to be shared or duplicated.

2.3.8 Pipeline Sharing

Figure 2-1 shows the instruction flow for the A2 core.

Figure 1. A2 Core Instruction Unit



2.3.8.1 Instruction Cache

The instruction cache is a shared resource between all threads where a single thread can be selected each cycle dependent upon the number of instructions currently contained within that thread's instruction buffers. There are two watermarks within the instruction buffer that determine a thread's priority level for fetches that are empty and half-empty. The empty watermarks gives the corresponding thread high priority and a half-empty level gives the thread a low-priority fetch request. The high-priority and low-priority fetches are two separate round-robin queues to give each thread an even chance at getting the next command. A low-priority fetch is only issued when none of the high-priority water marks are active. The instruction cache and instruction directories are 4-way associative and are a shared resource between all threads. The branch prediction unit that is part of the instruction cache in *Figure 2-1* on page 77 contains a branch history table and link stack to allow proper branch resolution.

2.3.8.2 Instruction Buffer and Decode Dependency

The colored portion of *Figure 1-1* on page 49 contains all of the instruction buffer, decode, and dependency logic for each of the threads. This logic is duplicated for each thread to allow other threads with nondependent commands to be issued to maximize usage for the integer and floating-point pipelines.

2.3.8.3 Instruction Dispatch

2.3.8.4 Instruction Issue

2.3.8.5 Ram Unit

The Ram unit allows an external command to be issued within a given thread's instruction stream. This unit is a shared resource within a core in that only one thread can issue a Ram command at a time. It is software's responsibility to only allow one outstanding command per core, and it is necessary to poll the core until this command has completed before issuing any new commands.

2.3.8.6 Microcode Unit

The microcode unit (uCode) is partially shared and partially duplicated logic. The ROM that contains the actual stream of instructions to be issued is a shared unit; however, each thread contains its own microcode engine so that all four threads can be within a uCode stream at the same time. One of the engines will read a single command from the ROM each cycle based upon a fair round-robin scheme (not based upon the thread priority level for the issue logic), and issue that command to the appropriate thread's instruction buffer. If the instruction buffer is over halfway filled, the uCode will stop issuing new commands. In addition, it will not include this thread for ROM reads until the instruction buffer has drained below this point.

2.3.8.7 Execution Units

All execution units are shared between threads. Exceptions and flushes from one thread usually will not affect another thread.

However, a flush that will affect all threads when encountered by one of the threads is caused by a data cache invalidate (DCI) or instruction cache invalidate (ICI) that reaches completion. A DCI or ICI will flush all threads for one cycle to allow the L1 caches to be invalidated. Software is required to guarantee that the load miss queue is empty for all threads before execution of a DCI.

Another flush condition caused by one thread that can affect another thread occurs when reload data returning for an outstanding load collides with a load or store at the data cache array pins.

For a comprehensive list of flush conditions, see *Interrupt Conditions* on page 854.

Some multiply operations and all divide operations require recirculation within the multiply/divide unit, therefore blocking all other threads from executing multiplies and divides. This does not prevent other threads from executing any instructions other than multiplies and divides. If any multiply or divide instructions are issued and collide with a recirculating multiply or divide, the younger instructions are flushed. In the case of the multiplier, the size of the operands determines how many cycles are needed for recirculation. The width of the multiplier is 32 bits by 32 bits, so any operations that require multiplying 64-bit operands will require recirculation. If both operands are 32 bits, no recirculation is needed (in other words, the instruction is pipelined as normal). The width of the divider is 64 bits. Divide instructions dealing with 64-bit operands recirculate for 65 cycles, and operations with 32-bit operands recirculate for 32 cycles. No divide instructions are pipelined; they all require some recirculation.

A forward progress timer monitors that each thread is making forward progress. If the thread appears to be hung, thread priorities are adjusted to break out of a potential live-lock condition.

2.4 Registers

This section provides an overview of the register categories and types provided by the A2O Core. Detailed descriptions of each of the registers are provided within the chapters covering the functions with which they are associated (for example, the cache control and cache debug registers are described in *Instruction and Data Caches* on page 153). An alphabetical summary of all registers, including bit definitions, is provided in *Register Summary* on page 521

All registers in the A2O Core are architected as 64 bits wide, although certain bits in some registers are *reserved* and thus not necessarily implemented. For all registers with fields marked as reserved, these reserved fields should be written as 0 and read as *undefined*. The recommended coding practice is to perform the initial write to a register with reserved fields set to 0, and to perform all subsequent writes to the register using a read-modify-write strategy: read the register; use logical instructions to alter defined fields, leaving reserved fields unmodified; and write the register.

All of the registers are grouped into categories according to the processor functions with which they are associated. In addition, each register is classified as being of a particular *type*, as characterized by the specific instructions that are used to read and write registers of that type. Finally, most of the registers contained within the A2O Core are defined by the Power ISA Architecture, although some registers are implementation-specific and unique to the A2O Core.

Figure 2-2 illustrates the A2O Core registers contained in the user programming model; that is, those registers to which access is nonprivileged and that are available to both user and supervisor programs.

Figure 2. User Programming Model Registers

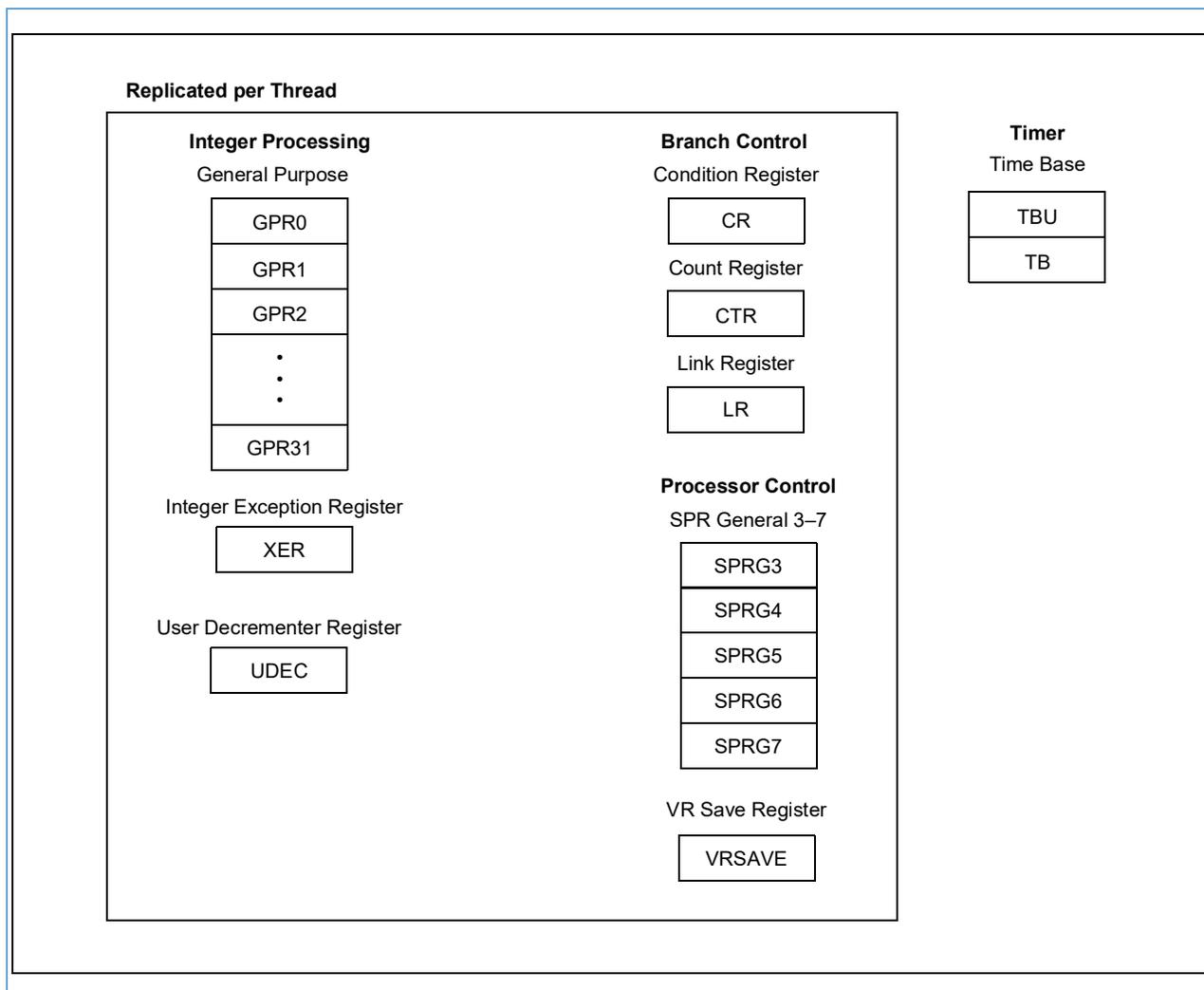


Table 14-1 on page 522 lists the A2 core registers contained in the supervisor or hypervisor programming model, to which access is privileged.

2.4.1 Register Mapping

Some special purpose register (SPR) accesses in guest state are mapped to analogous registers for the guest state. This removes the requirement for the hypervisor software to handle embedded hypervisor privilege interrupts for these accesses and make the required emulated changes by the hypervisor for these high-use registers.

Accesses to the registers listed in Table 2-6 are changed by the processor to the registers given in the table when the processor is in guest state (MSR[GS] = 1). Accesses to these registers are not mapped when not in guest state.

Table 6. Register Mapping

SPR Accessed	SPR Mapped to	Type of Access
SRR0	GSRR0	mtspr, mfspr
SRR1	GSRR1	mtspr, mfspr
ESR	GESR	mtspr, mfspr
DEAR	GDEAR	mtspr, mfspr
PIR	GPIR	mtspr, mfspr
SPRG0	GSPRG0	mtspr, mfspr
SPRG1	GSPRG1	mtspr, mfspr
SPRG2	GSPRG2	mtspr, mfspr
SPRG3	GSPRG3	mtspr, mfspr
USPRG3	GSPRG3	mtspr, mfspr

2.4.2 Register Types

There are five register types contained within and/or supported by the A2O Core. Each register type is characterized by the instructions that are used to read and write the registers of that type. The following subsections provide an overview of each of the register types and the instructions associated with them.

2.4.2.1 General Purpose Registers

The A2O Core contains 32 integer general purpose registers (GPRs); each contains 64 bits. In 32-bit mode, all instructions that operate on GPRs produce the same GPR results in 32-bit mode as in 64-bit mode.

Integer Processing on page 106 provides more information about integer operations and the use of GPRs.

2.4.2.2 Special Purpose Registers

Special Purpose Registers (SPRs) are directly accessed using the **mtspr** and **mfspr** instructions. In addition, certain SPRs might be updated as a side-effect of the execution of various instructions. For example, the Integer Exception Register (XER) (see *Integer Exception Register (XER)* on page 107) is an SPR that is updated with arithmetic status (such as carry and overflow) upon execution of certain forms of integer arithmetic instructions.

SPRs control the use of the debug facilities, timers, interrupts, memory management, caches, and other architected processor resources. *Table 14-1* on page 522 shows the mnemonic, name, and number for each SPR, in alphabetical order. Each of the SPRs is described in more detail within the section or chapter covering the function with which it is associated.

2.4.2.3 Condition Register

The Condition Register (CR) is a 32-bit register of its own unique type and is divided up into eight, independent 4-bit fields (CR0–CR7). The CR can be used to record certain conditional results of various arithmetic and logical operations. Subsequently, conditional branch instructions can designate a bit of the CR as one of the branch conditions (see *Wait Instruction* on page 95). Instructions are also provided for performing logical bit operations and for moving fields within the CR.

See *Condition Register (CR)* on page 103 for more information about the various instructions that can update the CR.

2.4.2.4 Machine State Register

The Machine State Register (MSR) is a register of its own unique type that controls important chip functions, such as the enabling or disabling of various interrupt types.

The MSR can be written from a GPR using the **mtmsr** instruction. The contents of the MSR can be read into a GPR using the **mfmsr** instruction. The MSR[EE] bit can be set or cleared atomically using the **wrtee** or **wrteei** instructions. The MSR contents are also automatically saved, altered, and restored by the interrupt-handling mechanism. See *Machine State Register (MSR)* on page 285 for more detailed information about the MSR and the function of each of its bits.

2.5 32-Bit Mode

2.5.1 64-Bit Specific Instructions

Instructions or registers that are categorized as 64-bit are only available in 64-bit implementations of the A2 core. In a 64-bit implementation in 32-bit mode, all instructions that operate on GPRs produce the same GPR results in 32-bit mode as in 64-bit mode. Instructions that set condition bits do so based on the 32-bit result computed. Effective addresses and all SPRs operate on the low-order 32 bits only unless otherwise stated.

2.5.2 32-Bit Instruction Selection

Any software that uses any of the instructions listed in the 64-bit category is considered 64-bit software. Generally speaking, 32-bit software should avoid using any instruction or instructions that depend on any particular setting of bits 0:31 of any 64-bit application-accessible system register, including General Purpose Registers, for producing the correct 32-bit results. Context switching might or might not preserve the upper 32 bits of application-accessible 64-bit system registers, and insertion of arbitrary settings of those upper 32 bits at arbitrary times during the execution of the 32-bit application must not affect the final result.

2.6 Instruction Categories

The Power ISA defines that each facility (including registers and fields therein) and instruction is in exactly one category. *Table 2-7* indicate the categories that are implemented by the A2 processor core.

Table 7. Category Listing (Sheet 1 of 3)

Implemented by A2 Core	Category	Abbreviation	Notes
Yes	Base	B	Required for all implementations.
No	Server	S	Required for server implementations.
Yes	Embedded	E	Required for embedded implementations.
No	Alternate Time Base	ATB	An additional time base; see Book II.
Yes	Cache Specification	CS	Specify a specific cache for some instructions; see Book II.

Table 7. Category Listing (Sheet 2 of 3)

Implemented by A2 Core	Category	Abbreviation	Notes
No	Decimal Floating-Point	DFP	Decimal floating-point facilities.
No	Decorated Storage	DS	Decorated storage facilities.
No	Embedded.Cache Debug	E.CD	Provides direct access to cache data and directory content.
Yes	Embedded.Cache Initialization	E.CI	Instructions that invalidate the entire cache.
No	Embedded.Device Control	E.DC	Embedded device control bus support.
No	Embedded.Enhanced Debug	E.ED	Embedded enhanced debug facility; see Book III-E.
Yes	Embedded.External PID	E.PD	Embedded external PID facility; see Book III-E.
Yes	Embedded.Hypervisor Embedded.Hypervisor.LRAT	E.HV E.HV.LRAT	Embedded logical partitioning and hypervisor facilities. Embedded hypervisor logical to real address translation.
Yes	Embedded.Little-Endian	E.LE	Embedded little-endian page attribute; see Book III-E.
Yes	Embedded.Page Table	E.PT	Embedded page table facility; see Book III-E.
Yes	Embedded.TLB Write Conditional	E.TWC	Embedded TLB write conditional facility; see Book III-E.
No	Embedded.Performance Monitor	E.PM	Embedded performance monitor example; see Book III-E.
Yes	Embedded.Processor Control	E.PC	Processor control facility; see Book III-E.
Yes	Embedded Cache Locking	ECL	Embedded cache locking facility; see Book III-E.
Yes	Embedded Multithreading Embedded multiThreading.Thread Management	EM EM.TM	Embedded multithreading; see Book III-E. Embedded multithreading thread management facility.
No	External Control	EXC	External control facility; see Book II.
No	External Proxy	EXP	External proxy facility; see Book III-E.
Yes	Floating-Point Floating-Point.Record	FP FP.R	Floating-point facilities. Floating-point instructions with Rc = 1.
No	Legacy Move Assist	LMV	Determine left most zero byte instruction.
No	Legacy Integer Multiply-Accumulate ¹	LMA	Legacy integer multiply-accumulate instructions.
No	Load/Store Quadword	LSQ	Load/store quadword instructions; see Book III-S.
Yes	Memory Coherence	MMC	Requirement for memory coherence; see Book II.
No	Move Assist	MA	Move assist instructions.
No	Processor Compatibility	PCR	Processor compatibility register.
No	Server.Performance Monitor	S.PM	Performance monitor example for servers; see Book III-S.
No	Server.Relaxed Page Table Alignment	S.RPTA	HTAB alignment on a 256 KB boundary; see Book III-S.
No	SPE.Processing Engine SPE.Embedded Float Scalar Double SPE.Embedded Float Scalar Single SPE.Embedded Float Vector	SP SP.FD SP.FS SP.FV	Facility for signal processing. GPR-based floating-point double-precision instruction set. GPR-based floating-point single-precision instruction set. GPR-based floating-point vector instruction set.
Yes	Store Conditional Page Mobility	SCPM	Store conditional accounting for page movement; see Book II.
No	Stream	STM	Stream variant of dcbt instruction; see Book II.

Table 7. Category Listing (Sheet 3 of 3)

Implemented by A2 Core	Category	Abbreviation	Notes
No	Strong Access Order	SAO	Assist for X86 emulation; see Book II.
No	Trace	TRC	Trace facility example; see Book III-S.
No	Variable Length Encoding	VLE	Variable length encoding facility; see Book VLE.
determined by AXU	Vector-Scalar Extension	VSX	Vector-scalar extension.
determined by AXU	Vector Little-Endian	V V.LE	Vector facilities. Little-endian support for vector storage operations.
Yes	Wait	WT	Wait instruction; see Book II.
Yes	64-Bit	64	Required for 64-bit implementations; not defined for 32-bit implementations.

2.7 Instruction Classes

Power ISA architecture defines all instructions as falling into exactly one of the following three classes, as determined by the primary opcode (and the extended opcode, if any):

1. Defined
2. Illegal
3. Reserved

2.7.1 Defined Instruction Class

This class of instructions consists of all the instructions defined in Power ISA. In general, defined instructions are guaranteed to be supported within a Power ISA system as specified by the architecture, either within the processor implementation itself or within emulation software supported by the system operating software.

As defined by Power ISA, any attempt to execute a defined instruction will:

- Cause an illegal instruction exception type of program interrupt, if the instruction is not recognized by the implementation; or
- Cause a floating-point unavailable interrupt if the instruction is recognized as a floating-point instruction, but floating-point processing is disabled; or
- Perform the actions described in the rest of this document, if the instruction is recognized and supported by the implementation. The architected behavior might cause other exceptions.

The A2O Core recognizes and fully supports all of the instructions in the defined class and in the categories supported, with a few exceptions. First, instructions that are defined for floating-point processing are not supported within the A2O Core, but can be implemented within an auxiliary processor and attached to the core using the AXU interface. If no such auxiliary processor is attached, attempting to execute any floating-point instructions causes an illegal instruction exception type of program interrupt. If an auxiliary processor that supports the floating-point instructions *is* attached, the behavior of these instructions is as defined above and as determined by the implementation details of the floating-point auxiliary processor.

2.7.2 Illegal Instruction Class

This class of instructions contains the set of instructions described in *Power ISA* Appendix D of Book Appendices. Illegal instructions are available for future extensions of the Power ISA; that is, some future version of the Power ISA might 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 will have no other effect.

An instruction consisting entirely of binary zeros is guaranteed always to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized storage will result in the invocation of the system illegal instruction error handler.

2.7.3 Reserved Instruction Class

This class of instructions contains the set of instructions described in *Power ISA* Appendix E of Book Appendices.

Reserved instructions are allocated to specific purposes that are outside the scope of the Power ISA.

Any attempt to execute a reserved instruction causes the system illegal instruction error handler to be invoked if the instruction is not implemented.

Because implementations are typically expected to treat reserved-nop instructions as true no-ops, these instruction opcodes are available for future extensions to Power ISA that have no effect on the architected state. Such extensions might include performance-enhancing hints, such as new forms of cache touch instructions. Software would be able to take advantage of the functionality offered by the new instructions and still remain backwards-compatible with implementations of previous versions of Power ISA.

The A2O Core implements all of the reserved-nop instruction opcodes as true no-ops. The specific reserved-nop opcodes are the following extended opcodes under primary opcode 31: 530, 562, 594, 626, 658, 690, 722, and 754.

2.8 Implemented Instruction Set Summary

This section provides an overview of the various types and categories of instructions implemented within the A2O Core. *Appendix A Processor Instruction Summary* on page 735 lists each implemented instruction alphabetically (and by opcode) along with a short-form description and its extended mnemonics.

Table 2-8 summarizes the A2O Core instruction set by category. Instructions within each category are described in subsequent sections.

Table 8. Instruction Categories

Category	Subcategory	Instruction Types
Integer	Integer Storage Access	load, store
	Integer Arithmetic	add, subtract, multiply, divide, negate
	Integer Logical	and, andc, or, orc, xor, nand, nor, xnor, extend sign, count leading zeros
	Integer Compare	compare, compare logical
	Integer Select	select operand
	Integer Trap	trap
	Integer Rotate	rotate and insert, rotate and mask
	Integer Shift	shift left, shift right, shift right algebraic
Branch		branch, branch conditional, branch to link, branch to count
Processor Control	Condition Register Logical	crand, crandc, cror, crorc, crnand, crnor, crxor, crxnor
	Register Management	move to/from SPR, move to/from MSR, write to external interrupt enable bit, move to/from CR
	System Linkage	system call, return from interrupt, return from critical interrupt, return from machine check interrupt
	Processor Synchronization	instruction synchronize
Storage Control	Cache Management	data allocate, data invalidate, data touch, data zero, data flush, data store, instruction invalidate, instruction touch
	TLB Management	read, write, search, synchronize
	Storage Synchronization	memory synchronize, memory barrier

Note: The A2 core does not implement any device control registers (DCRs). Move to and move from DCR instructions are dropped silently. They are no-ops and do not cause an exception.

2.8.1 Integer Instructions

Integer instructions transfer data between memory and the GPRs and perform various operations on the GPRs. This category of instructions is further divided into seven subcategories, described in the following sections.

2.8.1.1 Integer Storage Access Instructions

Integer storage access instructions load and store data between memory and the GPRs. These instructions operate on bytes, halfwords, and words. Integer storage access instructions also support loading and storing multiple registers, character strings, and byte-reversed data, and loading data with sign-extension.

Table 2-9 shows the integer storage access instructions in the A20 Core. In the table, the syntax “[u]” indicates that the instruction has both an “update” form (in which the RA addressing register is updated with the calculated address) and a “nonupdate” form. Similarly, the syntax “[x]” indicates that the instruction has both an “indexed” form (in which the address is formed by adding the contents of the RA and RB GPRs) and a “base + displacement” form (in which the address is formed by adding a 16-bit signed immediate value (specified as part of the instruction) to the contents of GPR RA).

Table 9. Integer Storage Access Instructions

Loads					Stores				
Byte	Halfword	Word	Double	Multiple/String	Byte	Halfword	Word	Double	Multiple/String
lbz[u][x]	lha[u][x] lhbrx lhz[u][x]	lwbrx lwz[u][x] lwa[u][x]	ld[u][x] ldbrx	lmw lswi lswx	stb[u][x]	sth[u][x] sthbrx	stw[u][x] stwbrx	std[u][x] stdbrx	stmw stswi stswx

Table 10. Integer Storage Access Instructions by External Process ID

Loads				Stores			
Byte	Halfword	Word	Double	Byte	Halfword	Word	Double
lbepx	lhexp	lwepx	ldexp	stbepx	sthepx	stwepx	stdepx

Table 2-11 shows how operands are handled depending on alignment. Optimal performance and configuration is achieved when operands are aligned.

Table 11. Operand Handling Dependent on Alignment (Sheet 1 of 2)

Operand		Big Endian - Boundary Crossing			Little Endian - Boundary Crossing		
Size	Byte Align	None	16 B Block	Virtual Page	None	16 B Block	Virtual Page
Integer							
8 Byte	8	Pipeline	N/A	N/A	Pipeline	N/A	N/A
	4	Pipeline	uCode	uCode	Pipeline	uCode	uCode
	<4	Pipeline	uCode	uCode	Pipeline	uCode	uCode
4 Byte	4	Pipeline	N/A	N/A	Pipeline	N/A	N/A
	<4	Pipeline	uCode	uCode	Pipeline	uCode	uCode
2 Byte	2	Pipeline	N/A	N/A	Pipeline	N/A	N/A
	<2	Pipeline	uCode	uCode	Pipeline	uCode	uCode
1 Byte	1	Pipeline	N/A	N/A	Pipeline	N/A	N/A
lmw, stmw	4	uCode	uCode	uCode	uCode	uCode	uCode
	<4	Trap	Trap	Trap	Trap	Trap	Trap
string		uCode	uCode	uCode	uCode	uCode	uCode
Float							
8 Byte	8	Pipeline	N/A	N/A	Pipeline	N/A	N/A
	4	Pipeline	uCode	uCode	Pipeline	uCode	uCode
	<4	Pipeline	uCode	uCode	Pipeline	uCode	uCode
4 Byte	4	Pipeline	N/A	N/A	Pipeline	N/A	N/A
	<4	Pipeline	uCode	uCode	Pipeline	uCode	uCode
Notes:							
1. If the storage operand spans two virtual pages that have different storage control attributes, an alignment interrupt occurs.							

Table 11. Operand Handling Dependent on Alignment (Sheet 2 of 2)

Operand		Big Endian - Boundary Crossing			Little Endian - Boundary Crossing		
Size	Byte Align	None	16 B Block	Virtual Page	None	16 B Block	Virtual Page
Any General Purpose AXU							
16 Byte	16	Pipeline	N/A	N/A	Pipeline	N/A	N/A
	8	Pipeline	uCode	uCode	Pipeline	uCode	uCode
	4	Pipeline	uCode	uCode	Pipeline	uCode	uCode
	<4	Pipeline	uCode	uCode	Pipeline	uCode	uCode
8 Byte	8	Pipeline	N/A	N/A	Pipeline	N/A	N/A
	4	Pipeline	uCode	uCode	Pipeline	uCode	uCode
	<4	Pipeline	uCode	uCode	Pipeline	uCode	uCode
4 Byte	4	Pipeline	N/A	N/A	Pipeline	N/A	N/A
	<4	Pipeline	uCode	uCode	Pipeline	uCode	uCode
2 Byte	2	Pipeline	N/A	N/A	Pipeline	N/A	N/A
	<2	Pipeline	uCode	uCode	Pipeline	uCode	uCode
1 Byte	1	Pipeline	N/A	N/A	Pipeline	N/A	N/A

Notes:

1. If the storage operand spans two virtual pages that have different storage control attributes, an alignment interrupt occurs.

2.8.1.2 Integer Arithmetic Instructions

Arithmetic operations are performed on integer or ordinal operands stored in registers. Instructions that perform operations on two operands are defined in a 3-operand format; an operation is performed on the operands, which are stored in two registers. The result is placed in a third register. Instructions that perform operations on one operand are defined in a 2-operand format; the operation is performed on the operand in a register, and the result is placed in another register. Several instructions also have immediate formats in which one of the source operands is a field in the instruction.

Most integer arithmetic instructions have versions that can update CR[CR0] and/or XER[SO, OV] (Summary Overflow, Overflow), based on the result of the instruction. Some integer arithmetic instructions also update XER[CA] (Carry) implicitly. See *Integer Processing* on page 106 for more information about how these instructions update the CR and/or the XER.

Table 2-12 lists the integer arithmetic instructions in the A20 Core. In the table, the syntax “[o]” indicates that the instruction has both an “o” form (which updates the XER[SO,OV] fields) and a “non-o” form. Similarly, the syntax “[.]” indicates that the instruction has both a “record” form (which updates CR[CR0]) and a “nonrecord” form.

Table 12. Integer Arithmetic Instructions

Add	Subtract	Multiply	Divide	Negate
add[o][.] addc[o][.] adde[o][.] addi addic[.] addis addme[o][.] addze[o][.]	subf[o][.] subfc[o][.] subfe[o][.] subfic subfme[o][.] subfze[o][.]	mulhw[.] mulhwu[.] mulli mullw[o][.] mulhd[.] mulhdu[.] mulld[o][.]	divw[o][.] divwu[o][.] divwe[o][.] divweu[o][.] divd[o][.] divdu[o][.] divde[o][.] divdeu[o][.]	neg[o][.]

2.8.1.3 Integer Logical Instructions

Table 2-13 lists the integer logical instructions in the A2O Core. See *Integer Arithmetic Instructions* on page 88 for an explanation of the “[.]” syntax.

Table 13. Integer Logical Instructions

And	And with Complement	Nand	Or	Or with Complement	Nor	Xor	Equivalence	Extend Sign	Count Leading Zeros	Permute	Parity
and[.] andi. andis.	andc[.]	nand[.]	or[.] ori oris	orc[.]	nor[.]	xor[.] xori xoris	eqv[.]	extsb[.] extsh[.] extsw[.]	cntlzw[.] cntlzd[.]	bpermd	prtyw prtyd

2.8.1.4 Integer Compare Instructions

These instructions perform arithmetic or logical comparisons between two operands and update the CR with the result of the comparison.

Table 2-14 lists the integer compare instructions in the A2O Core.

Table 14. Integer Compare Instructions

Arithmetic	Logical
cmp cmpi cmpb	cmpl cmpli

2.8.1.5 Integer Trap Instructions

Table 2-15 lists the integer trap instructions in the A2O Core.

Table 15. Integer Trap Instructions

Trap
tw
twi
td
tdi

2.8.1.6 Integer Rotate Instructions

These instructions rotate operands stored in the GPRs. Rotate instructions can also mask rotated operands.

Table 2-16 lists the rotate instructions in the A2O Core. See *Integer Arithmetic Instructions* on page 88 for an explanation of the “[.]” syntax.

Table 16. Integer Rotate Instructions

Rotate and Insert	Rotate and Mask	Rotate and Clear
rlwimi[.] rldimi[.]	rlwinm[.] rlwnm[.]	rldcl[.] rldcr[.] rldic[.] rldicl[.] rldicr[.]

2.8.1.7 Integer Shift Instructions

Table 2-17 lists the integer shift instructions in the A2O Core. Note that the shift right algebraic instructions implicitly update the XER[CA] field. See *Integer Arithmetic Instructions* on page 88 for an explanation of the “[.]” syntax.

Table 17. Integer Shift Instructions

Shift Left	Shift Right	Shift Right Algebraic
slw[.] sld[.]	srw[.] srd[.]	sraw[.] srawi[.] srad[.] sradi[.]

2.8.1.8 Integer Population Count Instructions

Table 2-18 lists the integer population count instructions in the A2O Core.

Table 18. Integer Population Count Instructions

Pop Count
popcntb
popcntw
popcntd

2.8.1.9 Integer Select Instruction

Table 2-19 lists the integer select instruction in the A2O Core. The RA operand is 0 if the RA field of the instruction is 0; it is the contents of GPR(RA) otherwise.

Table 19. Integer Select Instruction

Integer Select
isel

2.8.1.10 Binary Coded Decimal Assist Instructions

Table 2-20 lists the binary coded decimal assist instructions in the A2O Core.

Table 20. Binary Coded Decimal Assist Instructions

Pop Count
cdtbcd cbcdtd addg6s

2.8.2 Branch Instructions

These instructions unconditionally or conditionally branch to an address. Conditional branch instructions can test condition codes set in the CR by a previous instruction and branch accordingly. Conditional branch instructions can also decrement and test the Count Register (CTR) as part of branch determination and can save the return address in the Link Register (LR). The target address for a branch can be a displacement from the current instruction address or an absolute address or contained in the LR or CTR.

See *Wait Instruction* on page 95 for more information about branch operations.

Table 2-21 lists the branch instructions in the A2O Core. In the table, the syntax “[l]” indicates that the instruction has both a “link update” form (which updates LR with the address of the instruction after the branch) and a “nonlink update” form. Similarly, the syntax “[a]” indicates that the instruction has both an “absolute address” form (in which the target address is formed directly using the immediate field specified as part of the instruction) and a “relative” form (in which the target address is formed by adding the specified immediate field to the address of the branch instruction).

Table 21. Branch Instructions

Branch
b[l][a] bc[l][a] bcctr[l] bclr[l] bctar[l]

2.8.3 Processor Control Instructions

Processor control instructions manipulate system registers, perform system software linkage, and synchronize processor operations. The instructions in these three subcategories of processor control instructions are described below.

2.8.3.1 Condition Register Logical Instructions

These instructions perform logical operations on a specified pair of bits in the CR, placing the result in another specified bit. The benefit of these instructions is that they can logically combine the results of several comparison operations without incurring the overhead of conditional branching between each one. Software performance can significantly improve if multiple conditions are tested at once as part of a branch decision.

Table 2-22 lists the condition register logical instructions in the A20 Core.

Table 22. Condition Register Logical Instructions

crand	crnor
crandc	cror
creqv	crorc
crnand	crxor

2.8.3.2 Register Management Instructions

These instructions move data between the GPRs and control registers in the A20 Core.

Table 2-23 lists the register management instructions in the A20 Core.

Table 23. Register Management Instructions

CR	DCR ¹	MSR	SPR	TB
mcrf	mfdcr	mfmsr	mf spr	mttb
mcrxr	mfdcrx	mtmsr	mtspr	
mfcrr	mfdcrux	wrtree		
mfocrf	mtdcr	wrtreei		
mtcrf	mtdcrx			
mtocrf	mtdcrux			

1. When CCR2(EN_DCR) is zero, DCR instructions are dropped silently. They are no-ops and do not cause an exception.

2.8.3.3 System Linkage Instructions

These instructions invoke supervisor software level for system services and return from interrupts.

When executing in the guest state (MSR[GS,PR] = 0b10), execution of an **rfi** instruction is mapped to **rfgi** and the **rfgi** instruction is executed in place of the **rfi**.

Table 2-24 lists the system linkage instructions in the A20 Core.

Table 24. System Linkage Instructions

ehpriv
rfi
rfci
rfgi
rfmci
sc

2.8.3.4 Processor Control Instructions

The **msgsnd** and **msgclr** instructions are provided for sending and clearing messages to processors and other devices in the coherence domain. These instructions are hypervisor privileged.

Table 2-30 shows the processor control instructions in the A2O Core.

Table 25. Processor Control Instruction

msgsnd
msgclr

2.8.4 Storage Control Instructions

These instructions manage the instruction and data caches and the TLB of the A2O Core. Instructions are also provided to synchronize and order storage accesses. The instructions in these three subcategories of storage control instructions are described in the following sections.

2.8.4.1 Cache Management Instructions

These instructions control the operation of the data and instruction caches. Instructions are provided to fill, flush, invalidate, or zero data cache blocks, where a block is defined as a 64-byte cache line. Instructions are also provided to fill or invalidate instruction cache blocks.

Table 2-26 lists the cache management instructions in the A2O Core.

Table 26. Cache Management Instructions

Data Cache	Instruction Cache
dcbz	
dcbt	icbt
dcbtst	icbtst
dcbf	icbfc
dcbi	
dcbtls	
dcbtstls	
dcbz	
dcbt	
dcbtst	
dcbf	
dcbi	
dcbtls	
dcbtstls	
dcbz	

Table 27. Cache Management Instructions by External Process ID

Data Cache	Instruction Cache
dcbstep dcbtep dcbfep dcbtstep dcbzep	icbiep

2.8.4.2 Storage Visibility Instructions

The store visibility instruction is used to provide a hint that previous stores should be made visible with higher priority. This provides a method to push modified data to the coherence point.

Table 2-28 lists the storage visibility instructions in the A2O Core.

Table 28. Storage Visibility Instructions

makeitso

2.8.4.3 TLB Management Instructions

The TLB management instructions read and write entries of the TLB array and search the TLB array for an entry that will translate a given virtual address.

Table 2-29 lists the TLB management instructions in the A2O Core. See *Integer Arithmetic Instructions* on page 88 for an explanation of the “[.]” syntax.

Table 29. TLB Management Instructions

tlbre tlbsx[.] tlbsync tlbwe tlbivax

2.8.4.4 Processor Synchronization Instruction

The processor synchronization instruction, **isync**, forces the processor to complete all instructions preceding the **isync** before allowing any context changes as a result of any instructions that follow the **isync**. Additionally, all instructions that follow the **isync** will execute within the context established by the completion of all the instructions that precede the **isync**. See *Synchronization* on page 118 for more information about the synchronizing effect of **isync**.

Table 2-30 shows the processor synchronization instructions in the A2O Core.

Table 30. Processor Synchronization Instruction

isync sync dnh

2.8.4.5 Load and Reserve and Store Conditional Instructions

The load and reserve and store conditional instructions can be used to construct a sequence of instructions that appears to perform an atomic update operation on an aligned storage location.

The A2 core implements the exclusive access hint (EH) included in load and reserve instructions.

Table 31. Load and Reserve and Store Conditional Instructions

Loads				Stores			
Byte	Halfword	Word	Double	Byte	Halfword	Word	Double
lbarx	lharx	lwarx	ldarx	stbcx.	sthcx.	stwcx.	stdcx.

2.8.4.6 Storage Synchronization Instructions

The storage synchronization instructions allow software to enforce ordering amongst the storage accesses caused by load and store instructions, which by default are weakly-ordered by the processor. “Weakly-ordered” means that the processor is architecturally permitted to perform loads and stores generally out-of-order with respect to their sequence within the instruction stream, with some exceptions. However, if a storage synchronization instruction is executed, then all storage accesses prompted by instructions preceding the synchronizing instruction must be performed before any storage accesses prompted by instructions that come after the synchronizing instruction. See *Synchronization* on page 118 for more information about storage synchronization.

msync is an extended mnemonic for the *synchronize* instruction so that it can be coded with the L value as part of the mnemonic rather than as a numeric operand.

Table 2-30 shows the storage synchronization instructions in the A2O Core.

Table 32. Storage Synchronization Instructions

msync mbar

2.8.4.7 Wait Instruction

The **wait** instruction allows instruction fetching and execution to be suspended under certain conditions, depending on the value of the WC field. WC = 11 is treated as a no-op instruction. WC = 10 specifies a wake condition determined by the an A2 input signal called an_ac_sleep_en.

Table 2-30 shows the wait instructions in the A2O Core.

Table 33. Wait Instruction

wait

2.8.5 Initiate Coprocessor Instructions

Initiation of a coprocessor is requested by issuing the Initiate Coprocessor Store Word Indexed (**icswx**) instruction. A coprocessor is not a standard processor, but instead is a specialized processor that is capable of one or more particular tasks with the intent to provide acceleration of each task that might have otherwise been done by the program. See *Section 12.5 Coprocessor Instructions* on page 507.

Table 2-30 shows the **icswx** instructions in the A2O Core.

Table 34. Initiate Coprocessor Instructions

icswx[.]
icswepx[.]

2.8.5.1 Cache Initialization Instructions

The **dci** and **ici** instructions are privileged instructions, and if executed in supervisor mode they will flush invalidate the entire associated cache. They do not generate an address, nor are they affected by the access control mechanism.

Table 2-30 shows the cache initialization instructions in the A2O Core.

Table 35. Cache Initialization Instructions

dci
ici

The **dci** and **ici** instructions have a CT field. The following describes the affects of the CT field.

- CT = 0 indicates L1 only. The L1 cache will be invalidated and request is not sent to the L2.
- CT = 2 indicates L1 and L2. The L1 cache will be invalidated and request is sent to the L2.
- CT != 0,2 indicates a no-op. No L1 caches are invalidated and the request is not sent to the L2.

2.8.5.2 Debug Instructions

A **dnh** instruction is provided to stop instruction fetching and execution and allow the thread to be managed by an external debug facility. After the **dnh** instruction is executed, instructions are not fetched, interrupts are not taken, and the thread does not execute instructions.

Table 2-30 shows the debug instructions in the A2O Core.

Table 36. Debug Instructions

dnh

2.9 Branch Processing

The four branch instructions provided by A2O Core are summarized in *Table 2.8.2* on page 91. The following sections provide additional information about branch addressing, instruction fields, prediction, and registers.

2.9.1 Branch Addressing

The branch instruction (**b[l][a]**) specifies the displacement of the branch target address as a 26-bit value (the 24-bit LI field right-extended with 0b00). This displacement is regarded as a signed 26-bit number covering an address range of ± 32 MB. Similarly, the branch conditional instruction (**bc[l][a]**) specifies the displacement as a 16-bit value (the 14-bit BD field right-extended with 0b00). This displacement covers an address range of ± 32 KB.

For the relative form of the branch and branch conditional instructions (**b[l]** and **bc[l]**, with instruction field AA = 0), the target address is the address of the branch instruction itself (the current instruction address, or CIA) plus the signed displacement. This address calculation is defined to “wrap around” from the maximum effective address (0xFFFF_FFFF_FFFF_FFFF) to 0x0000_0000_0000_0000 and vice-versa.

For the absolute form of the branch and branch conditional instructions (**ba[l]** and **bca[l]**, with instruction field AA = 1), the target address is the sign-extended displacement. This means that with absolute forms of the branch and branch conditional instructions, the branch target can be within the first or last 32 MB or 32 KB of the address space, respectively.

The other two branch instructions, **bclr** (branch conditional to LR) and **bcctr** (branch conditional to CTR), do not use absolute or relative addressing. Instead, they use *indirect* addressing, in which the target of the branch is specified indirectly as the contents of the LR or CTR.

2.9.2 Branch Instruction BI Field

Conditional branch instructions can optionally test one bit of the CR, as indicated by instruction field BO[0] (see *Section 2.9.3*). The value of instruction field BI specifies the CR bit to be tested (32-63). The BI field is ignored if BO[0] = 1. The branch (**b[l][a]**) instruction is by definition unconditional; hence, it does not have a BI instruction field. Instead, the position of this field is part of the LI displacement field.

2.9.3 Branch Instruction BO Field

The BO field specifies the condition under which a conditional branch is taken and whether the branch decrements the CTR as shown in *Table 2-37*. In the table, M = 0 in 64-bit mode and M = 32 in 32-bit mode. The branch (**b[l][a]**) instruction is by definition unconditional; hence, it does not have a BO instruction field. Instead, the position of this field is part of the LI displacement field.

Conditional branch instructions can optionally test one bit in the CR. This option is selected when BO[0] = 0. If BO[0] = 1, the CR does not participate in the branch condition test. If the CR condition option is selected, the condition is satisfied (branch can occur) if the CR bit selected by the BI instruction field matches BO[1].

Conditional branch instructions can also optionally decrement the CTR by one and test whether the decremented value is 0. This option is selected when BO[2] = 0. If BO[2] = 1, the CTR is not decremented and does not participate in the branch condition test. If the CTR decrement option is selected, BO[3] specifies the condition that must be satisfied to allow the branch to be taken. If BO[3] = 0, CTR \neq 0 is required for the branch to occur. If BO[3] = 1, CTR = 0 is required for the branch to occur.

Table 37. BO Field Encodings

BO Description	Description
0000z	Decrement the CTR, then branch if the decremented CTRM:63 neq 0 and CRBI = 0.
0001z	Decrement the CTR, then branch if the decremented CTRM:63 = 0 and CRBI = 0.
001at	Branch if CRBI = 0.
0100z	Decrement the CTR, then branch if the decremented CTRM:63 neq 0 and CRBI = 1.
0101z	Decrement the CTR, then branch if the decremented CTRM:63 = 0 and CRBI = 1.
011at	Branch if CRBI = 1.
1a00t	Decrement the CTR, then branch if the decremented CTRM:63 neq 0.
1a01t	Decrement the CTR, then branch if the decremented CTRM:63 = 0.
1z1zz	Branch always.

Notes:

- 'z' denotes a bit that is ignored.
- The 'a' and 't' bits are used as described in *Table 2-38* on page 98.

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 *Table 2-38*.

Table 38. '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.

This implementation has dynamic mechanisms for predicting whether a branch will be taken. Because the dynamic prediction is likely to be very accurate and is likely to be overridden by any hint provided by the "at" bits, the "at" bits should be set to 0b00 unless the static prediction implied by at = 0b10 or at = 0b11 is highly likely to be correct.

2.9.4 Branch Prediction

The following sections detail the methods by which the branch predictor decodes incoming branches, generates predictions for both the direction and target of these branches, and guides instruction flow based on these predictions.

Features

- Static hardware, dynamic hardware, and software branch prediction
- 4 simultaneous predictions per cycle
- Alloy predictor: separate usually taken/not taken tables, selected via local history table
- 3 branch history tables (BHT): (2) 2-bits x 4 instructions x 1k entries = 8kb per table / (1) 1-bit x 4 instructions x 512 entries = 2k table
- BTB/BTAC for early prediction of target addresses, size TBD

- 8-entry link stack for target address prediction of subroutine-return indirect branches

The following sections detail the methods by which the Branch Predictor decodes incoming branches, generates predictions for both the direction and target of these branches, and guides instruction flow based on these predictions.

2.9.4.1 Branch Decoder

Prior to prediction, every instruction cacheline is passed through Branch Decoder (IUQ_BD logic included as part of unit IUQ_IC). The primary purpose of the Branch Decoder is to identify any valid branch instructions contained within the cacheline. Valid branches include b, bc, bclr, bcctr, and their derivatives.

Table 39.

Instruction	Description	Primary Opcode (0:5)	Extended Opcode (21:30)
B	Hard Branch	010010	~~~~~
BC	Branch Conditional	010000	~~~~~
BCLR	Branch Conditional to Link Register	010011	0000010000
BCCTR	Branch Conditional to Count Register	010011	1000010000

Branch Decoder also decodes any hints contained within the branch instructions. Hints may be specified for any branch conditional instruction (bc, bclr, bcctr, and their derivatives). Hints are encoded in the branch instruction's BO field.

Table 40.

Hint	BO(0:4)
Taken	0x111 110x1 1x1xx
Not Taken	0x110 110x0

All other values of BO indicate that no hint was specified. The use of these hints in software prediction is discussed in a later section.

Ultimately, Branch Decoder generates four flags that will be used by Branch Predictor at a later stage. These bits are appended to the original 32-bit instruction, and are carried along as part of the instruction until needed.

Table 41.

Decode Bit	Flag	Description	Instruction Bit
Branch_decode(0)	Br_val	This instruction is a valid branch	Instruction(32)
Branch_decode(1)	B	This instruction is a hard branch if br_val=1 This instruction is microcoded if br_val=0	Instruction(33)
Branch_decode(2)	Hint_val	This instruction contains a hint	Instruction(34)
Branch_decode(3)	Hint	0 = branch should not be taken 1 = branch should be taken	Instruction(35)

Branch Decoder is configurable. Each type of branch, as well as software hints, may be enabled or disabled at this point for prediction purposes. Configuration bits are controlled via the IUCR0 register.

2.9.4.2 Branch Direction Prediction

Branch direction prediction is performed simultaneously on up to 4 valid branch instructions per cycle in order to guide instruction flow appropriately. The first taken branch per cycle will cause Branch Predictor to redirect the instruction flow, and all subsequent instructions within that cycle are considered invalid.

Hard branches, by definition, are always taken. In all other cases, however, the Branch Predictor must make a choice, and predict which direction a branch will take. Branch Predictor contains three direction prediction mechanisms. In order of priority, they are: Software Prediction, Dynamic Hardware Prediction, and Static Hardware Prediction. Each of these mechanisms may be independently enabled or disabled via the IUCR0 register. If enabled, a higher priority prediction will always override a lower priority prediction. If all prediction mechanisms are disabled, the Branch Predictor will predict 'branch not taken', and will take no action.

All branch direction predictions are completed in IU2. Fetch redirect occurs in IU3 for indirect branches that use predicted target addresses, and IU4 for direct branches that use calculated target addresses. It is expected that the Branch Execution Unit (BR) will flag a misprediction and flush if any predicted branch direction does not match the actual resolved branch direction.

2.9.4.3 Software Prediction

Software prediction relies on hints encoded into the branch instruction itself to determine whether a branch will be predicted taken or not. Assuming well written software, this method can yield relatively accurate predictions with trivial resource utilization, since implementing software prediction requires only a small amount of combinatorial logic. If a hint is valid, and the hint is taken, the branch will be evaluated taken. If a hint is valid, and the hint is not taken, the branch will be evaluated not taken. If the hint is not valid, some other method must be used to predict the branch. The resolution of software hints is discussed in an earlier section.

When programming with branches, the user is expected to apply hints responsibly, using them only where appropriate. Appropriate usage generally involves a very predictable software construct, like a loop, where the user knows that a branch will be executed in every iteration except the last. Conversely, software hints may also be appropriate in cases where branching will be truly random, and not prone to trends. In truly random cases, hinting that a branch will NOT be taken may be the most efficient course, as it will eliminate frequent false branches and subsequent redirections to correct the instruction flow.

It is not required that software hints be used. If no hints are specified (or Software Prediction is disabled), a prediction will be generated based on either Dynamic Hardware Prediction or Static Hardware Prediction

2.9.4.4 Dynamic Hardware Prediction

Dynamic Hardware Prediction relies on a particular instruction's recent history to determine whether a branch will be predicted taken or not. This method requires a great deal more resources than Software Prediction, since a large array, or Branch History Table (BHT), is necessary to store history information for every branch instruction that has been executed. This method has the advantage, however, that it can identify and utilize trends in the software that the programmer has not explicitly identified or may not be aware of.

Dynamic prediction begins when a valid instruction initiates a simultaneous read access to all 3 BHT's in IU0. The BHT's are indexed based on the current instruction IFAR combined with some number of global history bits. In the Alloy prediction scheme, two identically indexed tables store branch history in 2-bit saturating counters. One of these tables contains normally taken branches, and the other contains normally not taken branches. A third table stores a single bit of local history for each indexed branch. The output of this table selects whether to use the prediction from the normally taken or normally not taken BHT. Since any or all of the four instructions presented per cycle may be valid branches, all four branch histories are accessed simultaneously. BHT prediction data is available in IU2. Based on the IU2 IFAR, BHT data is rotated to sync up with prior instruction rotation in IUQ_IC. Branch histories are available for evaluation in IU2.

Assuming the instruction is not a hard branch, and that software prediction has not been applied, the value returned from the BHT's are used to predict the current instruction. If the history MSB is a '0', the branch is predicted not taken; if it is a '1', the branch is predicted taken. Both history bits are appended to the branch instruction for later use. Once the instruction reaches the execution unit and the branch resolved, the executed branch direction is returned to the Branch Predictor, along with the associated history bits and instruction IFAR. The executed branch direction is then used to update the branch history for that instruction.

2.9.4.5 Performance Model Dynamic Predictor

The performance model contains an "Alloy" branch predictor. This is logically three tables, but physically two. One table is 16kb (8k entry x 2b/entry), and one table is 2kb (2k entry x 1b/entry). The basic idea is that a bimodal predictor (the small table) selects the output from one of two gshare tables. This tends to put taken outcomes in one table and non-taken outcomes in the other table, reducing aliasing and effectively improving learning time.

Per-branch logical view:

The bimodal table requires 11 index bits. These are the 11 low-order IAR bits (IAR[51:61]), XOR'd with the two most recent GBH bits. The GBH bits are XOR'd into IAR[51:52]. This results in one bit of local branch history.

The gshare tables require 12 index bits. These are the 12 low-order IAR bits (IAR[50:61]), XOR'd with 10 bits of GBH. Again, the GBH bits are XOR'd with the high-order bits of the IAR.

All three tables are accessed in parallel.

The local branch history bit is used to late-select which gshare table provides the prediction. The take/not-take prediction is computed in the normal way (upper bit of the 2b saturating counter).

Per-fetch group lower-level view:

Physically, four bits are read from the bimodal table, one bit for each of the four instructions in the fetch group. The 512x4b array is indexed using IAR[51:59]. This provides 1 local branch history for each instruction in the fetch group.

Similarly, 8 entries, 16b, are read from the single combined gshare table. The 1Kx16b array is indexed by IAR[50:59]. This gives a pair of 2b counters for each instruction in the fetch group. For each instruction, the 2b counter used for prediction is selected by the local branch history bit.

2.9.4.6 During BHT update, the bimodal table is always updated with the last direction. Only the gshare counter used during prediction is updated. The other gshare counter is not updated. Bimodal Branch History

An instruction's branch history is nothing more than a saturating 2-bit counter. Each time an executed branch is resolved taken, its history bits are incremented up to a maximum value of "11". Each time an executed branch is resolved not taken, its history bits are decremented down to a minimum value of "00". The updated history value is written back into the BHT based on the executed instruction IFAR, and will be available the next time the instruction is issued.

2.9.4.7 Bimodal Branch History

An instruction's branch history is nothing more than a saturating 2-bit counter. Each time an executed branch is resolved taken, its history bits are incremented up to a maximum value of "11". Each time an executed branch is resolved not taken, its history bits are decremented down to a minimum value of "00". The updated history value is written back into the BHT based on the executed instruction IFAR, and will be available the next time the instruction is issued.

2.9.5 Branch Control Registers

There are three registers in the A20 Core that are associated with branch processing. They are described in the following sections.

2.9.5.1 Link Register (LR)

The LR is written from a GPR using **mtspr**, and it can be read into a GPR using **mf spr**. The LR can also be updated by the "link update" form of branch instructions (instruction field LK = 1). Such branch instructions load the LR with the address of the instruction following the branch instruction (4 + address of the branch instruction). Thus, the LR contents can be used as a return address for a subroutine that was entered using a link update form of branch. The **bclr** instruction uses the LR in this fashion, enabling indirect branching to any address.

When being used as a return address by a **bclr** instruction, bits 62:63 of the LR are ignored because all instruction addresses are on word boundaries.

Access to the LR is nonprivileged.

Register Short Name:	LR	Read Access:	Any
Decimal SPR Number:	8	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	LR	0x0	<p><u>Link Register</u></p> <p>The Link Register (LR) is a 64-bit register. It can be used to provide the branch target address for the Branch Conditional to Link Register instruction, and it holds the return address after Branch instructions for which LK=1</p>

2.9.5.2 Count Register (CTR)

The CTR is written from a GPR using **mtspr**, and it can be read into a GPR using **mfspr**. The CTR contents can be used as a loop count that gets decremented and tested by conditional branch instructions that specify count decrement as one of their branch conditions (instruction field BO[2] = 0). Alternatively, the CTR contents can specify a target address for the **bcctr** instruction, enabling indirect branching to any address.

Access to the CTR is nonprivileged.

Register Short Name:	CTR	Read Access:	Any
Decimal SPR Number:	9	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	CTR	0x0	<p><u>Counter</u></p> <p>The Count Register (CTR) is a 64-bit register. It can be used to hold a loop count that can be decremented during execution of Branch instructions that contain an appropriately coded BO field. If the value in the Count Register is 0 before being decremented, it is -1 afterward. The Count Register can also be used to provide the branch target address for the Branch Conditional to Count Register instruction</p>

2.9.5.3 Condition Register (CR)

The CR is used to record certain information (“conditions”) related to the results of the various instructions that are enabled to update the CR. A bit in the CR can also be selected to be tested as part of the condition of a conditional branch instruction.

The CR is organized into eight 4-bit fields (CR0–CR7), as shown in the following table. *Table 2-42* on page 104 lists the instructions that update the CR.

Access to the CR is nonprivileged.

Register Short Name:	CR	Read Access:	Any
Decimal SPR Number:	N/A	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:35	CR0	0b0000	<u>Condition Register Field 0</u>
36:39	CR1	0b0000	<u>Condition Register Field 1</u>
40:43	CR2	0b0000	<u>Condition Register Field 2</u>
44:47	CR3	0b0000	<u>Condition Register Field 3</u>
48:51	CR4	0b0000	<u>Condition Register Field 4</u>
52:55	CR5	0b0000	<u>Condition Register Field 5</u>
56:59	CR6	0b0000	<u>Condition Register Field 6</u>
60:63	CR7	0b0000	<u>Condition Register Field 7</u>

Table 42. CR Updating Instructions

Integer						Processor Control	Storage Control	Implementation Specific
Storage Access	Arithmetic	Logical	Compare	Rotate	Shift	CR-Logical and Register Management	TLB Management	See Section 12 on page 475
stwcx. stdcx.	add.[o] addc.[o] adde.[o] addc. addme.[o] addze.[o] subf.[o] subfc.[o] subfe.[o] subfme.[o] subfze.[o] mulhw. mulhwu. mullw.[o] divw.[o] divwu.[o] divdo. divduo. neg.[o] mulhd. mulhdu. mulld. mulldo. divd. divdu.	and. andi. andis. andc. nand. or. orc. nor. xor. eqv. extsb. extsw. extsh. cntlzw. cntlzd.	cmp cmpi cmpl cmpli	rlwimi. rlwinm. rlwnm. rldcl. rldcr. rldic. rldicl. rldicr. rldimi.	slw. srw. sraw. srawi. sld. srad.[i] srd.	crand crandc creqv crnand crnor cror crorc crxor mcrf mcrxr mtrcf	tlbsx. tlbsrx.	icswx. icswepx. eratsx. ldawx.

To summarize, the CR can be accessed in any of the following ways:

- **mfc** reads the CR into a GPR. Note that this instruction does not *update* the CR and is therefore not listed in *Table 2-42*.
- Conditional branch instructions can designate a CR bit to be used as a branch condition. Note that these instructions do not *update* the CR and are therefore not listed in *Table 2-42*.
- **mtrcf** sets specified CR fields by writing to the CR from a GPR, under control of a mask field specified as part of the instruction.
- **mcrf** updates a specified CR field by copying another specified CR field into it.
- **mcrxr** copies certain bits of the XER into a specified CR field, and clears the corresponding XER bits.
- Integer compare instructions update a specified CR field.

- CR-logical instructions update a specified CR bit with the result of any one of eight logical operations on a specified pair of CR bits.
- Certain forms of various integer instructions (the “.” forms) implicitly update CR[CR0], as do certain forms of the auxiliary processor instructions implemented within the A2O Core.
- Auxiliary processor instructions can, in general update, a specified CR field in an implementation-specified manner. In addition, if an auxiliary processor implements the floating-point operations specified by the Power ISA, those instructions update the CR in the manner defined by the architecture. See *Book III-E: Power ISA Architecture Enhanced for Embedded Applications* for details.

CR[CR0] Implicit Update By Integer Instructions

Most of the CR-updating instructions listed in *Table 2-42* implicitly update the CR0 field. These are the various “dot-form” instructions, indicated by a “.” in the instruction mnemonic. Most of these instructions update CR[CR0] according to an arithmetic comparison of 0 with the 64-bit result in 64-bit mode or comparison with the lower 32 bits of the result in 32-bit mode. The compare to 0 uses a signed comparison, independent of whether the actual operation being performed by the instruction is considered “signed” or not. For example, logical instructions such as **and.**, **or.**, and **nor.** update CR[CR0] according to this signed comparison to 0, even though the result of such a logical operation is not typically interpreted as a signed value. For each of these dot-form instructions, the individual bits in CR[CR0] are updated as follows:

CR[CR0] ₀ — LT	Less than 0; set if the most-significant bit of the 64-bit result in 64-bit mode or the most-significant bit of the 32-bit result in 32-bit mode is 1.
CR[CR0] ₁ — GT	Greater than 0; set if the 64-bit result in 64-bit mode or the 32-bit result in 32-bit mode is nonzero and the most-significant bit of the result is 0.
CR[CR0] ₂ — EQ	Equal to 0; set if the 64-bit result in 64-bit mode or the 32-bit result in 32-bit mode is 0.
CR[CR0] ₃ — SO	Summary overflow; a copy of XER[SO] at the completion of the instruction (including any XER[SO] update being performed the instruction itself).

Note that if an arithmetic overflow occurs, the “sign” of an instruction result indicated in CR[CR0] might not represent the “true” (infinitely precise) algebraic result of the instruction that set CR0. For example, if an **add.** instruction adds two large positive numbers and the magnitude of the result cannot be represented as a twos-complement number in a 64-bit register in 64-bit mode or in a 32-bit register in 32-bit mode, an overflow occurs and CR[CR0]₀ is set, even though the infinitely precise result of the add is positive.

Similarly, adding the largest 64-bit twos-complement negative number (0x8000_0000_0000_0000) to itself in 64-bit mode or the largest 32-bit twos-complement negative number (0x8000_0000) to itself in 32-bit mode results in an arithmetic overflow and 0x0000_0000_0000_0000 in 64-bit mode or 0x0000_0000 (in bits 32:63) in 32-bit mode is recorded in the target register. CR[CR0]₂ is set, indicating a result of 0, but the infinitely precise result is negative.

CR[CR0]₃ is a copy of XER[SO] at the completion of the instruction, whether or not the instruction that is updating CR[CR0] is also updating XER[SO]. Note that if an instruction causes an arithmetic overflow but is not of the form that actually updates XER[SO], then the value placed in CR[CR0]₃ does not reflect the arithmetic overflow that occurred on the instruction (it is merely a copy of the value of XER[SO] that was already in the XER before the execution of the instruction updating CR[CR0]).

There are a few dot-form instructions that do not update CR[CR0] in the fashion described above. These instructions are: **stwcx.**, **stdcx.**

CR Update By Integer Compare Instructions

Integer compare instructions update a specified CR field with the result of a comparison of two 64-bit numbers in 64-bit mode or two 32-bit numbers in 32-bit mode, the first of which is from a GPR and the second of which is either an immediate value or from another GPR. There are two types of integer compare instructions, *arithmetic* and *logical*, and they are distinguished by the interpretation given to the 64-bit numbers in 64-bit mode or to the 32-bit numbers in 32-bit mode being compared. For *arithmetic* compares, the numbers are considered to be signed, whereas for *logical* compares, the numbers are considered to be unsigned. As an example, consider the comparison of 0 with 0xFFFF_FFFF_FFFF_FFFF. In an *arithmetic* compare, 0 is larger; in a *logical* compare, 0xFFFF_FFFF_FFFF_FFFF is larger.

A compare instruction can direct its result to any CR field. The BF field (bits 6:8) of the instruction specifies the CR field to be updated. After a compare, the specified CR field is interpreted as follows:

- CR[(BF)]₀ — LT The first operand is less than the second operand.
- CR[(BF)]₁ — GT The first operand is greater than the second operand.
- CR[(BF)]₂ — EQ The first operand is equal to the second operand.
- CR[(BF)]₃ — SO Summary overflow; a copy of XER[SO].

2.9.5.4 Target Address Register (TAR)

The TAR is written from a GPR using **mtspr**, and it can be read into a GPR using **mfspr**. The TAR provides the branch target address for the Branch Conditional to Branch Target Address Register instruction.

Register Short Name:	TAR	Read Access:	Any
Decimal SPR Number:	815	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	TAR	0x0	<p><u>Target Address Register</u></p> <p>The Target Address Register (TAR) is a 64-bit register. It can be used to provide bits 0:61 of the branch target address for the <i>Branch Conditional to Branch Target Address Register</i> instruction. Bits 62:63 are ignored by the hardware but can be set and reset by software.</p>

2.10 Integer Processing

Integer processing includes loading and storing data between memory and GPRs, as well as performing various operations on the values in GPRs and other registers (the categories of integer instructions are summarized in *Table 2-8* on page 86). The sections that follow describe the registers that are used for integer processing and how they are updated by various instructions. In addition, *Condition Register (CR)* on page 103 provides more information about the CR updates caused by integer instructions.

2.10.1 General Purpose Registers (GPRs)

The A2O Core contains 32 GPRs. The contents of these registers can be transferred to and from memory using integer storage access instructions. Operations are performed on GPRs by most other instructions.

Access to the GPRs is nonprivileged.

Table 43. GPR Registers

Bits	Field Name	Initial Value	Description
0:31	N/A	N/A	<u>General Purpose Register Data</u>

2.10.2 Integer Exception Register (XER)

The XER records overflow and carry indications from integer arithmetic and shift instructions. It also provides a byte count for string indexed integer storage access instructions (**lswx** and **stswx**). Note that the term *exception* in the name of this register does not refer to exceptions as they relate to interrupts, but rather to the *arithmetic* exceptions of carry and overflow.

The following table illustrates the fields of the XER, while *Table 2-44* and *Table 2-45* list the instructions that update XER[SO,OV] and the XER[CA] fields, respectively. The sections that follow the figure and tables describe the fields of the XER in more detail.

Access to the XER is nonprivileged.

Register Short Name:	XER	Read Access:	Any
Decimal SPR Number:	1	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32	SO	0b0	<u>Summary Overflow</u> The Summary Overflow bit is set to 1 whenever an instruction (except mtspr) sets the Overflow bit.
33	OV	0b0	<u>Overflow</u> The Overflow bit is set to indicate that an overflow has occurred during execution of an instruction.

Bit(s):	Field Name:	Init	Description
34	CA	0b0	<u>Carry</u> Carry bit from extend arithmetic ops.
35:56	///	0x0	<u>Reserved</u>
57:63	SI	0x0	<u>String Index</u> This field specifies the number of bytes to be transferred by a Load String Indexed or Store String Indexed instruction.

Table 44. XER[SO,OV] Updating Instructions

Integer Arithmetic					Processor Control
Add	Subtract	Multiply	Divide	Negate	Register Management
addo [,]	subfo [,]	mullwo [,]	divwo [,]	nego [,]	mtspr
addco [,]	subfco [,]		divwuo [,]		mcrxr
addeo [,]	subfeo [,]	mulldo [,]			
addmeo [,]	subfmeo [,]		divdo [,]		
addzeo [,]	subfzeo [,]		divduo [,]		

Table 45. XER[CA] Updating Instructions

Integer Arithmetic		Integer Shift	Processor Control
Add	Subtract	Shift Right Algebraic	Register Management
addc [o][,]	subfc [o][,]	sraw [,]	mtspr
adde [o][,]	subfe [o][,]	srawi [,]	mcrxr
addic [,]	subfic		
addme [o][,]	subfme [o][,]	srad [,]	
addze [o][,]	subfze [o][,]	sradi [,]	

2.10.2.1 Summary Overflow (SO) Field

This field is set to 1 when an instruction is executed that causes XER[OV] to be set to 1, except for the case of **mtspr**(XER), which writes XER[SO,OV] with the values in (RS)_{0:1}, respectively. Once set, XER[SO] is not reset until either an **mtspr**(XER) is executed with data that explicitly writes 0 to XER[SO], or until an **mcrxr** instruction is executed. The **mcrxr** instruction sets XER[SO] (as well as XER[OV,CA]) to 0 after copying all three fields into CR[CR0]_{0:2} (and setting CR[CR0]₃ to 0).

Given this behavior, XER[SO] does not necessarily indicate that an overflow occurred on the most recent integer arithmetic operation, but rather that one occurred at some time subsequent to the last clearing of XER[SO] by **mtspr**(XER) or **mcrxr**.

XER[SO] is read (along with the rest of the XER) into a GPR by **mf spr**(XER). In addition, various integer instructions copy XER[SO] into CR[CR0]₃ (see *Condition Register (CR)* on page 103).

2.10.2.2 Overflow (OV) Field

This field is updated by certain integer arithmetic instructions to indicate whether the infinitely precise result of the operation can be represented in 64 bits when in 64-bit mode or in 32 bits when in 32-bit mode. For those integer arithmetic instructions that update XER[OV] and produce *signed* results, XER[OV] = 1 if the result is greater than $2^{63}-1$ or less than -2^{63} (in 64-bit mode) or if the result is greater than $2^{31}-1$ or less than -2^{31}

(when in 32-bit mode); otherwise, XER[OV] = 0. For those integer arithmetic instructions that update XER[OV] and produce *unsigned* results (certain integer divide instructions and multiply-accumulate auxiliary processor instructions), XER[OV] = 1 if the result is greater than $2^{64}-1$ (when in 64-bit mode) or if the result is greater than $2^{31}-1$ (when in 32-bit mode); otherwise, XER[OV] = 0.

The **mtspr**(XER) and **mcrxr** instructions also update XER[OV]. Specifically, **mcrxr** sets XER[OV] (and XER[SO,CA]) to 0 after copying all three fields into CR[CR0]_{0:2} (and setting CR[CR0]₃ to 0), while **mtspr**(XER) writes XER[OV] with the value in (RS)₁.

XER[OV] is read (along with the rest of the XER) into a GPR by **mfspir**(XER).

2.10.2.3 Carry (CA) Field

This field is updated by certain integer arithmetic instructions (the “carrying” and “extended” versions of add and subtract) to indicate whether or not there is a carry-out of the most-significant bit of the 64-bit result when in 64-bit mode or a carry-out of the most-significant bit of the 32-bit result when in 32-bit mode. XER[CA] = 1 indicates a carry. The integer shift right algebraic instructions update XER[CA] to indicate whether or not any 1 bits were shifted out of the least significant bit of the result, if the source operand was negative.

The **mtspr**(XER) and **mcrxr** instructions also update XER[CA]. Specifically, **mcrxr** sets XER[CA] (as well as XER[SO,OV]) to 0 after copying all three fields into CR[CR0]_{0:2} (and setting CR[CR0]₃ to 0), while **mtspr**(XER) writes XER[CA] with the value in (RS)₂.

XER[CA] is read (along with the rest of the XER) into a GPR by **mfspir**(XER). In addition, the “extended” versions of the add and subtract integer arithmetic instructions use XER[CA] as a source operand for their arithmetic operations.

2.10.2.4 Transfer Byte Count (TBC) Field

The TBC field is used by the string indexed integer storage access instructions (**lswx** and **stswx**) as a byte count. The TBC field is also written by **mtspr**(XER) with the value in (RS)_{25:31}.

XER[TBC] is read (along with the rest of the XER) into a GPR by **mfspir**(XER).

2.11 Processor Control

Except for the MSR, each of the following registers is described in more detail in the following sections. The MSR is described in more detail in *Machine State Register (MSR)* on page 285.

- Machine State Register (MSR) - Controls interrupts and other processor functions.
- Special Purpose Registers General (SPRGs) - SPRs for general purpose software use.
- Vector Save Register (VRSAVE) - Can be used to indicate which VRs are currently in use by a program.
- Processor Version Register (PVR) - Indicates the specific implementation of a processor.
- Thread Identification Register (TIR) - Indicates the specific instance of a thread within in a processor.
- Processor Identification Register (PIR) - Indicates the specific instance of a processor in a multiprocessor system.
- Guest Processor Identification Register (GPIR) - Indicates the specific instance of a processor in a multiprocessor system for the guest state.

- Thread Enable Register (TENS, TENC) - Controls the thread run state.
- Thread Enable Status Register (TENSr) - Indicates the thread run state.
- External Process ID Registers (EPLC, EPSC) - Alternate PID for loads, stores, and cache operations.
- Core Configuration Register 0 (CCR0) - Controls specific processor functions, such as run controls.
- Core Configuration Register 1 (CCR1) - Controls specific processor functions, such as thread wakeup controls.
- Core Configuration Register 2 (CCR2) - Controls various other processor functions.
- Instruction Unit Configuration Register 0 (IUCR0) - Contains various configuration options for the instruction unit.
- Instruction Unit Configuration Register 1 (IUCR1) - Contains various configuration options for the instruction unit.
- Execution Unit Configuration Register 0 (XUCR0) - Contains various configuration options for the execution unit.
- Execution Unit Configuration Register 1 (XUCR1) - Contains various configuration options for the execution unit.
- Execution Unit Configuration Register 2 (XUCR2) - Contains various configuration options for the execution unit.
- Program Priority Register (PPR32) - Controls thread priority.

2.11.1 Special Purpose Registers General (SPRG0–SPRG8)

SPRG0 through SPRG8 are provided for general purpose, system-dependent software use. One common system usage of these registers is as temporary storage locations. For example, a routine might save the contents of a GPR to an SPRG and later restore the GPR from it. This is faster than a save/restore to a memory location. These registers are written using **mtspr** and read using **mfspr**.

- SPRG0 through SPRG2 These 64-bit registers can be accessed only in supervisor mode. Access to these registers when in guest state is mapped to GSPRG0 through GSPRG2.
- SPRG3 These 64-bit registers can be written only in supervisor mode. These registers can be read in supervisor and user modes. Access to these registers when in guest state is mapped to GSPRG3.
- SPRG4 through SPRG7 These 64-bit registers can be written only in supervisor mode. These registers can be read in supervisor and user modes.
- SPRG8 These 64-bit registers can be accessed only in supervisor mode.

Table 46. SPRG0 Register

Register Short Name:	SPRG0	Read Access:	Priv
Decimal SPR Number:	272	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	

Guest Supervisor Mapping:	GSPRG0	Scan Ring:	func
Bit(s):	Field Name:	Init	Description
0:63	SPRG0	0x0	<u>Software Special Purpose Register 0</u> A SPR for software use which has no defined functionality

Table 47. SPRG1 Register

Register Short Name:	SPRG1	Read Access:	Priv
Decimal SPR Number:	273	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GSPRG1	Scan Ring:	func
Bit(s):	Field Name:	Init	Description
0:63	SPRG1	0x0	<u>Software Special Purpose Register 1</u> A SPR for software use which has no defined functionality

Table 48. SPRG2 Register

Register Short Name:	SPRG2	Read Access:	Priv
Decimal SPR Number:	274	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GSPRG2	Scan Ring:	func
Bit(s):	Field Name:	Init	Description
0:63	SPRG2	0x0	<u>Software Special Purpose Register 2</u> A SPR for software use which has no defined functionality

Table 49. SPRG3 Register

Register Short Name:	SPRG3	Read Access:	Priv/Any
Decimal SPR Number:	275/259	Write Access:	Priv/None
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GSPRG3	Scan Ring:	func
Bit(s):	Field Name:	Init	Description
0:63	SPRG3	0x0	<u>Software Special Purpose Register 3</u> A SPR for software use which has no defined functionality

Table 50. SPRG4 Register

Register Short Name:	SPRG4	Read Access:	Priv/Any
Decimal SPR Number:	276/260	Write Access:	Priv/None
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG4	0x0	<u>Software Special Purpose Register 4</u> A SPR for software use which has no defined functionality

Table 51. SPRG5 Register

Register Short Name:	SPRG5	Read Access:	Priv/Any
Decimal SPR Number:	277/261	Write Access:	Priv/None
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG5	0x0	<u>Software Special Purpose Register 5</u> A SPR for software use which has no defined functionality

Table 52. SPRG6 Register

Register Short Name:	SPRG6	Read Access:	Priv/Any
Decimal SPR Number:	278/262	Write Access:	Priv/None
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG6	0x0	<u>Software Special Purpose Register 6</u> A SPR for software use which has no defined functionality

Table 53. SPRG7 Register

Register Short Name:	SPRG7	Read Access:	Priv/Any
Decimal SPR Number:	279/263	Write Access:	Priv/None
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG7	0x0	<u>Software Special Purpose Register 7</u> A SPR for software use which has no defined functionality

Table 54. SPRG8 Register

Register Short Name:	SPRG8	Read Access:	Hypv
Decimal SPR Number:	604	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG8	0x0	<u>Software Special Purpose Register 8</u> A SPR for software use which has no defined functionality

GSPRG0 through GSPRG2 These 64-bit registers can be accessed only in supervisor mode.

GSPRG3 These 64-bit registers can be written only in supervisor mode. These registers can be read in supervisor and user modes.

Table 55. GSPRG0 Register

Register Short Name:	GSPRG0	Read Access:	Priv
Decimal SPR Number:	368	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	GSPRG0	0x0	<u>Guest Software Special Purpose Register 0</u> A SPR for software use which has no defined functionality

Table 56. GSPRG1 Register

Register Short Name:	GSPRG1	Read Access:	Priv
Decimal SPR Number:	369	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	GSPRG1	0x0	<u>Guest Software Special Purpose Register 1</u> A SPR for software use which has no defined functionality

Table 57. GSPRG2 Register

Register Short Name:	GSPRG2	Read Access:	Priv
Decimal SPR Number:	370	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	GSPRG2	0x0	<u>Guest Software Special Purpose Register 2</u> A SPR for software use which has no defined functionality

Table 58. GSPRG3 Register

Register Short Name:	GSPRG3	Read Access:	Priv
Decimal SPR Number:	371	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	GSPRG3	0x0	<u>Guest Software Special Purpose Register 3</u> A SPR for software use which has no defined functionality

2.11.2 External Process ID Load Context (EPLC) Register

The EPLC register contains fields to provide the context for external process ID load instructions.

Register Short Name:	EPLC	Read Access:	Priv
Decimal SPR Number:	947	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	EPR	0b0	<u>External Load Context PR Bit</u> Used in place of MSR[PR] by the storage access control mechanism when an External Process ID Load instruction is executed. 0 Supervisor mode 1 User mode
33	EAS	0b0	<u>External Load Context AS Bit</u> Used in place of MSR[DS] for translation when an External Process ID Load instruction is executed. 0 Address space 0 1 Address space 1
34	EGS ^{HO}	0b0	<u>External Load Context GS Bit</u> ^{HO} Used in place of MSR[GS] for translation when an External Process ID Load instruction is executed. 0 Embedded Hypervisor state. 1 Guest state. This field is only writable in hypervisor state
35:39	///	0x0	<u>Reserved</u>
40:47	ELPID ^{HO}	0x0	<u>External Load Context Logical Process ID Value</u> ^{HO} Used in place of LPID register 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
48:49	///	0b00	<u>Reserved</u>
50:63	EPID	0x0	<u>External Load Context Process ID Value</u> Used in place of all Process ID register values for translation when an external Process ID Load instruction is executed.

2.11.3 External Process ID Store Context (EPSC) Register

The EPSC register contains fields to provide the context for External Process ID store instructions. The field encoding is the same as the EPLC register.

Register Short Name:	EPSC	Read Access:	Priv
Decimal SPR Number:	948	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	EPR	0b0	<u>External Store Context PR Bit</u> Used in place of MSR[PR] by the storage access control mechanism when an External Process ID Store instruction is executed. 0 Supervisor mode 1 User mode
33	EAS	0b0	<u>External Store Context AS Bit</u> Used in place of MSR[DS] for translation when an External Process ID Store instruction is executed. 0 Address space 0 1 Address space 1
34	EGS ^{HO}	0b0	<u>External Store Context GS Bit</u> ^{HO} Used in place of MSR[GS] for translation when an External Process ID Store instruction is executed. 0 Embedded Hypervisor state. 1 Guest state. This field is only writable in hypervisor state
35:39	///	0x0	<u>Reserved</u>
40:47	ELPID ^{HO}	0x0	<u>External Store Context Logical Process ID Value</u> ^{HO} Used in place of LPID register value for load translation when an External PID Store instruction is executed. Compared with TLB[TLPID] during translation. This field is only writable in hypervisor state
48:49	///	0b00	<u>Reserved</u>
50:63	EPID	0x0	<u>External Store Context Process ID Value</u> Used in place of all Process ID register values for translation when an external Process ID Store instruction is executed.

2.12 Privileged Modes

The Power ISA architecture defines two operating “states” or “modes”: supervisor (privileged) and user (nonprivileged). Which mode the processor is operating in is controlled by MSR[PR]. When MSR[PR] is 0, the processor is in supervisor mode and can execute all instructions and access all registers, including privileged ones. When MSR[PR] is 1, the processor is in user mode and can only execute nonprivileged instructions and access nonprivileged registers. An attempt to execute a privileged instruction or to access a privileged register while in user mode causes a privileged instruction exception type of program interrupt to occur.

Note that the name “PR” for the MSR field refers to an historical alternative name for user mode, which is “problem state.” Hence the value 1 in the field indicates “problem state,” and not “privileged” as one might expect.

MSR[GS]	MSR[PR]	Mode
0	0	Hypervisor
0	1	User
1	0	Guest Supervisor
1	1	Guest User

2.12.1 Privileged Instructions

An instruction that is hypervisor privileged must be in the hypervisor state (MSR[GS,PR] = 0b00) to successfully execute. If executed from guest privileged state (MSR[GS,PR] = 0b10), an embedded hypervisor privilege exception occurs. A register that is hypervisor privileged must be in the hypervisor state (MSR[GS,PR] = 0b00) to be accessed. If accessed from guest privileged state (MSR[GS,PR] = 0b10), an embedded hypervisor privilege exception occurs.

All the instructions listed in *Table 2-59* are privileged and cannot be executed in user mode; some instructions are also hypervisor privileged and must be executed in hypervisor mode.

Table 59. Privileged Instructions

Instruction	Hypervisor Privileged	Instruction	Hypervisor Privileged	Instruction	Hypervisor Privileged
dcbfep		ici		rfgi	
dcbi		icswepx[.]		stbepx	
dcbstep		lbepx		stdepex	
dcbtep		ldepex		stfdepex	
dcbtstep		lfdepex		sthepx	
dcbzep		lhexp		stwepx	
dci		lwepx		tlbilx	EPCR[DGDMI] = 1
ehpriv				tlbivax	Yes
eratilx	Yes	mfmsr		tlbre	Yes
erativax	Yes	mfspr ¹	Yes ²	tlbsrx.	EPCR[DGDMI] = 1
eratre	Yes	msgclr	Yes	tlbsx	Yes
eratsrx[.]	Yes	msgsnd	Yes	tlbsync	Yes
eratsx[.]	Yes			tlbwe ³	EPCR[DGDMI] = 1
eratwe	Yes	mtmsr		tlbwec ³	EPCR[DGDMI] = 1
icbiep		mtspr ¹	Yes ²	wrtee	
		rfci	Yes	wrteei	
		rfi			
		rfmci	Yes		

1. Applies to any SPR number with SPRN₅ = 1. See *Privileged SPRs* on page 118.
2. Applies to SPR numbers listed as hypervisor privileged. See *Table 14-1 Register Summary* on page 522.
3. This instruction is hypervisor privileged when MSR[GS] = 1 and TLB0CFG[GTWE] = 0.

2.12.1.1 Cache Locking Instructions

The cache locking instructions (**dcblic**, **dcbtlic**, **dcbtstlic**, **icblic**, **icbtlic**) are special and produce exceptions according to the following expression:

```

if MSRP[UCLEP]=1 AND MSR[GS]=1
  if MSR[PR]=1
    Cache Locking Type Data Storage Interrupt
  else
    Embedded Hypervisor Privilege Interrupt
    
```

```

endif
else // MSR[UCLEP]=0 OR MSR[GS] = 0
  if MSR[PR]=1 and MSR[UCLE]=0
    Cache Locking Type Data Storage Interrupt
  endif
endif
end

```

2.12.2 Privileged SPRs

Most SPRs are privileged. The only defined nonprivileged SPRs are LR, CTR, XER, VRSAVE, SPRG3 - 7 (read access only), TBU (read access only), and TBL (read access only). The A2O Core also treats all SPR numbers with a 1 in bit 5 of the SPRN field as privileged, whether the particular SPR number is defined or not. Thus, the core causes a privileged instruction exception type of program interrupt on any attempt to access such an SPR number while in user mode. In addition, the core causes an illegal instruction exception type of program interrupt on any attempt to access while in user mode an undefined SPR number with a 0 in SPRN₅. On the other hand, the result of attempting to access an undefined SPR number in supervisor mode is undefined, regardless of the value in SPRN₅.

2.13 Speculative Accesses

The Power ISA Architecture permits implementations to perform speculative accesses to memory, either for instruction fetching or for data loads. A speculative access is defined as any access that is not required by the sequential execution model (SEM).

For example, the A2O Core speculatively prefetches instructions down the predicted path of a conditional branch; if the branch is later determined to not go in the predicted direction, the fetching of the instructions from the predicted path is not required by the SEM and thus is speculative. The A2 core always executes load instructions in program order; load instructions are never speculative.

The architecture provides two mechanisms for protecting against errant accesses to such “non-well-behaved” memory addresses. The first is the guarded (G) storage attribute, and protects against speculative data accesses. The second is the execute permission mechanism, which protects against speculative instruction fetches. Both of these mechanisms are described in *Memory Management* on page 169.

2.14 Synchronization

The A2O Core supports the synchronization operations of the Power ISA architecture. There are three kinds of synchronization defined by the architecture, each of which is described in the following sections.

2.14.1 Context Synchronization

The context of a program is the environment in which the program executes. For example, the mode (user or supervisor) is part of the context, as are the address translation space and storage attributes of the memory pages being accessed by the program. Context is controlled by the contents of certain registers and other resources, such as the MSR and the translation lookaside buffer (TLB).

Under certain circumstances, it is necessary for the hardware or software to force the synchronization of a program's context. Context synchronizing operations include all interrupts except machine check, as well as the **isync**, **sc**, **rfi**, **rfdi**, and **rfmci** instructions. Context synchronizing operations satisfy the following requirements:

1. The operation is not initiated until all instructions preceding the operation have completed to the point at which they have reported any and all exceptions that they will cause.
2. All instructions *preceding* the operation must complete in the context in which they were initiated. That is, they must not be affected by any context changes caused by the context synchronizing operation or by any instructions *after* the context synchronizing operation.
3. If the operation is the **sc** instruction (which causes a system call interrupt) or is itself an interrupt, the operation is not initiated until no higher priority interrupt is pending (see *CPU Interrupts and Exceptions* on page 277).
4. All instructions that *follow* the operation must be refetched and executed in the context that is established by the completion of the context synchronizing operation and all of the instructions that *preceded* it.

Note that context synchronizing operations do not force the completion of storage accesses, nor do they enforce any ordering amongst accesses before and/or after the context synchronizing operation. If such behavior is required, then a storage synchronizing instruction must be used (see *Storage Ordering and Synchronization* on page 120).

Also note that, architecturally, machine check interrupts are not context synchronizing. Therefore, an instruction that *precedes* a context synchronizing operation can cause a machine check interrupt *after* the context synchronizing operation occurs and additional instructions have completed. For the A2O Core, this can only occur with data machine check exceptions, and not instruction machine check exceptions.

The following scenarios use pseudocode examples to illustrate the effects of context synchronization. Subsequent text explains how software can further guarantee "storage ordering."

1. Consider the following self-modifying code instruction sequence:

```
stw XYZ    Store to caching inhibited address XYZ.
isync
XYZ        Fetch and execute the instruction at address XYZ.
```

In this sequence, the **isync** instruction does not guarantee that the XYZ instruction is fetched after the store has occurred to memory. There is no guarantee which XYZ instruction will execute; either the old version or the new (stored) version might.

2. Now consider the required self-modifying code sequence:

```
stw        Write new instruction to data cache.
dcbst      Push the new instruction from the data cache to memory.
msync      Guarantee that dcbst completes before subsequent instructions begin.
icbi       Invalidate old copy of instruction in instruction cache.
msync      Guarantee that icbi completes before subsequent instructions begin.
isync      Force context synchronization, discarded instructions and refetch; fetch of
           stored instruction guaranteed to get new value.
```

3. This example illustrates the use of **isync** with context changes to the debug facilities

```
mtdbcr0    Enable the instruction address compare (IAC) debug event.
isync      Wait for the new Debug Control Register 0 (DBCR0) context to be established.
XYZ        This instruction is at the IAC address; an isync is necessary to guarantee that the
```

IAC event is recognized on the execution of this instruction; without the **isync**, the XYZ instruction might be prefetched and dispatched to execution before recognizing that the IAC event has been enabled.

2.14.2 Execution Synchronization

Execution synchronization is a subset of context synchronization. An execution synchronizing operation satisfies the first two requirements of context synchronizing operations, but not the latter two. That is, execution synchronizing operations guarantee that preceding instructions execute in the “old” context, but do not guarantee that subsequent instructions operate in the “new” context. An example of a scenario requiring execution synchronization would be just before the execution of a TLB-updating instructions (such as **tlbwe**). An execution synchronizing instruction should be executed to guarantee that all preceding storage access instructions have performed their address translations before executing **tlbwe** to invalidate an entry that might be used by those preceding instructions.

There are four execution synchronizing instructions: **mtmsr**, **wrtee**, **wrteei**, and **msync**. Of course, all context synchronizing instruction are also implicitly execution synchronizing, because context synchronization is a superset of execution synchronization.

Note that the Power ISA imposes additional requirements on updates to MSR[EE] (the external interrupt enable bit). Specifically, if an **mtmsr**, **wrtee**, or **wrteei** instruction sets MSR[EE] = 1, and an external input, decremter, or fixed interval timer exception is pending, the interrupt must be taken before the instruction that follows the MSR[EE]-updating instruction is executed. In this sense, these MSR[EE]-updating instructions can be thought of as being context synchronizing with respect to the MSR[EE] bit, in that it guarantees that subsequent instructions execute (or are prevented from executing and an interrupt taken) according to the new context of MSR[EE].

2.14.3 Storage Ordering and Synchronization

Storage synchronization enforces ordering between storage access instructions executed by the A2O Core. There are two storage synchronizing instructions: **msync** and **mbar**. The Power ISA defines different ordering requirements for these two instructions, but the A2O Core implements them in an identical fashion. Architecturally, **msync** is the “stronger” of the two, and is also execution synchronizing, whereas **mbar** is not.

The **mbar** instruction acts as a barrier between all storage access instructions executed before the **mbar** and all those executed after the **mbar**. That is, **mbar** ensures that all of the storage accesses initiated by instructions before the **mbar** are performed with respect to the memory subsystem before any of the accesses initiated by instructions after the **mbar**. However, **mbar** does not prevent subsequent instructions from executing (nor even from completing) before the completion of the storage accesses initiated by instructions before the **mbar**.

The **msync** instruction, on the other hand, does guarantee that all preceding storage accesses have actually been performed with respect to the memory subsystem before the execution of any instruction after the **msync**. Note that this requirement goes beyond the requirements of mere execution synchronization in that execution synchronization does not require the completion of preceding storage accesses.

The following two examples illustrate the distinctive use of **mbar** versus **msync**.

```

stw      Store data to an I/O device.
msync    Wait for store to actually complete.
stw
```

In this example, the **mtdcr** is reconfiguring the I/O device in a manner that would cause the preceding store instruction to fail, were the **mtdcr** to change the device before the completion of the store. Because **mtdcr** is not a storage access instruction, the use of **mbar** instead of **msync** does not guarantee that the store is performed before letting the **mtdcr** reconfigure the device. It only guarantees that subsequent storage accesses are not performed to memory or any device before the earlier store.

Now consider this next example:

stb X	Store data to an I/O device at address X, causing a status bit at address Y to be reset.
mbar	Guarantee preceding store is performed to the device before any subsequent storage accesses are performed.
lbz Y	Load status from the I/O device at address Y.

Here, **mbar** is appropriate instead of **msync**, because all that is required is that the store to the I/O device happens before the load does, but not that other instructions subsequent to the **mbar** will not get executed before the store.

2.15 Software Transactional Memory Acceleration

2.15.1 Summary

The A2 core is augmented with support for three new instructions: **ldawx** (load double-word and set watch indexed), **wchkall** (watch check all), and **wclr** (watch clear). These instructions are used to control a monitoring facility that detects writes by other threads to watched memory locations. For more information, see *Section 12.4 Software Transactional Memory Instructions* on page 503.

A thread can execute a sequence of **ldawx** instructions, setting watches for multiple memory locations, with one or more **wchkall** operations to detect whether any of its watched locations have potentially been written by another thread. The set of watches can then be cleared with a **wclr** instruction. If the number of watches exceeds the capacity of the watch facility, subsequent **wchkall** instructions will conservatively indicate that one of the watched cache blocks has been written by another thread.

2.15.2 Implementation

Three user-level instructions interact with a set of watch bits associated with the L1 D-cache. One bit per thread per cache block is added to the L1 D-cache to capture the common-case working set of watches.

The **ldawx** and **wchkall** instructions are performance critical. These instructions are fully-pipelined with performance similar to conventional load instructions.

The **wclr** instruction is less performance critical. When performed with a nonzero EA, the **wclr** instruction should be performed sequentially with respect to other memory operations to the same location. When performed with an EA of 0, **wclr** must simply complete before any subsequent **ldawx** instruction is able to complete (that is, gating **ldawx** instructions at dispatch pending a **wclr** instruction should be sufficient). When **wclr** is executed with EA = 0, a signal is raised to the L1 D-cache indicating that all of the watch bits should be flash cleared, and the watchlost sticky bit for the thread performing the **wclr** should be set to the L value from the instruction.

When **wchkall** is executed, the watchlost sticky bit (part of the L1 D-cache, see *Section 2.15.2.1*) corresponding to the executing thread is probed, and the CR is updated appropriately.

2.15.2.1 L1 D-Cache

Four bits are added per cache block, representing the set of watches that exist for that block corresponding to each thread. If not already available, the A2 core needs to provide a thread identifier associated with each request to the L1 D-cache to control the watch bits affected by each command. The L1 D-cache controller also maintains an additional “sticky” bit per thread denoted watchlost, which reflects whether any watches have been lost since that thread last reset its watchlost bit.

A regular load that misses the L1 allocates the line in L1D and resets all watchbits for that line. For each write operation performed by the processor, the watch bits for all threads other than the writing thread are reset for that particular block. For example, a write by thread 0 to a block whose watch bits are all set to 1 will result in all of the watch bits being reset to 0, aside from the watch bit corresponding to thread 0. A line's invalidation from the L1 (due to **dcbf**, **dcbz**, **dci**, **larx/stcx**, **icswx**, multi-hit error, parity error, back invalidate, **icsw[ep]x**, or capacity replacement) also result in the resetting of a block's watch bits for all threads. Any other requests to a block (from the processor or coherence interface) should have no effect on the watch bits, so long as the block remains valid in a readable state in the L1 D-cache. When a watch bit is reset from 1 to 0 for a thread, the sticky “watch lost” bit for that thread is updated to 1. A **dci** instruction sets the sticky “watch lost” bit for all threads regardless of any watch bits reset from 1 to 0.

The L1 D-cache must also provide an interface for flash clearing all of the watch bits for a designated thread and setting/resetting the designated thread's watchlost sticky bit. Watch bits and sticky bits for threads other than the designated thread should remain untouched.

2.15.3 Watch Operation Ordering Requirements

A **ldawx** by a processor P1 is performed with respect to any processor or mechanism P2 when the value and watchbit to be returned by the **ldawx** can no longer be changed by an operation by P2. A **wchkall** instruction by P1 is performed with respect to P2 when an operation by P2 can no longer affect the state of any watches summarized by the **wchkall** condition value. Watch values returned by **ldawx** and **wchkall** are consistent with the data protected by those watches. The ordering of **ldawx** and **wchkall** instructions with respect to the performance of prior operations (for example, outstanding writes) is controlled by the same rules governing ordinary loads as specified in *Power ISA Book II*, section 1.7.1.

Implementations are free to reorder **ldawx** instructions with respect to other memory operations (including other **ldawx** instructions) subject to data dependencies. Reordering of the **wclr** and **wchkall** instructions with respect to other instructions that manipulate watches is disallowed.

2.15.4 Impact on Existing Software

None.



3. FU Programming Model

The programming model of the product_name describes how the following features and operations appear to programmers:

- Storage addressing, including storage operands, effective address calculation, and data storage addressing modes, starting on page 123
- Floating-point exceptions, starting on page 125
- Floating-point registers, starting on page 125
- Floating-point data formats, starting on page 129
- Floating-point execution models, starting on page 136
- Floating-point instructions, starting on page 139

The Power ISA Architecture specification (referred to as Book III-E) specifies that the floating-point unit (FU) implements a floating-point system as defined in ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic* (referred to as IEEE 754). However, the architecture requires software support to conform fully with the standard. IEEE 754 defines certain required operations (addition, subtraction, and so on). The term “floating-point operation” is used to refer to one of these required operations or to the operation performed by one of the multiply-add or reciprocal estimate instructions. All floating-point operations conform to the IEEE standard. All floating-point operations produce the same results regardless of the value of IEEE mode (NI) bit.

3.1 Storage Addressing

Floating-point storage accesses use the same uniform 64-bit effective address (EA) space as all A2 core storage accesses. Effective addresses are expanded into virtual addresses and then translated to 42-bit (4 TB) real addresses by the memory management unit (MMU) of the processor core.

Note: In 32-bit mode, the A2 core forces bits 0:31 of the calculated 64-bit effective address to zeros. Therefore, for a translation to hit in 32-bit mode, software needs to set the effective address upper bits to zero in the ERATs and the TLB.

The product_name generates an effective address whenever it executes a load/store instruction.

3.1.1 Storage Operands

Bytes in storage are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

The data storage operands accessed by the product_name load/store instructions can be words (4 bytes or 32 bits) or doublewords (8 bytes or 64 bits). The address of a storage operand is the address of its first byte (that is, of its lowest-numbered byte). Byte ordering can be either big endian or little endian, as controlled by the endian (E) storage attribute.

Operand length is implicit for each scalar storage access instruction. The operand of such a scalar storage 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 storage operand is said to be aligned if it is aligned at its natural boundary; otherwise, it is said to be unaligned.

Data storage operands for storage access instructions have the characteristics shown in *Table 3-1*.

Table 1. Data Operand Definitions

Storage Access Instruction Type	Operand Length	A[60:63] if aligned
Word	4 bytes	0bxx00
Doubleword	8 bytes	0bx000

Note: An “x” in an address bit position indicates that the bit can be 0 or 1 regardless of the state of other bits in the address.

The alignment of the operand effective address of some storage access instructions can affect performance and in some cases can cause an alignment exception to occur. For such storage access instructions, the best performance is obtained when the storage operands are naturally aligned. *Table 2-11* on page 87 summarizes the effects of alignment on those storage access instruction types for which such effects exist.

3.1.2 Effective Address Calculation

For a storage access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address of $2^{64} - 1$ for 64-bit mode or $2^{32} - 1$ in 32-bit mode (that is, the storage operand itself crosses the maximum address boundary), the result of the operation is undefined, as specified by the architecture. The A2 core performs the operation as if the storage operand wrapped around from the maximum effective address to effective address 0. Software, however, should not depend upon this behavior, so that can be ported to other implementations that do not handle such accesses in the same manner. Software should ensure that no data storage operands cross the maximum address boundary.

Note: Because instructions are words and because the effective addresses of instructions are always implicitly on word boundaries, an instruction storage operand cannot cross any word boundary, including the maximum address boundary.

Effective address arithmetic, which calculates the starting address for storage operands, wraps around from the maximum address to address 0, for all effective address computations except next sequential instruction fetching.

3.1.3 Data Storage Addressing Modes

The product_name supports the following data storage addressing modes.

- Base + displacement (D-mode) addressing mode:

The 16-bit D field is sign-extended to 64 bits and added to the contents of the GPR designated by RA, or to zero if RA = 0. The 64-bit sum forms the effective address of the data storage operand.

Note: In 32-bit mode, the A2 core forces bits 0:31 of the calculated 64-bit effective address to zeros.

- Base + index (X-mode) addressing mode:

The contents of the GPR designated by RB (or the value 0 for **lswi** and **stswi**) are added to the contents of the GPR designated by RA, or to zero if RA = 0;

3.2 Floating-Point Exceptions

Each floating-point exception, and each category of invalid operation exception, is associated with an exception bit in the **FPSCR**. The following floating-point exceptions are detected by the processor. The associated FPSCR fields are listed with each exception and invalid operation exception category.

- Invalid operation exception (VX)

Table 2. Invalid Operation Exception Categories

Category	FPSCR Field
SNaN	VXSNAN
Infinity – Infinity	VXISI
Infinity ÷ Infinity	VXIDI
Zero ÷ Zero	VXZDZ
Infinity × Zero	VXIMZ
Invalid Compare	VXVC
Software Request	VXSOFT
Invalid Square Root	VXSQRT
Invalid Integer Convert	VXCVI

- Zero divide exception (ZX)
- Overflow exception (OX)
- Underflow exception (UX)
- Inexact exception (XI)

Each floating-point exception also has a corresponding enable bit in the FPSCR. See *Floating-Point Status and Control Register Instructions* on page 147 for descriptions of these exception and enable bits and *FU Interrupts and Exceptions* on page 355 for a detailed discussion of floating-point exceptions including the effects of the FPSCR enable bits.

3.3 Floating-Point Registers

This section provides an overview of the register types implemented in the product_name. Detailed descriptions of the floating-point registers are provided within the chapters covering the functions with which they are associated. An alphabetical summary of all registers, including bit definitions, is provided in *Register Summary* on page 521.

Certain bits in some registers are reserved and thus not necessarily implemented. For all registers with fields marked as reserved, these reserved fields should be written as 0 and read as undefined. The recommended coding practice is to perform the initial write to a register with reserved fields set to 0, and to perform all subsequent writes to the register using a read-modify-write strategy. That is, read the register; use logical instructions to alter defined fields, leaving reserved fields unmodified; and write the register.

Each register is classified as being of a particular type, as characterized by the specific instructions used to read and write registers of that type. The registers contained within the A2 core are defined by Book III-E.

3.3.1 Register Types

The product_name core provides two types of registers, Floating-Point Registers (FPRs) and the FPSCR. Each type is characterized by the instructions used to read and write the registers. The following subsections provide an overview of each register type and the instructions associated with them.

3.3.1.1 Floating-Point Registers (FPR0–FPR31)

The product_name provides 32 Floating-Point Registers (FPRs), each 64 bits wide. In any cycle, the FPR file can read the operands for a store instruction and an arithmetic instruction or write the data from a load instruction and the result of an arithmetic instruction.

0		63
---	--	----

Table 3. Floating-Point Registers (FPR0–FPR31)

Bits	Field Name	Description
0:63		Floating-Point Register Data

The FPRs are numbered FPR0–FPR31. The floating-point instruction formats provide 5-bit fields to specify the FPRs used as operands in the execution of the associated instructions.

Each FPR contains 64 bits that support the floating-point double format. All instructions that interpret the contents of an FPR as a floating-point value uses the floating-point double format for this interpretation.

The computational instructions, and the move and select instructions, operate on data located in FPRs. With the exception of the compare instructions, they place the result value into an FPR and optionally place status information into the Condition Register (CR).

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

Some floating-point instructions update the FPSCR and CR explicitly. Some of these instructions move data to and from an FPR to the FPSCR or from the FPSCR to an FPR.

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 not, the result placed into the target FPR and the setting of status bits in the FPSCR are undefined.

3.3.1.2 Floating-Point Status and Control Register (FPSCR)

The FPSCR controls the handling of floating-point exceptions and records status resulting from the floating-point operations.

Table 4. Floating-Point Status and Control Register (FPSCR) (Sheet 1 of 3)

Bits	Field Name	Description
0:28		<p><u>Reserved</u></p> <p>Note: FPSCR[28] is reserved for extension of the DRN field; therefore DRN can be set by using the mtfsfi instruction to set the rounding mode.</p>
29:31	DRN	<p><u>DFP Rounding Control</u></p> <p>000 Round to nearest, ties to even. 001 Round toward zero. 010 Round toward +infinity. 011 Round toward -infinity. 100 Round to nearest, ties away from 0. 101 Round to nearest, ties toward 0. 110 Round to away from zero. 111 Round to prepare for shorter precision. See Section 5.5.2 in PowerISA Version 2.06B.</p>
32	FX	<p><u>Floating-Point Exception Summary</u></p> <p>0 No FPSCR exception bits changed from 0 to 1. 1 At least one FPSCR exception bit changed from 0 to 1. All floating-point instructions, except mtfsfi and mtfsf, implicitly set this field to 1 if the instruction causes any floating-point exception bits in the FPSCR to change from 0 to 1. mcrfs, mtfsfi, mtfsf, mtfsb0, and mtfsb1 can alter this field explicitly.</p>
33	FEX	<p><u>Floating-Point Enabled Exception Summary</u></p> <p>The OR of all the floating-point exception fields masked by their respective enable fields. mcrfs, mtfsfi, mtfsf, mtfsb0, and mtfsb1 cannot alter this field explicitly.</p>
34	VX	<p><u>Floating-Point Invalid Operation Exception Summary</u></p> <p>The OR of all the invalid operation exception fields. mcrfs, mtfsfi, mtfsf, mtfsb0, and mtfsb1 cannot alter this field explicitly.</p>
35	OX	<p><u>Floating-Point Overflow Exception</u></p> <p>0 A floating-point overflow exception did not occur. 1 A floating-point overflow exception occurred. See <i>Overflow Exception</i> on page 362.</p>
36	UX	<p><u>Floating-Point Underflow Exception</u></p> <p>0 A floating-point underflow exception did not occur. 1 A floating-point underflow exception occurred. See <i>Underflow Exception</i> on page 363.</p>
37	ZX	<p><u>Floating-Point Zero Divide Exception</u></p> <p>0 A floating-point zero divide exception did not occur. 1 A floating-point zero divide exception occurred. See <i>Zero Divide Exception</i> on page 361.</p>
38	XX	<p><u>Floating-Point Inexact Exception</u></p> <p>0 A floating-point inexact exception did not occur. 1 A floating-point inexact exception occurred. This field is a sticky version of FPSCR[FI]. The following rules describe how a given instruction sets this field. If the instruction affects FPSCR[FI], the new value of this field is obtained by ORing the old value of this field with the new value of FPSCR[FI]. If the instruction does not affect FPSCR[FI], the value of this field is unchanged.</p>

Table 4. Floating-Point Status and Control Register (FPSCR) (Sheet 2 of 3)

Bits	Field Name	Description
39	VXSNAN	<u>Floating-Point Invalid Operation Exception (SNaN)</u> 0 A floating-point invalid operation exception (VXSNAN) did not occur. 1 A floating-point invalid operation exception (VXSNAN) occurred. See <i>Invalid Operation Exception</i> on page 359.
40	VXISI	<u>Floating-Point Invalid Operation Exception ($\infty - \infty$)</u> 0 A floating-point invalid operation exception (VXISI) did not occur. 1 A floating-point invalid operation exception (VXISI) occurred. See <i>Invalid Operation Exception</i> on page 359.
41	VXIDI	<u>Floating-Point Invalid Operation Exception ($\infty \div \infty$)</u> 0 A floating-point invalid operation exception (VXIDI) did not occur. 1 A floating-point invalid operation exception (VXIDI) occurred. See <i>Invalid Operation Exception</i> on page 359.
42	VXZDZ	<u>Floating-Point Invalid Operation Exception ($0 \div 0$)</u> 0 A floating-point invalid operation exception (VXZDZ) did not occur. 1 A floating-point invalid operation exception (VXZDZ) occurred. See <i>Invalid Operation Exception</i> on page 359.
43	VXIMZ	<u>Floating-Point Invalid Operation Exception ($\infty \times 0$)</u> 0 A floating-point invalid operation exception (VXIMZ) did not occur. 1 A floating-point invalid operation exception (VXIMZ) occurred. See <i>Invalid Operation Exception</i> on page 359.
44	VXVC	<u>Floating-Point Invalid Operation Exception (Invalid Compare)</u> 0 A floating-point invalid operation exception (VXVC) did not occur. 1 A floating-point invalid operation exception (VXVC) occurred. See <i>Invalid Operation Exception</i> on page 359.
45	FR	<u>Floating-Point Fraction Rounded</u> The last arithmetic or rounding and conversion instruction either produced an inexact result during rounding or caused a disabled overflow exception. See <i>Rounding</i> on page 135. This bit is not sticky.
46	FI	<u>Floating-Point Fraction Inexact</u> The last arithmetic or rounding and conversion instruction either produced an inexact result during rounding or caused a disabled overflow exception. See <i>Rounding</i> on page 135. This bit is not sticky. See the definition of FPSCR[XX] regarding the relationship between FPSCR[FI] and FPSCR[XX].
47	FPRF	<u>Floating-Point Result Flag (FPRF)</u>
48	FL	<u>Floating-Point Less Than or Negative</u>
49	FG	<u>Floating-Point Greater Than or Positive</u>
50	FE	<u>Floating-Point Equal to Zero</u>
51	FU	<u>Floating-Point Unordered or NaN</u>
52		Reserved
53	VXSOFT	<u>Floating-Point Invalid Operation Exception (Software Request)</u> 0 A floating-point invalid operation exception (software request) did not occur. 1 A floating-point invalid operation exception (software request) occurred. See <i>Invalid Operation Exception</i> on page 359.
54	VXSQRT	<u>Floating-Point Invalid Operation Exception (Invalid Square Root)</u> 0 A floating-point invalid operation exception (invalid square root) did not occur. 1 A floating-point invalid operation exception (invalid square root) occurred. See <i>Invalid Operation Exception</i> on page 359.

Table 4. Floating-Point Status and Control Register (FPSCR) (Sheet 3 of 3)

Bits	Field Name	Description
55	VXCVI	<u>Floating-Point Invalid Operation Exception (Invalid Integer Convert)</u> 0 A floating-point invalid operation exception (invalid integer convert) did not occur. 1 A floating-point invalid operation exception (invalid integer convert) occurred. See <i>Invalid Operation Exception</i> on page 359.
56	VE	<u>Floating-Point Invalid Operation Exception Enabled</u> 0 Floating-point invalid operation exceptions are disabled. 1 Floating-point invalid operation exceptions are enabled.
57	OE	<u>Floating-Point Overflow Exception Enable</u> 0 Floating-point overflow exceptions are disabled. 1 Floating-point overflow exceptions are enabled.
58	UE	<u>Floating-Point Underflow Exception Enable</u> 0 Floating-point underflow exceptions are disabled. 1 Floating-point underflow exceptions are enabled.
59	ZE	<u>Floating-Point Zero Divide Exception Enable</u> 0 Floating-point zero divide exceptions are disabled. 1 Floating-point zero divide exceptions are enabled.
60	XE	<u>Floating-Point Inexact Exception Enable</u> 0 Floating-point inexact exceptions are disabled. 1 Floating-point inexact exceptions are enabled.
61	NI	<u>Floating-Point Non-IEEE Mode</u> 0 Non-IEEE mode is disabled. 1 Non-IEEE mode is enabled. If FPSCR[NI] = 1, the remaining FPSCR bits might have meanings other than those given in this document, and the results of floating-point operations need not conform to the IEEE standard. If the IEEE-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. The behavior when FPSCR[NI] = 1 can vary from one implementation to another
62:63	RN	<u>Floating-Point Rounding Control</u> 00 Round to nearest. 01 Round toward zero. 10 Round toward +infinity. 11 Round toward -infinity. See <i>Rounding</i> on page 135.

Programming Note: All floating-point operations conform to the IEEE standard. All floating-point operations produce the same results regardless of the value of IEEE mode (NI) bit.

3.4 Floating-Point Data Formats

This section describes floating-point data formats, representation of floating-point values, data handling and precision, and rounding.

Floating-point values are represented in two binary fixed-length formats. Single-precision values are represented in the 32-bit single format. Double-precision values are represented in the 64-bit double format. The single format can be used for data in storage, but cannot be stored in the FPRs. The double format can be used for data in storage and for data in the FPRs. When a floating-point value is loaded from storage using a

load single instruction, it is converted to double format and placed in the target FPR. Conversely, a floating-point value stored from an FPR into storage using a store single instruction is converted to single format before being placed in storage.

The lengths of the exponent and the fraction fields differ between these two formats. The structure of the single and double formats are shown in *Table 3-5* and *Table 3-6*, respectively.

Table 5. Floating-Point Single Format

S	EXP	FRACTION	
0	1	9	31

Table 6. Floating-Point Double Format

S	EXP	FRACTION	
0	1	12	63

Values in floating-point format are composed of three fields:

Table 7. Format Fields

Field	Description
S	Sign bit
EXP	Exponent + bias
FRACTION	Fraction

If only a portion of a floating-point data item in storage is accessed, such as with a load or store instruction for a byte or halfword (or word in the case of floating-point double format), the value affected depends on whether the Power ISA embedded system is operating with big-endian or little-endian byte ordering.

3.4.1 Value Representation

Representation of numeric values in the floating-point formats consists of a sign bit (S), a biased exponent (EXP), and the fraction portion (FRACTION) of the significand. 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). Values representable within the two floating-point formats can be specified by the parameters listed in *Table 3-8*.

Table 8. IEEE 754 Floating-Point Fields (Sheet 1 of 2)

	Single	Double
Exponent Bias	+127	+1023
Maximum Exponent	+127	+1023
Minimum Exponent	-126	-1022
Field Widths (Bits)		
Sign	1	1

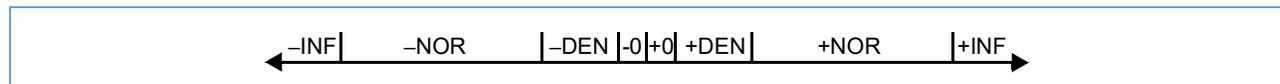
Table 8. IEEE 754 Floating-Point Fields (Sheet 2 of 2)

	Single	Double
Exponent	8	11
Fraction	23	52
Significand	24	53

The FPRs support the floating-point double format only.

The numeric and nonnumeric values representable within each of the two supported formats are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The nonnumeric values representable 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. The relative location on the real number line for each of the defined entities is shown in Figure 3-1.

Figure 1. Approximation to Real Numbers



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

Descriptions of the different floating-point values defined in the architecture follow.

3.4.2 Binary Floating-Point Numbers

These are machine-representable values used as approximations to real numbers. Three categories of numbers are supported: normalized numbers, denormalized numbers, and zero values.

3.4.2.1 Normalized Numbers

Normalized numbers (\pm NOR) have an unbiased exponent value in the range:

- 126 to 127 in single format
- 1022 to 1023 in double format

They are values in which the implied unit bit is 1. Normalized numbers are interpreted as follows:

$$\text{NOR} = (-1)^s \times 2^E \times (1.\text{fraction})$$

where s is the sign, E is the unbiased exponent, and 1.fraction is the significand, which is composed of a leading unit bit (implied bit) and a fraction part.

The ranges covered by the magnitude (M) of a normalized floating-point number are approximately equal to:

- Single format:
 $1.2 \times 10^{-38} \leq M \leq 3.4 \times 10^{38}$
- Double format:
 $2.2 \times 10^{-308} \leq M \leq 1.8 \times 10^{308}$

3.4.2.2 Denormalized Numbers

Denormalized numbers (\pm DEN) 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 follows:

$$\text{DEN} = (-1)^s \times 2^{\text{Emin}} \times (0.\text{fraction})$$

where Emin is the minimum representable exponent value (−126 for single-precision, −1022 for double-precision).

3.4.2.3 Zero Values

Zero values (± 0) 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; comparison treats +0 as equal to −0.

3.4.3 Infinities

Infinities ($\pm\infty$) are values that 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:

$$-\infty < \text{every finite number} < +\infty$$

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 *Invalid Operation Exception* on page 359.

3.4.3.1 Not a Numbers

Not a Numbers (NaNs) are values that 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, the NaN is a signalling NaN (SNaN); otherwise it is a quiet NaN (QNaN).

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 instructions except **fcmpo**, **frsp**, and **fctiw**. 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 is applied to determine the NaN with the high-order fraction bit set to 1 that is to be stored as the result.

```

if FPR(FRA) is a NaN
then FPR(FRT) ← FPR(FRA)
else if FPR(FRB) is a NaN
then if instruction is frsp
    then FPR(FRT) ← FPR(FRB)0:34 || 290
    else FPR(FRT) ← FPR(FRB)
else if FPR(FRC) is a NaN
    then FPR(FRT) ← FPR(FRC)
    else if generated QNaN
        then FPR(FRT) ← generated QNaN
    
```

If the operand specified by FRA is a NaN, that NaN is stored as the result. Otherwise, if the operand specified by FRB is a NaN (if the instruction specifies an FRB operand), that NaN is stored as the result, with the low-order 29 bits of the result set to 0 if the instruction is **frsp**. Otherwise, if the operand specified by FRC is a NaN (if the instruction specifies an FRC operand), that NaN is stored as the result. Otherwise, if a QNaN was generated due to a disabled invalid operation exception, that QNaN is stored as the result. If a QNaN is to be generated as a result, 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 must generate this QNaN (that is, 0x7FF8_0000_0000_0000).

A double-precision NaN is representable in single format if and only if the low-order 29 bits of the double-precision NaNs fraction are zero.

3.4.4 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 when the 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 **frsqrte** instruction is always positive, except that the reciprocal square root of -0 is $-\text{Infinity}$.
- The sign of the result of an **frsp**[.], or **fctiw** operation is the sign of the operand being converted.

For the multiply-add instructions, the preceding rules are applied first to the multiply operation 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).

3.4.5 Normalization and Denormalization

- The intermediate result of an arithmetic or **frsp** instruction might require normalization and/or denormalization. Normalization and denormalization do not affect the sign of the result.
- When an arithmetic or **frsp** instruction produces an intermediate result consisting of a sign bit, an exponent, and a nonzero significand with a 0 leading bit; it is not a normalized number and must be normalized before it is stored.

A number is normalized by shifting its significand left while decrementing its exponent by 1 for each bit shifted, until the leading significand bit becomes 1. The G bit and the R bit (see *Execution Model for IEEE Operations* on page 137) 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 can 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 *Underflow Exception* on page 363. These rules might 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. If any significant bits are lost in this shifting process, “loss of accuracy” has occurred (see *Underflow Exception* on page 363) and an underflow exception is signaled.

3.4.6 Data Handling and Precision

Instructions are defined to move floating-point data between the FPRs and storage. For double format data, the data is not altered during the move. For single format data, a format conversion from single to double is performed when loading from storage into an FPR. A format conversion from double to single is performed when storing from an FPR to storage. The load/store instructions do not cause floating-point exceptions.

- All computational, move, and **fsel** instructions use the floating-point double format.

Floating-point single-precision values are obtained with the following types of instruction.

- 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.

- Round to floating-point single-precision

The **frsp** 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 as a double-precision operand. For results produced by single-precision arithmetic instructions, single-precision loads, and other instances of the **frsp** instruction, this operation does not alter the value.

Programming Note: The **frsp** instruction enables 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) to single-precision floating-point values before 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 ele-

ments, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by an **frsp** instruction.

- Single-precision arithmetic instructions

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 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.

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, 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.)

When the result of a load floating-point single, **frsp**, or single-precision arithmetic instruction is stored in an FPR, the low-order 29 fraction bits are zero.

Programming Note: 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.

3.4.7 Rounding

Rounding applies to operations that have numeric operands (operands that are not infinities or NaNs). Rounding the intermediate result of such operations can cause an overflow exception, an underflow exception, or an inexact exception. The following description assumes that the operations cause no exceptions and that the result is numeric. See *Value Representation* on page 130 and *FU Interrupts and Exceptions* on page 355 for the cases not covered here. *Execution Model for IEEE Operations* on page 137 provides a detailed explanation of rounding.

The arithmetic and rounding and conversion instructions produce intermediate results that can be regarded as having infinite precision and unbounded exponent range. Such intermediate results are normalized or denormalized if required, then rounded to the target format. The final result is then placed into the target FPR in double format or in integer format, depending on the instruction.

The arithmetic and rounding and conversion instructions, which round intermediate results, set FPSCR[FR, FI]. If the fraction was incremented during rounding, FPSCR[FR] = 1; otherwise, FPSCR[FR] = 0. If the rounded result is inexact, FPSCR[FI] = 1; otherwise, FPSCR[FI] = 0.

The estimate instructions set FPSCR[FR, FI] to undefined values. The remaining floating-point instructions do not alter FPSCR[FR, FI].

FPSCR[RN] specifies one of four programmable rounding modes.

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.

Figure 3-2 shows the relation of z , $z1$, and $z2$ in this case. The following rules specify the rounding in the four modes. "LSb" means "least-significant bit."

Figure 2. Selection of $z1$ and $z2$

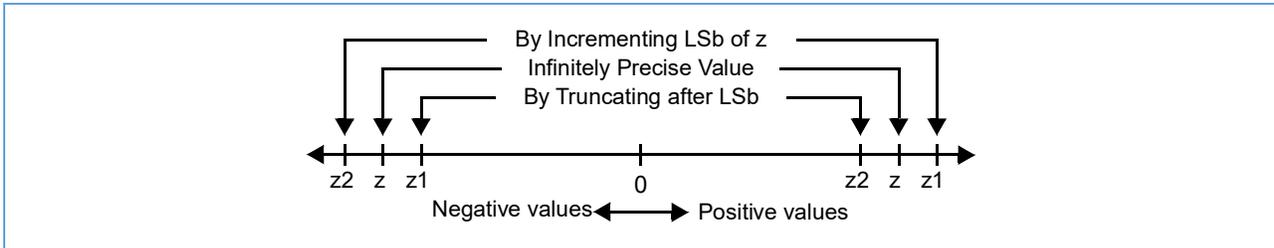


Table 3-9 describes the rounding modes.

Table 9. Rounding Modes

FPSCR[RN]	Rounding Mode	Description
00	Round to Nearest	Choose the value that is closest to z , either $z1$ or $z2$. In case of a tie, choose the one that is even (the LSb is 0).
01	Round toward Zero	Choose the smaller in magnitude ($z1$ or $z2$).
10	Round toward +Infinity	Choose $z1$.
11	Round toward -Infinity	Choose $z2$.

3.5 Floating-Point Execution Models

All implementations of this architecture must provide the equivalent of the following execution models to ensure that identical results are obtained.

Special rules are provided in the definition of the computational instructions for the infinities, denormalized numbers, and NaNs. The material in the remainder of this section applies to instructions that have numeric operands and a numeric result (that is, operands and a result that are not infinities or NaNs) and that cause no exceptions. See *Value Representation* on page 130 and *Floating-Point Exceptions* on page 355 for the cases not covered here.

Although the double format specifies an 11-bit exponent, exponent arithmetic makes use of two additional bits to avoid potential transient overflow conditions. One extra bit is required when denormalized double-precision numbers are prenormalized. The second bit is required to permit the computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is 1:

- Underflow during multiplication using a denormalized operand.
- Overflow during division using a denormalized divisor.

The IEEE standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision floating-point operations to have either (or both) single-precision or double-precision operands, but states that single-precision floating-point operations should not accept double-precision operands. Book III-E follows these guidelines: double-precision arithmetic instructions can have operands of either or both precisions, while single-precision

arithmetic instructions require all operands to be single-precision. Double-precision arithmetic instructions produce double-precision values, while single-precision arithmetic instructions produce single-precision values.

For arithmetic instructions, conversions from double-precision to single-precision must be done explicitly by software, while conversions from single-precision to double-precision are done implicitly.

3.5.1 Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. 32-bit arithmetic is similar except that the FRACTION 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-conforming significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:55 comprise the significand of the intermediate result.

Table 10. IEEE 64-Bit Execution Model

S	C	L	FRACTION	G	R	X
		0 1		52 53	54	55

The S bit is the sign bit.

The C bit is the carry bit, which captures the carry out of the significand.

The L bit is the leading unit bit of the significand, which receives the implicit bit from the operand.

The FRACTION is a 52-bit field that accepts the fraction of the operand.

The Guard (G), Round (R), and Sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for post-normalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that can appear to the low-order side of the R bit, due either to shifting the accumulator right or to other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. *Table 3-11* shows the significance of the G, R, and X bits with respect to the intermediate result (IR), the representable number next lower in magnitude (NL), and the representable number next higher in magnitude (NH).

Table 11. Interpretation of the G, R, and X Bits

G	R	X	Interpretation
0	0	0	IR is exact.
0	0	1	IR is closer to NL.
0	1	0	
0	1	1	
1	0	0	IR is midway between NL and NH.
1	0	1	IR is closer to NH.
1	1	0	
1	1	1	

After normalization, the intermediate result is rounded using the rounding mode specified by FPSCR[RN]. If rounding results in a carry into C, the significand is shifted right one position and the exponent incremented by one. This yields an inexact result and possibly also 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 *Rounding* on page 135. For rounding, the conceptual Guard, Round, and Sticky bits are defined in terms of accumulator bits. *Table 3-12* shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the IEEE execution model.

Table 12. Location of the Guard, Round, and Sticky Bits in the IEEE 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 is nonzero, then the result is inexact.

Z1 and Z2, as defined in *Rounding* on page 135, 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 1 (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 0 (result midway between closest representable values) and if the low-order bit of the result is 1, the result is incremented. Otherwise (the low-order bit of the result is 0), the result is truncated (this is the case of a tie rounded to even).
- 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 the 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 is either normalized or placed in correct denormalized form before being rounded.

3.5.2 Execution Model for Multiply-Add Type Instructions

The product_name provides a special form of instruction that performs up to three operations in one instruction (a multiplication, an addition, and a negation). With this added capability comes the special ability to produce a more exact intermediate result as input to the rounder. 32-bit arithmetic is similar except that the FRACTION field is smaller.

Multiply-add significand arithmetic is considered to be performed with a floating-point accumulator having the following format, where bits 0:106 comprise the significand of the intermediate result.

Table 13. Multiply-Add 64-Bit Execution Model

S	C	L	FRACTION	X'
	0	1		105 106

The first part of the operation is a multiplication. The multiplication 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), then the significand is shifted right one position, shifting the L bit (leading unit bit) into the most significant bit of the FRACTION and shifting the C bit (carry out) into the L bit. All 106 bits (L bit, 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 that is 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 addition is then normalized, with all bits of the addition 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.

Table 3-14 shows the positions of the Guard, Round, and Sticky bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

Table 14. Location of Guard, Round, and Sticky Bits in the 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 *Execution Model for IEEE Operations* on page 137.

If the instruction is a floating negative multiply-add or floating negative multiply-subtract, the final result is negated.

3.6 Floating-Point Instructions

Primary opcode 63 is used for the double-precision arithmetic instructions and miscellaneous instructions, such as the Floating-Point Status and Control Register Manipulation instructions. Primary opcode 59 is used for the single-precision arithmetic instructions.

The single-precision instructions for which there is a corresponding double-precision instruction have the same format and extended opcode as the corresponding double-precision instruction.

Instructions are provided to perform arithmetic, rounding, conversion, comparison, and other operations in floating-point registers; to move floating-point data between storage and these registers; and to manipulate the FPSCR explicitly.

These instructions are divided into two categories.

- Computational instructions

The computational instructions are those that perform addition, subtraction, multiplication, division, extracting the square root, rounding, conversion, comparison, and combinations of these operations. These instructions provide the floating-point operations. They place status information into the FPSCR. They are the instructions described in *Floating-Point Arithmetic Instructions* on page 144, *Floating-Point Rounding and Conversion Instructions* on page 145, and *Floating-Point Compare Instructions* on page 146.

- Noncomputational instructions

The noncomputational instructions, which perform loads and stores, move the contents of a floating-point register to another floating-point register possibly altering the sign, manipulate the FPSCR explicitly, and select a value from one of two floating-point registers based on the value in a third floating-point register. These operations are not considered floating-point operations. With the exception of the instructions that manipulate the FPSCR explicitly, they do not alter the FPSCR. Those instructions are described in *Floating-Point Status and Control Register Instructions* on page 147.

A floating-point number consists of a signed exponent and a signed significand. The quantity expressed by this number is the product of the significand and the number 2^{exponent} . Encodings are provided in the data format to represent finite numeric values, \pm infinity, and values that are Not a Number (NaN). Operations involving infinities produce results following traditional mathematical conventions. NaNs have no mathematical interpretation, but their encoding supports a variable diagnostic information field. NaNs can be used to indicate such things as uninitialized variables and can be produced by certain invalid operations.

One class of exceptions that occur during floating-point instruction execution is unique to floating-point operations: the floating-point exception. Bits set in the FPSCR indicate floating-point exceptions. They can cause an enabled exception type of program interrupt to be taken, if the proper control bits are set.

3.6.1 Instructions by Category

The floating-point instructions can be classified into computational and noncomputational categories. The computational instructions include those that perform arithmetic operations or conversions on operands. Noncomputational instructions perform loads/stores and moves (with possible sign changes) or select data. Additionally, some noncomputational instructions can write directly to the FPSCR. All instructions executed in the load/store pipeline are noncomputational, while most executed in the arithmetic pipeline are computational.

All floating-point operands are stored internally in double-precision format. Arithmetic operations specified as single require that the internal data is representable as single (that is, having an unbiased exponent between -126 and 127 and a significand accurately representable in 24 bits). If the data cannot be represented in this way, the results stored in the FPR, and the status bits set in FPSCR and CR (as appropriate) are undefined.

For consistency, to reduce the likelihood of causing a serious malfunction due to user error, and to enable random testing, single-precision operations are performed on double-precision operands. For all cases except for **fdivs** and **fsqrts**, the operation is performed as if it were double-precision; the result is then rounded to single-precision. For **fdivs** and **fsqrts**, the appropriate number of iterations are performed to accomplish a single-precision result (potentially with early out); the quotient is then properly rounded.

In all cases, result exceptions (overflow, underflow, and inexact) are detected and reported based on the result, not on the source operands. Default (masked exception) results are the same as for the single-precision instructions. In the case of masked overflow or underflow exceptions, the least significant 11 bits of the adjusted true exponent are returned.

The results of all single-precision operations are rounded to single precision. These results are stored in double-precision format, but are restricted to single-precision range (exponent and fraction). All status bits are set based upon the single-precision result.

3.6.2 Load and Store Instructions

The product_name instruction set includes instructions to load from memory to an FPR and to store from an FPR to memory.

For load instructions, the function of the load/store logic is to receive data from the 16-byte bus from the A2O Core and present it to the FPRs. Data received from the A2O Core could be single- or double-precision and in the big- or little-endian formats. Also, the data received is word aligned. Data to the FPR must be in the big-endian, double-precision format.

For store instructions, one operand from the FPR is received. Data is to be word aligned on the output bus, There are two basic forms of load instruction: single-precision and double-precision. Because the FPRs support only floating-point double format, single-precision load floating-point instructions convert single-precision data to double format before loading the operand into the target FPR. The conversion and loading steps are as follows.

Let $WORD_{0:31}$ be the floating-point single-precision operand accessed from storage.

Normalized Operand

```

if  $WORD_{1:8} > 0$  and  $WORD_{1:8} < 255$  then
FPR(FRT) $_{0:1} \leftarrow WORD_{0:1}$ 
FPR(FRT) $_2 \leftarrow \neg WORD_1$ 
FPR(FRT) $_3 \leftarrow \neg WORD_1$ 
FPR(FRT) $_4 \leftarrow \neg WORD_1$ 
FPR(FRT) $_{5:63} \leftarrow WORD_{2:31} \parallel 29_0$ 

```

Denormalized Operand

```

if  $WORD_{1:8} = 0$  and  $WORD_{9:31} \neq 0$  then
sign  $\leftarrow WORD_0$ 
exp  $\leftarrow -126$ 
frac $_{0:52} \leftarrow 0b0 \parallel WORD_{9:31} \parallel 29_0$ 
normalize the operand
do while frac $_0 = 0$ 
    frac  $\leftarrow frac_{1:52} \parallel 0b0$ 
    exp  $\leftarrow exp - 1$ 
FPR(FRT) $_0 \leftarrow sign$ 
FPR(FRT) $_{1:11} \leftarrow exp + 1023$ 
FPR(FRT) $_{12:63} \leftarrow frac_{1:52}$ 

```

Zero / Infinity / NaN

if $\text{WORD}_{1:8} = 255$ or $\text{WORD}_{1:31} = 0$ then

$\text{FPR}(\text{FRT})_{0:1} \leftarrow \text{WORD}_{0:1}$

$\text{FPR}(\text{FRT})_2 \leftarrow \text{WORD}_1$

$\text{FPR}(\text{FRT})_3 \leftarrow \text{WORD}_1$

$\text{FPR}(\text{FRT})_4 \leftarrow \text{WORD}_1$

$\text{FPR}(\text{FRT})_{5:63} \leftarrow \text{WORD}_{2:31} \parallel {}^{29}0$

For double-precision load floating-point instructions, no conversion is required because the data from storage is copied directly into the FPR.

Some of the floating-point load instructions update GPR(RA), the effective address. For these forms, if $\text{RA} \neq 0$, the effective address is placed into GPR(RA) and the storage element (byte, halfword, word, or doubleword) addressed by EA is loaded into FPR(RT). If $\text{RA} = 0$, the instruction form is invalid.

Floating-point load storage accesses cause data storage exceptions if the program is not allowed to read the storage location. Floating-point load storage accesses cause data TLB error exceptions if the program attempts to access storage that is unavailable.

Note: RA and RB denote GPRs, while FRT denotes an FPR.

Both big-endian and little-endian byte orderings are supported.

Table 15. Floating-Point Load Instructions

Mnemonic	Operands	Instruction
lfd	FRT, D(RA)	Load Floating-Point Double
lfd u	FRT, D(RA)	Load Floating-Point Double with Update
lfd ux	FRT, RA, RB	Load Floating-Point Double with Update Indexed
lfd x	FRT, RA, RB	Load Floating-Point Double Indexed
lfd epx	FRT, RA, RB	Load Floating-Point Double External Process ID Indexed
lfs	FRT, D(RA)	Load Floating-Point Single
lfs u	FRT, D(RA)	Load Floating-Point Single with Update
lfs ux	FRT, RA, RB	Load Floating-Point Single with Update Indexed
lfs x	FRT, RA, RB	Load Floating-Point Single Indexed
lfiwax	FRT, RA, RB	Load Floating-Point as Integer Word Algebraic Indexed
lfiwzx	FRT, RA, RB	Load Floating-Point as Integer Word and Zero Indexed

Note: For complete instruction descriptions, see the *Power ISA V2.06* specification.

3.6.3 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 **stfiwx** instruction, described in the *Power ISA V2.06* specification. 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 format before storing the operand in storage. The conversion steps are as follows.

Let $\text{WORD}_{0:31}$ be the word in storage 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

$sign \leftarrow FPR(FRS)_0$

$exp \leftarrow FPR(FRS)_{1:11} - 1023$

$frac \leftarrow 0b1 \parallel FPR(FRS)_{12:63}$

denormalize operand

do while $exp < -126$

$frac \leftarrow 0b0 \parallel frac_{0:62}$

$exp \leftarrow exp + 1$

$WORD_0 \leftarrow sign$

$WORD_{1:8} \leftarrow 0x00$

$WORD_{9:31} \leftarrow frac_{1:23}$

else $WORD \leftarrow$ undefined

Notice 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 above (“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. The result of a single-precision load floating-point from WORD will not compare equal to the contents of the original source register.

For double-precision store floating-point instructions and for the store floating-point as integer word instruction, no conversion is required because the data from the FPR is copied directly into storage.

Some of the floating-point store instructions update GPR(RA) with the effective address. For these forms, if $RA \neq 0$, the effective address is placed into GPR(RA).

Floating-point store storage accesses cause a data storage interrupt if the program is not allowed to write to the storage location. Integer store storage accesses cause a data TLB error interrupt if the program attempts to access storage that is unavailable.

Note: RA and RB denote GPRs, while FRS denotes an FPR.

Both big-endian and little-endian byte orderings are supported.

Table 16. Floating-Point Store Instructions (Sheet 1 of 2)

Mnemonic	Operands	Instruction
stfd	FRS, D(RA)	Store Floating-Point Double
stfdu	FRS, D(RA)	Store Floating-Point Double with Update
stfdux	FRS, RA, RB	Store Floating-Point Double with Update Indexed
stfdx	FRS, RA, RB	Store Floating-Point Double Indexed
stfdexp	FRS, RA, RB	Store Floating-Point Double External Process ID Indexed
stfiwx	FRS, RA, RB	Store Floating-Point as Integer Word Indexed
stfs	FRS, D(RA)	Store Floating-Point Single

Table 16. Floating-Point Store Instructions (Sheet 2 of 2)

Mnemonic	Operands	Instruction
stfsu	FRS, D(RA)	Store Floating-Point Single with Update
stfsux	FRS, RA, RB	Store Floating-Point Single with Update Indexed
stfsx	FRS, RA, RB	Store Floating-Point Single Indexed

Note: For complete instruction descriptions, see the *Power ISA V2.06* specification.

3.6.4 Floating-Point Move Instructions

These instructions copy data from one floating-point register to another, altering the sign bit (bit 0) as described in the instruction descriptions in *Power ISA V2.06* specification for **fneg**, **fabs**, **fnabs**, and **fcpsgn**. These instructions treat NaNs just like any other kind of value (for example, the sign bit of a NaN can be altered by **fneg**, **fabs**, **fnabs**, and **fcpsgn**). These instructions do not alter the FSPCR.

Table 17. Floating-Point Move Instructions

Mnemonic	Operands	Instruction
fabs	FRT, FRB	Floating Absolute Value
fcpsgn	FRT, FRA, FRB	Floating Copy Sign
fmr	FRT, FRB	Floating Move Register
fnabs	FRT, FRB	Floating Negative Absolute Value
fneg	FRT, FRB	Floating Negate

Note: For complete instruction descriptions, see the *Power ISA V2.06* specification.

3.6.5 Floating-Point Arithmetic Instructions

These instructions perform elementary arithmetic operations.

Table 18. Floating-Point Elementary Arithmetic Instructions (Sheet 1 of 2)

Mnemonic	Operands	Instruction
fadd	FRT, FRA, FRB	Floating Add
fadds	FRT, FRA, FRB	Floating Add Single
fdiv	FRT, FRA, FRB	Floating Divide
fdivs	FRT, FRA, FRB	Floating Divide Single
fmul	FRT, FRA, FRB	Floating Multiply
fmuls	FRT, FRA, FRB	Floating Multiply Single
fre	FRT, FRB	Floating Reciprocal Estimate
fres	FRT, FRB	Floating Reciprocal Estimate Single
frsqrte	FRT, FRB	Floating Reciprocal Square Root Estimate
frsqrtes	FRT, FRB	Floating Reciprocal Square Root Estimate Single
fsqrt	FRT, FRB	Floating Square Root

Table 18. Floating-Point Elementary Arithmetic Instructions (Sheet 2 of 2)

Mnemonic	Operands	Instruction
fsqrts	FRT, FRB	Floating Square Root Single
fsub	FRT, FRA, FRB	Floating Subtract
fsubs	FRT, FRA, FRB	Floating Subtract Single

Note: For complete instruction descriptions, see the *Power ISA V2.06* specification.

3.6.5.1 Floating-Point Multiply-Add Instructions

These instructions combine a multiply and an add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide (L bit, FRACTION), and all 106 bits take part in the add/subtract portion of the instruction.

FPSCR 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, and 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, multiplication of infinity by 0 or of anything by an SNaN, and addition of an SNaN, cause the corresponding exception bits to be set.

Table 19. Floating-Point Multiply-Add Instructions

Mnemonic	Operands	Instruction
fmadd	FRT, FRA, FRB, FRC	Floating Multiply-Add
fmadds	FRT, FRA, FRB, FRC	Floating Multiply-Add Single
fmsub	FRT, FRA, FRB, FRC	Floating Multiply-Subtract
fmsubs	FRT, FRA, FRB, FRC	Floating Multiply-Subtract Single
fnmadd	FRT, FRA, FRB, FRC	Floating Negative Multiply-Add
fnmadds	FRT, FRA, FRB, FRC	Floating Negative Multiply-Add Single
fnmsub	FRT, FRA, FRB, FRC	Floating Negative Multiply-Subtract
fnmsubs	FRT, FRA, FRB, FRC	Floating Negative Multiply-Subtract Single

Note: For complete instruction descriptions, see the *Power ISA V2.06* specification.

3.6.6 Floating-Point Rounding and Conversion Instructions

Examples of uses of these instructions to perform various conversions can be found in *Appendix E.2 Floating-Point Conversions* on page 861.

Table 20. Floating-Point Rounding and Conversion Instructions

Mnemonic	Operand	Instruction
fcfid	FRT, FRB	Floating Convert From Integer Doubleword
fcfidu	FRT, FRB	Floating Convert From Integer Doubleword Unsigned
fcfids	FRT, FRB	Floating Convert From Integer Doubleword Single
fcfidus	FRT, FRB	Floating Convert From Integer Doubleword Unsigned Single
fctid	FRT, FRB	Floating Convert to Integer Doubleword
fctidu	FRT, FRB	Floating Convert to Integer Doubleword Unsigned
fctidz	FRT, FRB	Floating Convert to Integer Doubleword and Round to Zero
fctiduz	FRT, FRB	Floating Convert to Integer Doubleword Unsigned and Round to Zero
fctiw	FRT, FRB	Floating Convert to Integer Word
fctiwu	FRT, FRB	Floating Convert to Integer Word Unsigned
fctiwz	FRT, FRB	Floating Convert to Integer Word and Round to Zero
fctiwuz	FRT, FRB	Floating Convert to Integer Word Unsigned and Round to Zero
frim	FRT, FRB	Floating Round to Integer Minus
frin	FRT, FRB	Floating Round to Integer Nearest
frip	FRT, FRB	Floating Round to Integer Plus
friz	FRT, FRB	Floating Round to Integer Zero
frsp	FRT, FRB	Floating Round to Single-Precision

Note: For complete instruction descriptions, see the *Power ISA V2.06* specification.

3.6.7 Floating-Point Compare Instructions

The floating-point compare instructions compare the contents of two floating-point registers. Comparison ignores the sign of zero (+0 is treated as equal to -0). The comparison result can be ordered or unordered.

The comparison sets one bit in the designated CR field to 1 and the other three bits to 0. FPSCR[FPCC] is set in the same way.

The CR field and FPSCR[FPCC] are set as shown in *Table 3-21*.

Table 21. Comparison Sets

Bit	Name	Description
0	FL	(FRA) < (FRB)
1	FG	(FRA) > (FRB)
2	FE	(FRA) = (FRB)
3	FU	(FRA) ? (FRB) (unordered)

Table 22. Floating-Point Compare and Select Instructions

Mnemonic	Operands	Instruction
fcmpo	BF, FRA, FRB	Floating Compare Ordered
fcmpu	BF, FRA, FRB	Floating Compare Unordered
fsel	FRT, FRA, FRB, FRC	Floating Select
ftdiv	BF, FRA, FRB	Floating Test for Software Divide
ftsqrt	BF, FRA, FRB	Floating Test for Software Square Root

Note: For complete instruction descriptions, see the *Power ISA V2.06* specification.

3.6.8 Floating-Point Status and Control Register Instructions

Every Floating-Point Status and Control Register instruction synchronizes the effects of all floating-point instructions executed by a given processor. Executing a Floating-Point Status and Control Register instruction ensures that all floating-point instructions previously initiated by the given processor have completed before the Floating-Point Status and Control Register instruction is initiated, and that no subsequent floating-point instructions are initiated by the given processor until the Floating-Point Status and Control Register instruction has completed. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the Floating-Point Status and Control Register instruction is initiated.
- All invocations of the enabled exception type of program interrupt that will be caused by the previously initiated instructions have occurred before the Floating-Point Status and Control Register instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits is initiated until the Floating-Point Status and Control Register instruction has completed.

Floating-point load and floating-point store instructions are not affected.

Table 23. Floating-Point Status and Control Register Instructions

Mnemonic	Operands	Instruction
mcrfs		Move to Condition Register from FPSCR
mffs	FRT	Move from FPSCR
mtfsb0	BT	Move to FPSCR Bit 0
mtfsb1	BT	Move to FPSCR Bit 1
mtfsf	FLM, FRB	Move to FPSCR Fields
mtfsfi	BF,U	Move to FPSCR Field Immediate

Note: For complete instruction descriptions, see the *Power ISA V2.06* specification.



4. Initialization

TBD

4.1 Core Reset

TBD

4.2 A2O Core State After Reset

TBD

4.3 Core Reset Request and Status Signals

The Power ISA Book III-E defines two sets of core facilities that can be used to request a reset: debug and the watchdog timer. Whether or not these facilities are used is implementation-dependent. This section describes how the A2O core supports their usage through a set of generic reset request outputs and reset status inputs. The meaning that a system associates with these signals, and how they are used, is beyond the scope of this document.

The following sections describe how these facilities can be used by software to initiate a reset operation.

4.3.1 Core Reset Requests

The A2O core implements four reset request output signals. They originate from SPRs in the debug and timer facilities. Once activated, they remain set until cleared by software or through the reset operation. System software can assign different levels of reset actions (that is, core, chip, system) to these signals, and respond accordingly.

ac_an_reset_1_request	The core is requesting a type 1 reset.
ac_an_reset_2_request	The core is requesting a type 2 reset.
ac_an_reset_3_request	The core is requesting a type 3 reset.
ac_an_reset_wd_request	Is active with the reset request signal when it is coming from the watchdog timer; otherwise, the request is from DBCR0[RST].

Because the reset request logic is implemented in multiple thread-specific sources, it is possible that more than one reset request output signal can be active at the same time. Software should coordinate the use of these facilities within a partition so as to minimize multiple reset request types being activated together. On the other hand, the system reset control logic should be able to support multiple active requests, probably by initiating the highest requested reset type.

4.3.1.1 From Debug

Software can request a reset by writing a 2-bit encoded value to DBCR0[RST]. The A2O core decodes the bits and activates one of three reset type requests as shown below.

DBCR0[RST]:

- 0b00 - No reset.
- 0b01 - Activates a type 1 reset request.
- 0b10 - Activates a type 2 reset request.
- 0b11 - Activates a type 3 reset request.

4.3.1.2 From Watchdog Timer

Software enables watchdog timer resets through various fields in the Timer Control Register. The 2-bit TCR[WRC] field defines the type of reset that will be requested, and must be nonzero for a watchdog timer exception to activate a reset request. On the second watchdog timer exception, the TCR[WRC] value determines the corresponding reset type request as shown below:

TCR[WRC] when second watchdog timer exception is activated:

- 0b00 - Watchdog timer reset is disabled.
- 0b01 - Activates a type 1 reset request.
- 0b10 - Activates a type 2 reset request.
- 0b11 - Activates a type 3 reset request.

4.3.2 Reset Request Status

The system can obtain the cause of a reset request before taking any reset actions. The DBCR0[RST] and Fault Isolation Registers indicate which thread requested the reset. Additional state and status information for the failing thread can then be obtained by interrogating various registers within the core. Once the core is reset, however, this and any other status related to the problem is lost.

After reset operations have initialized the core's registers, status information can be written to the DBSR[MRR] and TSR[WRS] fields indicating that a core had request the reset. The A2O core implements four reset status inputs for this purpose:

an_ac_reset_1_complete	A type 1 reset occurred.
an_ac_reset_2_complete	A type 2 reset occurred.
an_ac_reset_3_complete	A type 3 reset occurred.
an_ac_reset_wd_complete	If this signal is active with another of the reset complete signals, it indicates that the reset was the result of a watchdog timer request. The TSR[WRS] field is updated in addition to the DBSR[MRR] field.

The an_ac_reset_x_complete inputs must be active for a minimum of one clock pulse to set the DBSR[MRR] and TSR[WRS] reset status bits. If more than one reset input is active at the same time, they are set using the following priority: highest = type 3, next = type 2, lowest = type 1.

4.3.2.1 Debug Facility Reset Status

The Most Recent Reset field of the Debug Status Register (DBSR[MRR]) indicates the type of reset that occurred last. The flush 0 scan of the core's reset initializes this field to 0b00. Once initialized by the core reset, external reset controls can use the an_an_reset_x_complete signals to provide additional reset status to software. The DBSR[MRR] field of all threads is updated with the same reset status.

DBSR[MRR]:

- 0b00 - No reset status since last cleared.
- 0b01 - A type 1 reset has occurred.
- 0b10 - A type 2 reset has occurred.
- 0b11 - A type 3 reset has occurred.

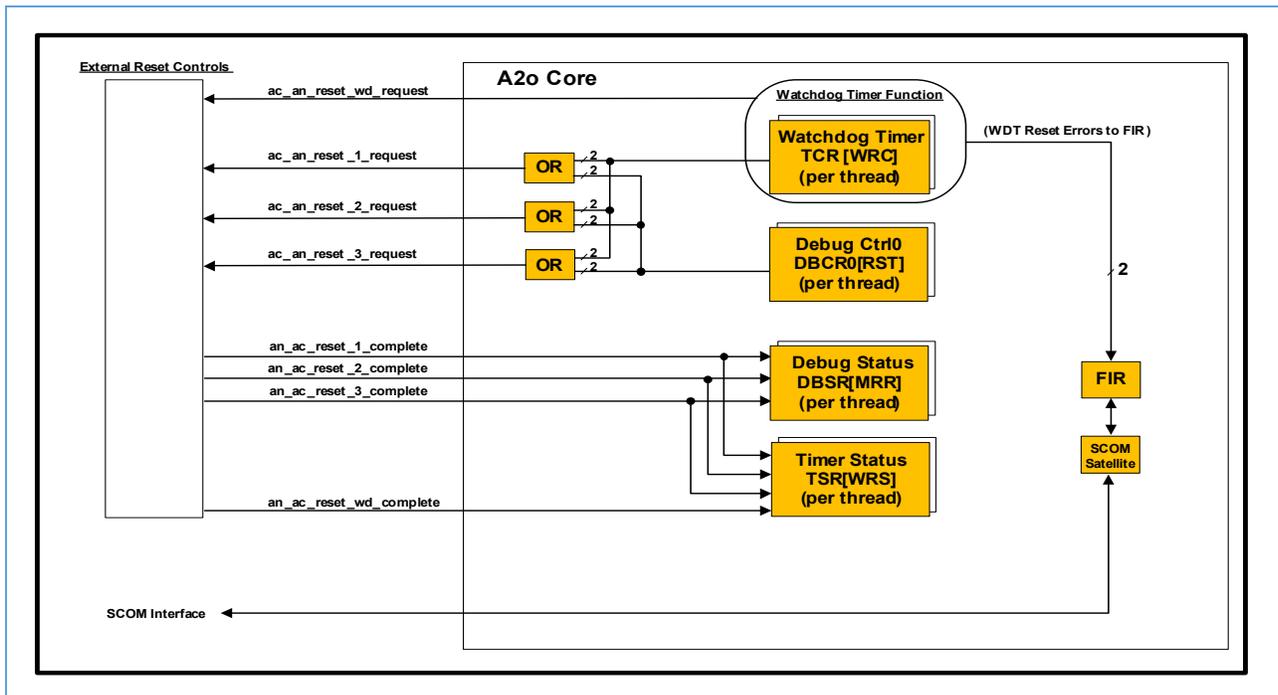
4.3.2.2 Timer Facility Reset Status

The Watchdog Timer Reset Status field of the Timer Status Register (TSR[WRS]) indicates that a reset was caused by a watchdog timer exception. The flush 0 scan of the core's reset initializes this field to 0b00. Once initialized by the core reset, external reset controls can use the an_an_reset_x_complete signals to provide additional reset status to software. The TSR[WRS] field of all threads is updated with the same reset status.

TSR[WRS]:

- 0b00 - No watchdog timer reset has occurred.
- 0b01 - The watchdog timer caused a type 1 reset.
- 0b10 - The watchdog timer caused a type 2 reset.
- 0b11 - The watchdog timer caused a type 3 reset.

Figure 1. Software-Initiated Reset Request Overview



4.4 Initialization Software Requirements

TBD

5. Instruction and Data Caches

The A20 Core provides separate instruction and data cache controllers and arrays, which allow concurrent access and minimize pipeline stalls. The storage capacity of the cache arrays is 32KB each. Both cache controllers have 64-byte lines. Both are set associative, with the data cache having 8-way set-associativity, and the instruction cache having 4-way set associativity. The Power ISA instruction set provides a rich set of cache management instructions for software-enforced coherency. The cache controllers interface to the processor interface for connection to the system-on-a-chip environment.

Both the data and instruction caches support a 42-bit real address, and both are parity protected against soft errors. The details of suggested interrupt handling are described in *Instruction Cache Controller* on page 154 and in *Data Cache Controller* on page 157

The rest of this chapter provides more detailed information about the operation of the instruction and data cache controllers and arrays.

5.1 Data Cache Array Organization and Operation

The data cache is 8-way set-associative, with 64 sets and a 64-byte line size.

Table 5-1 illustrates generically the ways and sets of the cache arrays, while *Table 5-2* provides specific values for the parameters used in *Table 5-1*. As shown in *Table 5-2*, the tag field for each line in each way holds the high-order address bits associated with the line that currently resides in that way. The middle-order address bits form an index to select a specific set of the cache, while the six lowest-order address bits form a byte-offset to choose a specific byte (or bytes, depending on the size of the operation) from the 64-byte cache line.

Table 1. Data Cache Array Organization

	Way 0	Way 1	...	Way $W - 2$	Way $W - 1$
Set 0	Line 0	Line n	...	Line $(W - 2)n$	Line $(W - 1)n$
Set 1	Line 1	Line $n + 1$...	Line $(W - 2)n + 1$	Line $(W - 1)n + 1$
•	•	•	•	•	•
•	•	•	•	•	•
•	•	•	•	•	•
Set $n - 2$	Line $n - 2$	Line $2n - 2$...	Line $(W - 1)n - 2$	Line $wn - 2$
Set $n - 1$	Line $n - 1$	Line $2n - 1$...	Line $(W - 1)n - 1$	Line $wn - 1$

Table 2. Cache Size and Parameters

Array Size	w (Ways)	n (Sets)	Tag Address Bits	Set Address Bits	Byte Offset Address Bits
32 KB	8	64	$A_{22:51}$	$A_{52:57}$	$A_{58:63}$

5.2 Instruction Cache Array Organization and Operation

The instruction is 4-way set-associative, with 128 sets and a 64-byte line size.

Table 5-3 illustrates generically the ways and sets of the cache arrays, while Table 5-4 provides specific values for the parameters used in Table 5-3. As shown in Table 5-4, the tag field for each line in each way holds the high-order address bits associated with the line that currently resides in that way. The middle-order address bits form an index to select a specific set of the cache, while the six lowest-order address bits form a byte-offset to choose the specific bytes from the 64-byte cache line.

Table 3. Instruction Cache Array Organization

	Way 0	Way 1	...	Way $w - 2$	Way $w - 1$
Set 0	Line 0	Line n	...	Line $(w - 2)n$	Line $(w - 1)n$
Set 1	Line 1	Line $n + 1$...	Line $(w - 2)n + 1$	Line $(w - 1)n + 1$
•	•	•	•	•	•
•	•	•	•	•	•
•	•	•	•	•	•
Set $n - 2$	Line $n - 2$	Line $2n - 2$...	Line $(w - 1)n - 2$	Line $wn - 2$
Set $n - 1$	Line $n - 1$	Line $2n - 1$...	Line $(w - 1)n - 1$	Line $wn - 1$

Table 4. Cache Size and Parameters

Array Size	w (Ways)	n (Sets)	Tag Address Bits	Set Address Bits	Byte Offset Address Bits
32 KB	4	128	$A_{22:50}$	$A_{51:57}$	$A_{58:63}$

5.3 Cache Line Replacement Policy

Memory addresses are specified as being cacheable or caching inhibited on a page basis, using the caching inhibited (I) storage attribute (see *Caching Inhibited (I)* on page 180). When a program references a cacheable memory location and that location is not already in the cache (a *cache miss*), the line can be brought into the cache (a *cache line fill* operation) and placed into any one of the ways within the set selected by the middle portion of the address (the specific address bits that select the set are specified in Table 5-2 and Table 5-4). If the particular way within the set already contains a valid line from some other address, the existing line is removed and replaced by the newly referenced line from memory. The line being replaced is referred to as the *victim*.



The way selected to be the victim for replacement is controlled by a pseudo LRU policy. The L1 data directory uses a pseudo LRU replacement algorithm, which allocates on data reload. The valid bits and the LRU are used to determine which way will be replaced. The way that is determined for replacement is then invalidated and updated.

5.4 Instruction Cache Controller

The instruction cache controller (ICC) delivers up to four instructions per cycle to the instruction unit of the A20 Core. The ICC uses a 128-bit interface. The ICC frequency is always 1:1 with the A20 Core.

The ICC provides a speculative prefetch mechanism that automatically prefetches up to one line per thread upon any fetch request that misses in the instruction cache.

The ICC also handles the execution of the Power ISA instruction cache management instructions, for invalidating cache lines or for flash invalidation of the entire cache. Resources for controlling and debugging the instruction cache operation are also provided.

The rest of this section describes each of these functions in more detail.

5.4.1 ICC Operations

When the ICC receives an instruction fetch request from the instruction unit of the A2O Core, the ICC simultaneously searches the instruction cache array for the cache line associated with the virtual address of the fetch request and translates the virtual address into a real address (see *Memory Management* on page 169 for information about address translation). If the requested cache line is found in the array (a cache hit), the four instructions at the requested address are returned to the instruction unit. If the requested cache line is *not* found in the array (a cache miss), the ICC sends a request for the entire cache line (64 bytes) to the A2 core interface, using the real address. If the caching inhibited (I) storage attribute is set for the memory page containing that cache line (see *Caching Inhibited (I)* on page 180) the ICC sends a request for 16 bytes.

As the ICC receives each portion of the cache line from the A2 core interface, it is written directly to the instruction cache. If the memory page containing the line is caching inhibited, the instructions are sent directly to the instruction unit.

After a request for a cache line read has been requested on the A2 core interface, the entire line read is performed and the line is written into the instruction cache (assuming no error occurs on the read), regardless of whether or not the instruction stream branches (or is interrupted) away from the line being read. This behavior is due to the nature of the A2 core interface and the fact that, once started, a cache line read request type cannot be abandoned. The ICC does not wait for this cache line read to complete before responding to a new request from the instruction unit on the same thread (due, perhaps, to a branch redirection or an interrupt). Instead, the ICC immediately accesses the cache to determine if the cache line at the new address requested by the instruction unit is already in the cache. If so, the requested four instructions from this line are immediately forwarded to the instruction unit, while the ICC in parallel continues to fill the previously requested cache line. In other words, the instruction cache is completely nonblocking.

Programming Note: It is a programming error for an instruction fetch request to reference a valid cache line in the instruction cache if the caching inhibited storage attribute is set for the memory page containing the cache line. The result of attempting to execute an instruction from such an access is undefined. After processor reset, hardware automatically sets the caching inhibited storage attribute for the memory page containing the reset address and also automatically flash invalidates the instruction cache. Subsequently, lines will not be placed into the instruction cache unless they are accessed by reference to a memory page for which the caching inhibited attribute has been turned off. If software subsequently turns on the caching inhibited storage attribute for such a page, software must make sure that no lines from that page remain valid in the instruction cache before attempting to fetch and execute instructions from the (now caching inhibited) page.

5.4.2 Instruction Cache Coherency

In general, the A2O Core does not automatically enforce coherency between  instruction cache,  data cache, and memory. Automatic enforcement of coherency is a function of the L2 cache or the SOC. See the chip specification to determine if instruction cache coherency is maintained by the hardware implementation. When not maintained by hardware, if the contents of the memory location are changed either within the data

cache or within memory itself, by the A2O Core through the execution of store instructions or by some other mechanism in the system writing to memory, software must use cache management instructions to ensure that the instruction cache is made coherent with these changes. This involves invalidating any obsolete copies of these memory locations within the instruction cache so that they will be reread from memory the next time they are referenced by program execution.

5.4.2.1 Self-Modifying Code

To illustrate the use of the cache management instructions to enforce instruction cache coherency, consider the example of self-modifying code, whereby the program executing on the A2O Core stores new data to memory with the intention of later branching to and executing this new “data,” which are actually instructions.

The following code example illustrates the required sequence for software to use when writing self-modifying code. This example assumes that *addr1* references a cacheable memory page.

```
stw regN, addr1# Store the data (an instruction) in regN to addr1 in the data cache.
dcbstaddr1# Write the new instruction from the data cache to memory.
msync # Wait until the data actually reaches the memory.
icbi addr1# Invalidate addr1 in the instruction cache if it exists.
msyncaddr1# Wait for the instruction cache invalidation to take effect.
isync # Flush any prefetched instructions within the ICC and instruction
      # unit and refetch them (an older copy of the instruction at addr1
      # might have already been fetched).
```

At this point, software can begin executing the instruction at *addr1* and be guaranteed that the new instruction will be recognized.

5.4.2.2 Instruction Cache Synonyms

A synonym is a cache line that is associated with the same real address as another cache line that is in the cache array at the same time. Such synonyms can occur when different virtual addresses are mapped to the same real address, and the virtual address is used either as an index to the cache array (a virtually-indexed cache) or as the cache line tag (a virtually-tagged cache).

The instruction cache on the A2O Core is real-indexed and real-tagged. It is not possible for synonyms to exist in the cache.

5.4.3 Instruction Cache Control and Debug

The A2O Core provides various registers and instructions to control instruction cache operation and to help debug instruction cache problems.

5.4.3.1 Instruction Cache Management and Debug Instruction Summary

In the instruction descriptions, the term “block” describes the unit of storage operated on by the cache block instructions. For the A2O Core, this is the same as a cache line.

The following instructions are used by software to manage the instruction cache.

icbi	Instruction Cache Block Invalidate Invalidates a cache block.
icbt	Instruction Cache Block Touch CT = 0 is treated as a no-op.
ici	Instruction Cache Invalidate Flash invalidates the entire instruction cache. Execution of this instruction is privileged.
icread	Not supported. Use scan operations to access the I-cache.

5.4.3.2 Instruction Cache Parity Operations

The instruction cache contains parity bits to protect against data errors. Both the instruction tags and data are protected. Instruction cache lines consist of a tag field,  bits of data, and parity on each group of 8 or less bits. The tag, data and parity bits are stored in normal RAM cells. The instruction cache is real-indexed and real-tagged.

Two types of errors can be detected by the instruction cache parity logic. In the first type, the parity bits stored in the RAM array are checked against the appropriate data in the instruction cache line when the RAM line is read for an instruction fetch.

A parity error that is detected within the instruction directory array causes only the way with parity problems to be invalidated. The instruction fetch that caused the lookup to occur does not take any other actions. This also causes bit 33 of Fault Isolation Register 0 (FIR0) to be set.

Any parity error that is detected on the data array causes the directory to be invalidated for the way that had the error, and also flushes if the way in error is for the current real address. This also causes bit 32 of FIR0 to be set.

5.4.3.3 Simulating Instruction Cache Parity Errors for Software Testing

Because parity errors occur in the cache infrequently and unpredictably, it is desirable to provide users with a way to simulate the effect of an instruction cache parity error so that interrupt handling software can be exercised. See *Section 15.2.3.2 Error Injection Register (ERRINJ)* on page 709.

5.5 Data Cache Controller

The data cache controller (DCC) handles the execution of the storage access instructions, moving data between memory and the data cache. The DCC interfaces to the A2 core interface using a shared command interface, a 128-bit data interface for read operations (shared with instruction fetches) and a 128-bit data interface for writes. The DCC frequency is always 1:1 with the A2O Core.

The DCC also handles the execution of the Power ISA data cache management instructions, for touching (perfecting), flushing, invalidating, or zeroing cache lines, or for flash invalidation of the entire cache. Resources for controlling and debugging the data cache operation are also provided.

The DCC interfaces to the auxiliary execution unit (AXU) port to provide direct load/store access to the data cache for AXU load and store operations, as well as for floating-point load and store instructions. AXU load and store instructions can access up to 16 bytes (one double quadword) in a single cycle.

Extensive load, store, and flush queues are also provided, such that up to eight outstanding load misses with the DCC continuing to service subsequent load and store hits in an in-order fashion.

The rest of this section describes each of these functions in more detail.

5.5.1 DCC Operations

When the DCC executes a load, store, or data cache management instruction, the DCC first translates the effective address specified by the instruction into a real address (see *Memory Management* on page 169 for more information about address translation). Next, the DCC accesses the data cache array for the cache line associated with the real address of the requested data. If the cache line is found in the array (a cache hit), that cache line is used to satisfy the request, according to the type of operation (load, store, and so on).

If the cache line is *not* found in the array (a cache miss), the next action depends upon the type of instruction being executed, as well as the storage attributes of the memory page containing the data being accessed. For most operations, and assuming the memory page is cacheable (see *Caching Inhibited (I)* on page 180), the DCC sends a request for the entire cache line (64 bytes) to the system interface. The request to the system interface is sent using the specific byte address requested by the instruction, so that the memory subsystem can read the cache line *target word first* (if it supports such operation) and supply the specific bytes requested before retrieving the rest of the cache line.

While the DCC is waiting for a cache line read to complete, it can continue to process subsequent instructions and handle those accesses that hit in the data cache. That is, the data cache is completely nonblocking.

As the DCC receives each portion of the cache line from the data read A2 core interface, data can be bypassed to the GPR file to satisfy load instructions, without waiting for the entire cache line to be filled. Data is written into the data cache immediately.

Once a data cache line read request has been made, the entire line read is performed and the line is written into the data cache. The DCC never aborts any A2 core interface request once it has been made, except when a processor reset occurs while the request is being made.

The DCC does not initiate speculative loads. Load requests to memory are always initiated in program order. Write requests to memory cannot be initiated speculatively.

If the guarded storage attribute is set for the memory page being accessed, then the memory request will *not* be initiated until it is guaranteed that the access is required by the **SEM**. Once initiated, the access will not be abandoned, and the instruction is guaranteed to complete *before* any change in the instruction stream. That is, if the instruction stream is interrupted, then upon return the instruction execution resumes *after* the instruction that accessed guarded storage, such that the guarded storage access will *not* be re-executed.

See *Guarded (G)* on page 180 for more information about accessing guarded storage.

Programming Note:

It is a programming error for a load, store, or **dcbz** instruction to reference a valid cache line in the data cache if the caching inhibited storage attribute is set for the memory page containing the cache line. The result of such an access is undefined. After processor reset, hardware automatically sets the caching inhibited storage attribute for the memory page containing the reset address. Software should flash invalidate the data cache (using **dci**; see *Data Cache Management Instruction Summary* on page 161) before executing any load, store, or **dcbz** instructions. Subsequently, lines are not placed into the data cache unless they are accessed by reference to a memory page for which the caching inhibited attribute has been turned off. If software subsequently turns on the caching inhibited storage attribute for such a page,

software must make sure that no lines from that page remain valid in the data cache (typically by using the **dcbf** instruction) before attempting to access the (now caching inhibited) page with load, store, or **dcbz** instructions.

The only instructions that are permitted to reference a caching inhibited line that is a hit in the data cache are the cache management instructions **dcbst**, **dcbf**, **dcbi**, **dci**. The **dcbt**, **dcbtst**, **dcbtls**, **dcbtstls** and **dcblic** instructions have no effect if they reference a caching inhibited address, regardless of whether the line exists in the data cache.

5.5.1.1 Load and Store Alignment

The DCC implements all of the integer load and store instructions defined for 64-bit implementations by the Power ISA. These include byte, halfword, and word loads and stores, and doubleword loads and stores, as well as load and store string (0 to 127 bytes) and load and store multiple (1 to 32 registers) instructions. Integer byte, halfword, word, and doubleword loads and stores are performed with a single access to memory if the entire data operand is contained within an aligned 32-byte (double quadword) block of memory, regardless of the actual operand alignment within that block. If the data operand crosses a double quadword boundary, the load or store is performed using two or more accesses to memory.

The load and store string and multiple instructions are performed using one memory access for each byte, until the end of the load or store string or multiple is reached.

The DCC handles all misaligned integer load and store accesses in hardware, without causing an alignment exception. However, the control bit XUCR0[FLSTA] can be set to force all misaligned storage access instructions to cause an alignment exception. When this bit is set, all integer storage accesses must be aligned on an operand-size boundary, or an alignment exception results. Load and store multiple instructions must be aligned on a 4-byte boundary, while load and store string instructions can be aligned on any boundary (these instructions are considered to reference *byte* strings, and hence the operand size is a byte).

The DCC also supports load and store operations over the AXU interface. These can include floating-point load and store instructions (as defined by the Power ISA), as well as AXU load and store instructions for auxiliary processors. While floating-point loads and stores can access either 4 or 8 bytes, AXU loads and store can access up to a 16 bytes.

The DCC handles all misaligned floating-point and AXU loads and stores with a single memory access, as long as they do not cross a double quadword boundary. If such an access crosses a quadword boundary, the DCC signals an alignment exception, and an interrupt results.

The AXU interface also supports other options with regards to the handling of misaligned AXU and floating-point loads and stores. The AXU interface can specify that the DCC handle any AXU or floating-point load or store access that is not aligned on either an operand-size boundary or a word boundary as specified in *Table 2-2 Alignment Effects for Storage Access Instructions* on page 61. Alternatively, the AXU interface can specify that the DCC should force the storage access to be aligned on an operand-size boundary by zeroing the appropriate number of low-order address bits.

Floating-point and AXU loads and stores are also subject to the function of XUCR0[AFLSTA].

5.5.1.2 Load Operations

Load instructions that reference cacheable memory pages and miss in the data cache result in cache line read requests being presented to the system interface. Load operations to caching inhibited memory pages, however, only access the bytes specifically requested, according to the type of load instruction. This behavior

(of only accessing the requested bytes) is only architecturally required when the guarded storage attribute is also set, but the DCC enforces this requirement on any load to a caching inhibited memory page. Subsequent load operations to the same caching inhibited locations cause new requests to be sent to the data read A2 core interface.

The DCC also includes a 8-entry load miss queue (LMQ), which holds up to eight outstanding load instructions that have either missed in the data cache or access caching inhibited memory pages. A load instruction in the LMQ remains there until the requested data arrives, at which time the data is delivered to the register file, the data cache is updated (if cacheable), and the instruction is removed from the LMQ.

5.5.1.3 Store Operations

The processing of store instructions in the DCC is not affected by the following storage attribute bits: write-through (W) and guarded (G). All store instructions are treated in a similar manner; the data is written directly to memory. The DCC never gathers memory write operations caused by separate store instructions into one simultaneous access to memory. The DCC never allocates data in the cache for store instructions.

The processing of store instructions in the DCC is affected by the caching inhibited (I) storage attribute. If the store is cacheable and hits in the data cache, the data cache is updated.

5.5.1.4 Data Read and Instruction Fetch Interface Requests

When an A2 core interface read request results from an access to a cacheable memory location, the request is always for a 64-byte line read, regardless of the type and size of the access that prompted the request. The address presented is for the first byte of the target of the access.

On the other hand, when an A2 core interface read request results from an access to a caching-inhibited memory location, only the bytes specifically accessed are requested from the interface, according to the type of instruction prompting the access. Based on the type of storage access instructions (including integer, floating-point, and AXU), and based on the mechanism for handling misaligned accesses that cross a quadword boundary (see *Load and Store Alignment* on page 159), the following types of read requests can occur due to caching inhibited requests:

- 1-byte read (any byte address 0–31 within a double quadword)
- 2-byte read (any byte address 0–30 within a double quadword)
- 4-byte read (any byte address 0–28 within a double quadword)
- 8-byte read (any byte address 0–24 within a double quadword)
- 16-byte read (any byte address 0–16 within a double quadword)

This request can only occur due to a quadword AXU load instruction or an I = 1 instruction fetch.

5.5.1.5 Data Write Interface Requests

When an A2 core interface write request results from store operations, the type and size of the request can be any one of the following

- 1-byte write request (any byte address 0–31 within a double quadword)
- 2-byte write request (any byte address 0–30 within a double quadword)
- 4-byte write request (any byte address 0–28 within a double quadword)
- 8-byte write request (any byte address 0–24 within a double quadword)

- 16-byte write request (any byte address 0–16 within a double quadword)
Only possible due to an AXU quadword store.

5.5.1.6 Storage Access Ordering

The DCC may perform load and store operations out of order with respect to the instruction stream. The DCC does enforce the requirements of the SEM, such that the net result of a sequence of load and store operations is the same as that implied by the order of the instructions. This means, for example, that if a later load reads the same address written by an earlier store, the DCC guarantees that the load will use the data written by the store, and not the older “pre-store” data.

The A2O Core provides storage synchronization instructions to enable software to control the order in which the memory accesses associated with a sequence of instructions are performed. See *Storage Ordering and Synchronization* on page 120 for more information about the use of these instructions.

5.5.2 Data Cache Coherency

The A2O Core does not enforce the coherency of the data cache with respect to alterations of memory performed by entities other than the A2O Core. Similarly, if entities other than the A2O Core attempt to read memory locations that currently exist within the A2O Core data cache, the A2O Core does not recognize such accesses and thus will not respond to such accesses. In other words, the data cache on the A2O Core is not a snooping data cache, and there is no hardware enforcement of data cache coherency with memory with respect to other entities in the system that access memory.

It is either the responsibility of software to manage this coherency through the appropriate use of the caching inhibited storage attribute and/or the data cache management instructions, or it is the responsibility of an entity outside the A2 core (such as an L2 cache controller) to provide cache coherency. The A2 core interface provides an input bus for such an entity to invalidate cache lines in the L1 data cache.

5.5.3 Data Cache Control

The A2O Core provides various registers and instructions to control data cache operation and to help debug data cache problems.

5.5.3.1 Data Cache Management Instruction Summary

In the instruction descriptions, the term “block” describes the unit of storage operated on by the cache block instructions. For the A2O Core, this is the same as a cache line.

Section 2.12.1 Privileged Instructions on page 117 summarizes which data cache management instructions are privileged.

The following instructions are used by software to manage the data cache.

<p>dcba</p>	<p>Data Cache Block Allocate This instruction is implemented as a no-op on the A2O Core.</p>
--------------------	--

dcbf dcbfep	Data Cache Block Flush Invalidate the cache block. CT indicates the targeted cache. Always send dcbf to the A2 core interface except when L = 3.
dcbi	Data Cache Block Invalidate Invalidates a cache block and then send dcbi to the A2 core interface.
dcbst dcbstep	Data Cache Block Store Send the dcbst to the A2 core interface.
dcbt dcbtep	Data Cache Block Touch Initiates a cache block fill, enabling the fill to begin before the executing program requiring any data in the block. The program can subsequently access the data in the block without incurring a cache miss. Send the dcbt to the A2 core interface. CT indicates the targeted cache. The instruction is a no-op for CT values other than 0 or 2.
dcbtst dcbtstep	Data Cache Block Touch for Store Send the dcbtst to the A2 core interface. CT indicates the targeted cache. The instruction is a no-op for CT values other than 0 or 2.
dcbz dcbzep	Data Cache Block Set to Zero Invalidate the block and send the dcbz to the A2 core interface.
dci	Data Cache Congruence Class Invalidate Flash invalidates the entire data cache.
dcread	Not supported. Use scan operations to access the data cache.
dcbtls	Data Cache Block Touch and Lock Set This instruction is similar in nature to the dcbt , but in addition locks the cache line in the data cache
dcbtstls	Data Cache Block Touch for Store and Lock Set This instruction is similar in nature to the dcbtst , but in addition locks the cache line in the data cache
dcblic	Data Cache Block Lock Clear

5.5.3.2 *dcbt* and *dcbtst* Operation

The **dcbt** instruction is typically used as a “hint” to the processor that a particular block of data is likely to be referenced by the executing program in the near future. Thus, the processor can begin filling that block into the data cache, so that when the executing program eventually performs a load from the block it will already be present in the cache, thereby improving performance.

The **dcbtst** instruction is typically used for a similar purpose, but specifically for cases where the executing program is likely to store to the referenced block in the near future. The differentiation in the purpose of the **dcbtst** instruction relative to the **dcbt** instruction is only relevant within shared-memory systems with hardware-enforced support for cache coherence. In such systems, the **dcbtst** instruction attempts to establish the block within the data cache in such a fast manner that the processor can most readily subsequently write to the block (for example, in a processor with a MESI-protocol cache subsystem, the block might be obtained in the exclusive state).

The **dcbt** instruction can also be used as a convenient mechanism for setting up a fixed, known environment within the data cache. This is useful for deterministic performance on a particular sequence of code or even for debugging of low-level hardware and software problems.

When being used for these latter purposes, it is important that the **dcbt** instruction deliver a deterministic result, namely the guaranteed establishment in the cache of the specified line. Accordingly, the execution of **dcbt(TH = 0)** is guaranteed to establish the specified cache line in the data cache (assuming that a TLB entry

for the referenced memory page exists and has read permission, and that the caching inhibited storage attribute is not set). The cache line fill associated with such a guaranteed **dcbt** occurs regardless of any potential instruction execution-stalling circumstances within the DCC.

5.5.3.3 Cache Locking Mechanisms

A2 supports the embedded cache locking instruction category. In addition the data cache supports way locking for transient data.

L1 Data Cache Way Locking

Setting XUCR0[WLK] = 1 enables data cache way locking. See *Section 14.5.142 XUCR0 - Execution Unit Configuration Register 0* on page 693. The data cache way locking mechanism alters the data caches normal LRU replacement policy. This LRU mode is used to indicate that one or more of the eight ways of the 8-way set associative data cache is eligible to be replaced. Memory pages are mapped to one of four ClassIDs by the TLB[WLC] bits in the ERAT/TLB.

- TLB[WLC] = 00 indicates ClassID 0 / RMT entry 0 when XUCR0[WLK] = 1.
- TLB[WLC] = 11 indicates ClassID 1 / RMT entry 1 when XUCR0[WLK] = 1.
- TLB[WLC] = 10 indicates ClassID 2 / RMT entry 2 when XUCR0[WLK] = 1.
- TLB[WLC] = 01 indicates ClassID 3 / RMT entry 3 when XUCR0[WLK] = 1.

A typical usage might be to use ClassID 1 to identify transient data and limit transient data to one way in the data cache. ClassID 0 is then used for all other data and limited to ways 0 - 6 in the data cache.

The ClassID is used to select a replacement management table (RMT) entry. A RMT entry indicates which sets are eligible for replacement for a given data cache miss. Each RMT entry is 8 bits, with 1 bit corresponding to each way in the data cache. The value of each bit indicates the following:

- 0 = Way is not eligible for replacement.
- 1 = Way is eligible for replacement.

If the RMT entry = all zeros, then all ways are locked, which results in an overlocking situation. The new line is not placed in the cache, and the data cache overlock bit XUCR0[CLO] is set. This does not cause an exception condition.

Embedded Cache Locking

User-mode instructions perform cache line locking and unlocking based on the cache line address. **dcblc**, **dcbtls**, and **dcbstls** are for data cache locking and unlocking, and **icblc** and **icbtls** are for instruction cache locking. Instruction cache locking is not supported in the instruction cache.

The CT operand is used to indicate the cache target of the cache line locking instruction.

For locking instructions:

- CT = 0 indicates L1 only. Note that **icbtls** CT = 0 is treated the same as **icbt** CT = 0 because instruction cache locking/unlocking is not supported.
- CT = 2 indicates L2 only. The cache line is not placed in the L1 if it does not exist and is not locked in the L1.

For unlocking instructions:

- CT = 0 indicates L1 only. Note that **icblc** CT = 0 is not sent to the L2.
- CT = 2 indicates L2 only.

Lock instructions are treated as loads when translated by the data TLB, and they cause exceptions when data TLB errors or data storage interrupts occur.

The user-mode cache lock enable bit, MSR[UCLE], is used to restrict user-mode cache line locking by the operating system. If MSR[UCLE] = 0, any cache lock instruction executed in user mode (MSR[PR] = 1) causes a cache-locking **DSI** exception and sets ESR[DLK]. This allows the operating system to manage and track the locking and unlocking of cache lines by user-mode tasks. If MSR[UCLE] is set to 1, the cache-locking instructions can be executed in user mode and do not cause a DSI for cache locking. However, they can still cause a DSI for access violations.

XUCR0[WLK] = 0: If all of the ways are locked in a cache set due to line locking, an attempt to lock another line in that set results in an overlocking situation. The new line is not placed in the cache, and the data cache overlock bit XUCR0[CLO] is set. This does not cause an exception condition.

XUCR0[WLK] = 1: If all of the ways are locked in a cache set with the combination of line locking for that cache set and way locking via the RMT table; an attempt to bring in a new line, via a data cache load miss, to that cache set results in an overlocking situation. The new line is not placed in the cache, and the data cache overlock bit XUCR0[CLO] is set. This does not cause an exception condition.

The following cases cause an attempted lock or unlock to fail:

- The target address is marked caching-inhibited.
- The L1 D-cache is disabled, and the CT operand of the data cache locking instruction = 0.
- The CT operand of the cache locking instruction is not equal to 0 or 2.

In these cases, the lock set instruction is treated as a no-op and the data cache unable-to-lock bit XUCR0[CUL]) is set. This condition does not cause an exception.

It is acceptable to lock all ways of the data cache. A nonlocking line fill for a new address in a completely locked data cache set is not put into the data cache.

Locking all ways in the L2 cache that might be shared by multiple A2 cores causes capacity evictions of potentially locked lines. See the L2 *User's Manual* for a detailed description.

The cache-locking DSI handler must decide whether to lock a given cache line based on available cache resources.

If the locking instruction is a set lock instruction, to lock the line, the handler should do the following:

1. Add the line address to its list of locked lines.
2. Execute the appropriate set lock instruction to lock the cache line.
3. Modify Save/Restore Register 0 (SRR0) to point to the instruction immediately after the locking instruction that caused the DSI.
4. Execute an **rfi**.

If the locking instruction is a clear lock instruction, to unlock the line, the handler should do the following:

1. Remove the line address from its list of locked lines.
2. Execute the appropriate clear lock instruction to unlock the cache line.

3. Modify SRR0 to point to the instruction immediately after the locking instruction that caused the DSI.
4. Execute an **rfi**.

Failure to update SRR0 to point to the instruction after the locking/unlocking instruction causes the exception handler to be repeatedly invoked for the same instruction.

Effects of Other Cache Instructions on Locked Lines

The following cache instructions do not affect the state of a cache line's lock bit:

- **dcbt** (CT = 0)
- **dcbtst** (CT = 0)

If **dcbt** is performed to a line that is locked in the cache, **dcbt** takes no action. However, if the line is invalid and therefore not locked, **dcbt** executes normally.

If a **dcbtst** (CT = 0) is performed to a line that is locked in the cache, **dcbtst** takes no action. If the line is invalid and therefore not locked, **dcbtst** executes normally.

The following instructions invalidate and unlock a line in the data cache of the current processor. These instructions are sent to the A2 system interface and can flush/invalidate and unlock caches on this processor and caches in other processors in a multiprocessor system (See the system user's manual.)

- **dcbf**
- **dcbi**
- **dcbz**
- **lbarx, lharx, lwarx, ldarx**
- **stbwcx., sthwcx., stwcx., stdcx.**

Flash Clearing of Lock Bits:

The A2 core allows flash clear of the data cache lock bits under software control. The cache's lock bits can be flash cleared through the CLFC control bit in XUCR.

Lock bits in both caches are cleared automatically upon power-up. A subsequent soft reset operation does not clear the lock bits automatically. Software must use the CLFC controls if flash clearing of the lock bits is desired after a soft reset. Setting the CLFC bit causes a flash clearing to be performed in a single CPU cycle, after which the CLFC bit is automatically cleared (CLFC bits are not sticky).

Instructions:

Name	Mnemonic	Syntax	Implementation Details
Data Cache Block Lock Clear	dcblc	CT,rA,rB	If CT = 0 and the line is in the L1 data cache, the data cache lock bit for that line is cleared, making it eligible for replacement. if CT = 2 and the line is in the L2 cache, the lock bit for that line is cleared, making it eligible for replacement. The line is unlocked in the L1 data cache if it exists.
Data Cache Block Touch and Lock Set	dcblts	CT,rA,rB	If CT = 0, the line is loaded and locked into the L1 data cache and the L2 cache. If CT = 0 and the block is already in the data cache, the line is locked in the data cache and the L2 cache. If CT = 2, the line is loaded and locked in the unified L2 cache. If CT = 2 and the block is already in the L2 cache, the line is locked in the L2 cache.

Name	Mnemonic	Syntax	Implementation Details
Data Cache Block Touch for Store and Lock Set	dcbtstls	CT,rA,rB	If CT = 0, the line is loaded and locked into the L1 data cache and the L2 cache. If CT = 0 and the block is already in the data cache, the line is locked in the data cache and the L2 cache. If CT = 2, the line is loaded and locked in the unified L2 cache. If CT = 2 and the block is already in the L2 cache, the line is locked in the L2 cache.
Instruction Cache Block Lock Clear	icbhc	CT,rA,rB	If CT = 0, it is treated the same as CT = 2 because instruction cache locking/unlocking is not supported. If CT = 2 and the line is in the L2 cache, the lock bit for that line is cleared.
Instruction Cache Block Touch and Lock Set	icbtl	CT,rA,rB	If CT = 0, it is treated the same as CT = 2 because instruction cache locking/unlocking is not supported. If CT = 2, the line is loaded into the unified L2 cache and the line is locked into the L2 cache. If CT = 2 and the block already exists in the L2 cache, the line is locked into the L2 cache.

Notes:

- In the L1 data cache, the A2 implements a lock bit for every index and way, allowing a line locking granularity. Setting CT = 0 specifies the L1 cache.
- The A2 supports CT = 0 and CT = 2.
- If the CT value is not supported, the instruction is treated as a no-op.
- Setting XUCR0[CLFC] flash clears all data cache lock bits, allowing system software to clear all cache locking in the L1 cache without knowing the addresses of the lines locked.
- Overlocking occurs when **dcbtl**, **dcbtstls**, or **icbtl** is performed to an index in either the L1 or L2 cache that already has all ways locked. In the A2, overlocking does not generate an exception. Instead, if a touch and lock set is performed with CT = 0 to an index in which all cache ways are already locked, XUCR0[CLO] is set indicating an overlock. The new line is not locked or cached.

To precisely detect an overlock condition in the data cache, system software must perform the following code sequence:

```

dcbtl
msync
mfspr (XUCR0)
  (check XUCR0[CUL] for data cache index unable-to-lock condition)
  (check XUCR0[CLO] for data cache index overlock condition)

```

Cache locking in the instructions cache (CT = 0) is not supported.

Touch and lock set instructions (**icbtl**, **dcbtl** and **dcbtstls**) are always executed and are not treated as hints. When one of these instructions is performed to an index and the way cannot be locked, XUCR0[CUL] is set to indicate an unable-to-lock condition. This occurs if the instruction must be no-op'ed.

The A2 implements a flash clear for all data cache lock bits (using XUCR0[CLFC]). This allows system software to clear all data cache locking bits without knowing the addresses of the lines locked.

Table 5. XUCR Bits

XUCR Bits	Description
CSLC	Cache snoop lock clear. A sticky bit is set by hardware if a dcbi snoop (either internally or externally generated) invalidated a locked cache block. Note that the lock bit 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 dcbi snoop that invalidated a locked line. 1 The cache has encountered a dcbi snoop that invalidated a locked line.
CUL	Cache unable to lock. A sticky bit is set by hardware and cleared by writing 0 to this bit location. 0 Indicates a lock set instruction was effective in the cache. 1 Indicates a lock set instruction was not effective in the cache.
CLO	Cache lock overflow. A sticky bit is set by hardware and cleared by writing 0 to this bit location. 0 Indicates a lock overflow condition was not encountered in the cache. 1 Indicates a lock overflow condition was encountered in the cache.
CLFC	Cache lock bits flash clear. Writing a 1 during a flash clear operation causes an undefined operation. Writing a 0 during a flash clear operation is ignored. Clearing occurs regardless of the data cache enable XUCR0[DC_DIS] value. 0 Default. 1 Hardware initiates a cache lock bits flash clear operation. CLFC clears to 0 when the operation completes.

5.5.3.4 Data Cache Parity Operations

The data cache contains parity bits to protect against soft data errors. Both the data cache tags and data are protected. Data cache lines consist of a tag field, 512 bits of data, and parity on each 8 or less bits. The tag field, data, and parity bits are stored in normal RAM cells. The data cache is physically tagged and indexed, so that the tag field contains a real address that is compared to the real address produced by the translation hardware when a load, store, or other cache operation is executed.

Parity bits stored in the RAM array are checked against the appropriate data in the RAM line any time the RAM line is read. The RAM data can be read by a directory lookup that matches the tag address, such as a load, **dcbf**, **dcbi**, or **dcbst**.

Parity error recovery is handled by flushing newer instructions for that thread, invalidating the cache line, and converting the load hit to a load miss by inserting it into the load miss queue and removing any load misses from the queue that are being flushed due to the parity error.

5.5.3.5 Simulating Data Cache Parity Errors for Software Testing

Because parity errors occur in the cache infrequently and unpredictably, it is desirable to provide users with a way to simulate the effect of a data cache parity error so that interrupt handling software can be exercised.

See *Section 15.2.3.2 Error Injection Register (ERRINJ)* on page 709. The error injection is performed by raising the error indication signal, not by multiplexing actual bad data into the dataflow path.

5.5.3.6 Data Cache Disable

The data cache can be disabled by setting XUCR0[DC_DIS]. When the data cache is disabled; no accesses to the cache occur, translation still occurs, exceptions scenarios cause an interrupt to occur, and the current state of the cache remains. All load/store operations miss the cache, all cache line management instructions are treated as misses and do not update the contents of the directory, and back-invalidates from the L2 do not invalidate any cache lines. A **dci** instruction does however invalidate the entire data cache directory

contents including valid, line locked indicator, and watchbit for all threads. A **wclr** instruction with $L[0] = 0$ does not invalidate the issuing threads directory watch contents, but does update the STM_WATCHLOST indicator. The **wchkall** instruction ignores the state of XUCR0[DC_DIS]; it executes as normal and updates the CR register with the contents of the STM_WATCHLOST indicator.

When turning on the data cache after it has been disabled, the cache is in a noncoherent state. It is the responsibility of software to execute the following instruction sequence before setting XUCR0[DC_DIS] to 0:

```
sync  
dci  
sync  
wclr 0,r0,r0  
isync  
mtspr xucr0, CACHE_EN
```

This guarantees that the contents of the data cache have been cleared and will reset the STM_WATCHLOST indicator.

6. Memory Management

The A20 core supports a uniform, 2^{64} bytes (64 bits) effective address (EA) space and a 4 TB (42-bit) real address (RA) space. The A20 memory management unit (MMU) performs address translation between virtual and real addresses, as well as protection functions. With appropriate system software, the MMU supports:

- Translation of an 88-bit virtual address (1-bit guest space identifier, 8-bit logical partition identifier, 1-bit address space (AS) identifier, 14-bit process ID (PID), and 64-bit effective address) into the 42-bit real address (note that the indirect entry identifier [IND] is not considered part of the virtual address).
- Software control of the page replacement strategy.
- Page-level access control for instruction and data accesses.
- Page-level storage attribute control.

6.1 MMU Overview

The A2 address translation facility operates in one of two modes: MMU mode or “ERAT-only” mode (controlled by CCR2[NOTLB]). The MMU mode assumes an underlying hardware MMU containing a translation lookaside buffer (TLB). The “ERAT-only” mode assumes no underlying TLB. For a detailed description of the “ERAT-only” mode of operation, see *Section 6.14* on page 216. Unless otherwise noted, what follows is a description of this implementation’s MMU mode. Also, unless otherwise noted, this description is assumed to be the 64-bit mode operational description.

The A20 core generates effective addresses for instruction fetches and data accesses. An effective address is a 64-bit address formed by adding an index or displacement to a base address (see *Effective Address Calculation* on page 62). Instruction effective addresses are for sequential instruction fetches and for fetches caused by changes in program flow (branches and interrupts). Data effective addresses are for load, store, and cache management instructions. The MMU expands effective addresses into virtual addresses (VAs) and then translates them into real addresses (RAs); the instruction and data caches use real addresses to access memory.

The A20 MMU supports demand-paged virtual memory and other management schemes that depend on precise control of effective to real address mapping and flexible memory protection. Translation misses and protection faults cause precise interrupts. The hardware provides sufficient information to correct the fault and restart the faulting instruction.

The A2 MMU supports hardware page table walking. The MMU uses software installed indirect translation entries (tagged with IND = 1 designations) in the TLB to assist the hardware in finding hardware page table entries (PTEs). These page table entries are fetched and used to form and install direct (IND = 0) TLB entries that are subsequently used for virtual to real address translation. See *Section 6.16 Hardware Page Table Walking (Category E.PT)* for a description of this function.

The MMU divides storage into pages. The page representation is the granularity of address translation, access control, and storage attribute control. The Power ISA MAV 2.0 architecture defines page sizes, of which the A20 MMU supports five (for direct IND = 0 entries). These five page sizes (4 KB, 64 KB, 1 MB, 16 MB, and 1 GB) are simultaneously supported. The A2 MMU also supports indirect (IND = 1) page sizes of 1 MB and 256 MB. See *Section 6.16.2 Indirect TLB Entry Page and Sub-Page Sizes* for a description of indirect page size usage. A valid entry for a page referenced by an effective address must be in the TLB for the

address to be accessed. An attempt to access an address for which no direct or indirect TLB entry exists causes an instruction or data TLB error interrupt, depending on the type of access (instruction or data). See *CPU Interrupts and Exceptions* on page 277 for more information about these and other interrupt types.

The TLB is parity protected against soft errors. The details of parity checking are described in the following sections.

6.1.1 Support for Power ISA MMU Architecture



The A2 memory management unit is based on *Power ISA* Book III-E Embedded MMU Architecture Version 2.0 (MAV 2.0). Unless otherwise noted, the A2 MMU conforms to this architecture and the following additional categories: Embedded.Hypervisor (E.HV), Embedded.Hypervisor.LRAT (E.HV.LRAT), Embedded.TLB Write Conditional (E.TWC), and Embedded.Page Table (E.PT).

Extensions

The Power ISA Architecture defines specific requirements for MMU implementations, but also leaves the details of several features implementation-dependent. The A2 core is fully compliant with the required MMU mechanisms defined by the Power ISA, but a few optional mechanisms are not supported. These are:

- Page Sizes

The Power ISA for MAV 2.0 defines 32 different page sizes, but does not require that an implementation support all of them. Accordingly, the A2O core supports five of these page sizes, from 4 KB up to 1 GB (nonconsecutive), as mentioned in *MMU Overview* on page 169 and as listed in *Table 6-1 Page Size and Effective Address to EPN Comparison* on page 175. The Power ISA page sizes are defined as power of 2×1 KB sizes and represented by a 5-bit value. The page sizes supported by A2 all happen to be power of 4×1 KB sizes. For this reason, the LSB of the architected page size encoding is assumed to be zero always and is not implemented in the A2 MMU.

- Address Space

The A2 effective page number (EPN) field varies from 34 to 52 bits, depending on page size. The real page number (RPN) field varies from 12 to 30 bits, depending on page size. The total 42 bits of the real address is the combination of the RPN with the page offset portion of the effective address. See *Address Translation* on page 175 for a more detailed explanation of these fields and the formation of the real address.

6.2 Page Identification

The TLB is the hardware resource that controls page identification and address translation; it contains page protection and storage attributes. The Valid (V), Effective Page Number (EPN), Translation Guest Space identifier (TGS), Translation Logical Partition identifier (TLPID), Translation Space identifier (TS), Translation ID (TID), and Page Size (SIZE) fields of a particular TLB entry identify the page associated with that TLB entry. In addition, the indirect (IND) bit of a TLB entry identifies it as a direct virtual to real translation entry (IND = 0) or an indirect (IND = 1) hardware page table pointer entry that requires additional processing. Except as noted, all comparisons using these fields must succeed to validate this entry for subsequent translation and access control processing. Failure to locate a matching direct or indirect TLB entry based on this criteria for instruction fetches causes a TLB miss exception that results in an instruction TLB error interrupt. Failure to locate a matching direct or indirect TLB entry based on this criteria for data storage accesses

causes a TLB miss exception that might result in a data TLB error interrupt, depending on the type of data storage access (certain cache management instructions do not result in an interrupt if they cause an exception; they simply result in a no-op).

6.2.1 Virtual Address Formation

The first step in page identification is the expansion of the effective address into a virtual address. Again, the effective address is the 64-bit address calculated by a load, store, or cache management instruction, or as part of an instruction fetch. The virtual address is formed by prepending the effective address with a 1-bit guest space identifier, an 8-bit logical partition identifier, a 1-bit address space identifier, and a 14-bit process identifier. The resulting 88-bit value forms the virtual address, which is then compared to the virtual addresses contained in the TLB entries (note that the “IND” bit, or indirect entry identifier, is not formally considered part of the virtual address, although it does participate in entry identification and invalidation).

For instruction fetches, cache management operations, and for nonexternal P₀ storage accesses, these parameters are obtained as follows. The guest space identifier is provided by MSR[GS]. The logical partition identifier is provided by the Logical Partition ID (LPID) Register. The process identifier is contained in the Process ID (PID) Register. The address space identifier is provided by MSR[IS] for instruction fetches and by MSR[DS] for data storage accesses and cache management operations, including instruction cache management operations.

For external PID type load and store accesses, these parameters are obtained from the External PID Load Context (EPLC) or External PID Store Context (EPSC) Registers. The guest space identifier is provided by the EPLC or EPSC[EGS] field. The logical partition identifier is provided by the EPLC or EPSC[ELPID] field. The process identifier is provided by the EPLC or EPSC[EPID] field, and the address space identifier is provided by EPLC[EAC] or EPSC[EAS].

The `tlbsx[.]` instruction also forms a virtual address for software controlled searches of the TLB. This instruction calculates the effective address in the same manner as a data access instruction, but the guest space and address space identifiers, as well as the process and logical partition identifiers, are provided by fields in the MAS5 and MAS6 registers, rather than by the MSR, PID, and LPID registers (see *TLB Search Instruction (tlbsx[.])* on page 199 and *Section 6.17 Storage Control Registers (Architected)* on page 228).

Likewise, the `eratsx[.]` instruction also forms a virtual address for software controlled searches of the ERAT structures. This instruction calculates the effective address in the same manner as a data access instruction, but the guest space and address space identifiers, as well as the process identifier, are provided by fields in the MMUCR0 register, rather than by the MSR and PID registers (see *Section 12.3.3 ERAT Search Indexed (eratsx[.])* on page 496 and *Section 6.18 Storage Control Registers (Non-Architected)* on page 261). Note that the ERAT entries, unlike the TLB, do not contain the LPID value. Hence, the LPID does not participate in the search of the ERAT.

6.2.2 Address Space Identifier Convention

The address space identifier differentiates between two distinct virtual address spaces, one generally associated with interrupt-handling and other system-level code and/or data, and the other generally associated with application-level code and/or data.

Typically, user mode programs run with MSR[IS,DS] both set to 1, allowing access to application-level code and data memory pages. Then, on an interrupt, MSR[IS,DS] are both automatically cleared to 0, so that the interrupt handler code and data areas can be accessed using system-level TLB entries (that is, TLB entries with the TS field = 0). It is also possible that an operating system could set up certain system-level code and

data areas (and corresponding TLB entries with the TS field = 1) in the application-level address space, allowing user mode programs running with MSR[IS,DS] set to 1 to access them (system library routines, for example, which can be shared by multiple user mode and/or supervisor mode programs). System-level code that needs to use these areas must first set the corresponding MSR[IS,DS] field to use the application-level TLB entries, or must set up alternative system-level TLB entries.

By convention, application-level code runs with MSR[IS,DS] set to 1 and uses corresponding TLB entries with the TS = 1. Conversely, system-level code runs with MSR[IS,DS] set to 0 and uses corresponding TLB entries with TS = 0. It is possible to run in user mode with MSR[IS,DS] set to 0, and conversely to run in supervisor mode with MSR[IS,DS] set to 1, with the corresponding TLB entries being used. The only fixed requirement is that, because MSR[IS,DS] are cleared on an interrupt, there *must* be a TLB entry for the system-level interrupt handler code with TS = 0 to be able to fetch and execute the interrupt handler itself. Whether or not other system-level code switches MSR[IS,DS] and creates corresponding system-level TLB entries depends upon the operating system environment.

Programming Note: Software must ensure there is always a valid TLB entry with TS = 0 and with supervisor mode execute access permission (SX = 1) corresponding to the effective address of the interrupt handlers. Otherwise, an instruction TLB error interrupt might result upon the fetch of the interrupt handler for some other interrupt type. The registers holding the state of the routine that was executing at the time of the original interrupt (SRR0/SRR1) might be corrupted. See *CPU Interrupts and Exceptions* on page 277 for more information.

6.2.3 Exclusion Range (X-bit) Operation

One limiting property of the TLB and ERAT entries is that, for a given page size, the page start address must be aligned to the page size. This is problematic when using a mix of small and large page sizes because it requires either that the large pages are adjacent to one another or that the “gaps” between large pages are filled in with numerous smaller pages. This, in turn, requires using more TLB and/or ERAT entries to define a large, contiguous range of memory that is subject to translation. The exclusion range (X bit) function of the TLB and ERAT entries can be used to relax the requirement of starting large pages (> 4 KB) on the page size alignment.

The X bit of a TLB or ERAT entry is used to enable the creation of a variable sized “hole” at the base of the current large page (> 4 KB) entry that does not provide a match for the EPN being compared against. Normally, the least significant bits of the entry EPN for large pages do not participate in the page number address compare; they are ordinarily set to zero. When the X bit of a translation entry is set, a subset of the entry's EPN least significant bits can be set to one to define an address match exclusion range such that any effective address low enough in the page to fall into the exclusion range will not compare. This allows for large pages to be defined that possess a subregion that can be filled in with smaller page sizes, which would normally overlap (coexist within) the larger page. This allows for more efficient use of entries (especially in ERAT-only mode) because the user does not need to “fill in” with small pages up to a larger page alignment in the system memory map (potentially using more entries with this approach). It also provides for more flexibility in mixing large and small page organization in the overall system memory map.

The rules for configuring an exclusion range “hole” for a given TLB entry and placing one or more pages within the “hole” are as follows:

1. Only TLB entries with page sizes greater than 4 KB can have an exclusion range hole enabled via X = 1.
2. A virtual address to be translated that falls within the hole will not match this TLB entry.
3. The size of the hole configured must be smaller than the page size of this TLB entry.
4. The size of the hole is configurable to $2^n \times 4$ KB, where $n = 0$ to \log_2 (entry size in bytes) - 13.

5. The legal binary values of the unused EPN bits of a given TLB entry are contained in the set defined by $2^n - 1$, where $n = 0$ to \log_2 (entry size in bytes) - 13.
6. Other TLB entries of valid page sizes (less than or equal to the hole size) can be mapped into the hole.
7. Multiple other TLB entries can be mapped into the hole simultaneously.
8. Not all of the address space defined by the hole needs to be mapped by other entries.
9. Pages mapped in the hole must be page-size aligned.
10. Pages mapped in the hole must not overlap.
11. Pages mapped in the hole must be collectively fully contained within the hole.

For example, for a 64 KB page size, the EPN bits 48:51 is ordinarily set to zero because they do not participate in the virtual address matching. The set of legal values for EPN(48:51) representing the hole size would be {0000, 0001, 0011, 0111}. If software needs to create a 16 KB exclusion range at the base of the 64 KB page, it sets $X = 1$ and the EPN bits 48:49 = '00' and 50:51 = '11' (EPN bit 50 is the MSB of the 16 KB address base to exclude). Addresses above the first 16 KB, but still within the 64 KB page, match. Addresses within the first 16 KB of the 64 KB do not match. Software is then free to create four additional 4 KB virtual pages overlaying the 16 KB hole within the 64 KB page in question.

6.2.4 TLB Match Process

This virtual address is used to locate the associated entry in the TLB. The guest state identifier, logical partition identifier, address space identifier, the process identifier, and a portion of the effective address of the storage access are compared to the TGS, TLPID, TS, TID, and EPN fields, respectively, of each TLB entry.

The virtual address matches a TLB entry if the following conditions are met:

- The valid (V) field of the TLB entry is 1, and
- The thread ID (ThdID) field of the TLB entry has the bit corresponding to the issuing hardware thread set to 1, and
- The value of the guest state identifier is equal to the value of the TGS field of the TLB entry, and
- Either the value of the logical partition identifier is equal to the value of the TLPID field of the TLB entry (partition page) or the value of the TLPID field is 0 (nonguest page), and
- The value of the address space identifier is equal to the value of the TS field of the TLB entry, and
- Either the value of the process identifier is equal to the value of the TID field of the TLB entry (private page) or the value of the TID field is 0 (globally shared page), and
- Either the value of the TLB entry X-bit is 0, or the value of bits $n:51$ of the effective address (where $n = 64 - \log_2$ (page size in bytes), and page size is specified by the value of the SIZE field of the TLB entry) is greater than the value of bits $n:51$ of the EPN field in the TLB entry that are set to 1 (that is, the EA is "above" the exclusion region of the page), and
- The value of bits $0:n-1$ of the effective address is equal to the value of bits $0:n-1$ of the EPN field of the TLB entry (where $n = 64 - \log_2$ (page size in bytes), and page size is specified by the value of the SIZE field of the TLB entry). See *Table 6-1 Page Size and Effective Address to EPN Comparison* on page 175.

A TLB miss exception occurs if there is no matching direct or indirect entry in the TLB for the page specified by the virtual address (except for the **tlbsx[.]** instruction, which simply returns certain default or undefined values to the MMU Assist Registers (MAS), and for **tlbsx.**, which sets $CR[CR0]_2$ to 0). See *TLB Search Instruction (tlbsx[.])* on page 199.

Programming Note: Although it is possible for software to create multiple TLB entries that match the same virtual address, doing so is a programming error and the results are undefined.

Figure 6-1 illustrates the criteria for a virtual address to match a specific direct or indirect TLB entry, while Table 6-1 defines the page sizes associated with each SIZE field value and the associated comparison of the effective address to the EPN field.

Figure 1. Virtual Address to TLB Entry Match Process

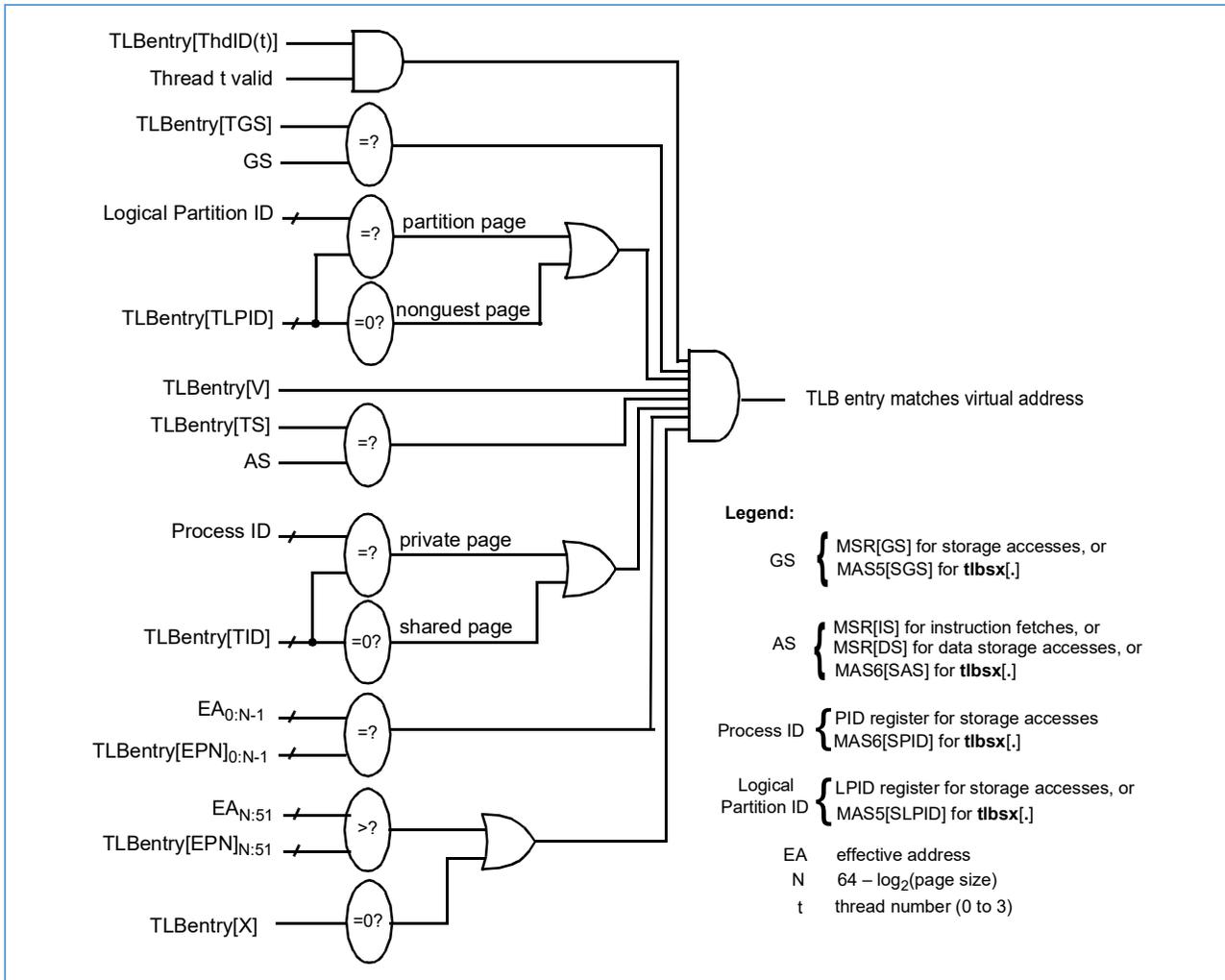


Table 1. Page Size and Effective Address to EPN Comparison

SIZE ¹	Page Size	EA to EPN Comparison
0b0000	not supported	not supported
0b0001	4 KB	EPN _{0:51} =? EA _{0:51}
0b0010	not supported	not supported
0b0011	64 KB	EPN _{0:47} =? EA _{0:47}
0b0100	not supported	not supported
0b0101	1 MB ²	EPN _{0:43} =? EA _{0:43}
0b0110	not supported	not supported
0b0111	16 MB	EPN _{0:39} =? EA _{0:39}
0b1000	not supported	not supported
0b1001	256 MB ³	EPN _{0:35} =? EA _{0:35}
0b1010	1 GB	EPN _{0:33} =? EA _{0:33}
0b1011	not supported	not supported
0b1100	not supported	not supported
0b1101	not supported	not supported
0b1110	not supported	not supported
0b1111	not supported	not supported

1. The Power ISA page sizes are defined as power of 2×1 KB sizes and represented by a 5-bit value. The page sizes supported by A2 all happen to be power of 4×1 KB sizes. For this reason, the LSB of the architected page size encoding is assumed to be zero always and is not implemented in A2.
2. This page size is supported for both direct and indirect translation entries.
3. This page size is supported for indirect translation entries only.

6.3 Address Translation

After a direct (IND = 0) TLB entry is found that matches the virtual address associated with a given storage access, as described in *Section 6.2 Page Identification* on page 170, the virtual address is translated to a real address according to the procedures described in this section. If a matching direct entry is not found, but a matching indirect entry is found, the hardware page table walker is invoked for further processing. See *Section 6.16 Hardware Page Table Walking (Category E.PT)* for a description of this process.

The RPN field of the matching direct TLB entry provides the page number portion of the real address. Let $n = 64 - \log_2(\text{page size in bytes})$ where *page size* is specified by the SIZE field of the matching TLB entry. Bits $n:63$ of the effective address (the “page offset”) are appended to bits $22:n-1$ of the RPN field to produce the 42-bit real address (that is, $RA = RPN_{22:n-1} \parallel EA_{n:63}$).

Depending on the page size, some number of the entry RPN least-significant bits are required to be set to 0 (as shown in *Table 6-2*). This is because the logic that produces the final RPN result is a logical OR between some number of bits contained in the entry RPN and the corresponding bits contained in the EA. This eliminates the need to know the page size for this calculation, which might not be readily available to logic external to the physical embodiment of the translation entries themselves.

Figure 6-2 illustrates the address translation process, while *Table 6-2* defines the relationship between the different page sizes and the real address formation.

Figure 2. Effective-to-Real Address Translation Flow

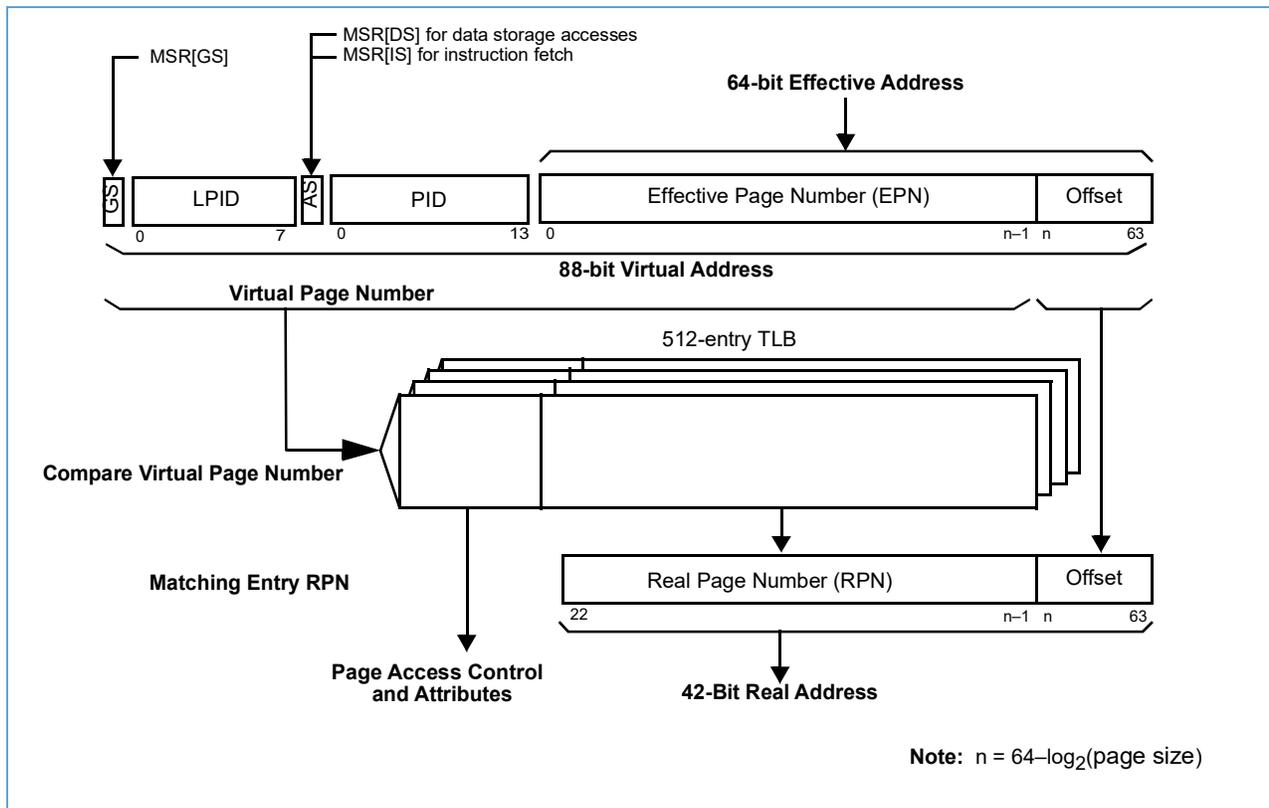


Table 2. Page Size and Real Address Formation

SIZE ¹	Page Size	RPN Bits Required to be 0	Real Address
0b0000	not supported	not supported	not supported
0b0001	4 KB	None	RPN _{22:51} EA _{52:63}
0b0010	not supported	not supported	not supported
0b0011	64 KB	RPN _{48:51} = 0	RPN _{22:47} EA _{48:63}
0b0100	not supported	not supported	not supported
0b0101	1 MB	RPN _{44:51} = 0	RPN _{22:43} EA _{44:63}
0b0110	not supported	not supported	not supported
0b0111	16 MB	RPN _{40:51} = 0	RPN _{22:39} EA _{40:63}
0b1000	not supported	not supported	not supported
0b1001	not supported	not supported	not supported
0b1010	1 GB	RPN _{34:51} = 0	RPN _{22:33} EA _{34:63}
0b1011	not supported	not supported	not supported
0b1100	not supported	not supported	not supported
0b1101	not supported	not supported	not supported
0b1110	not supported	not supported	not supported
0b1111	not supported	not supported	not supported

1. The Power ISA page sizes are defined as a power of 2×1 KB sizes and represented by a 5-bit value. The page sizes supported by A2 all happen to be power of 4×1 KB sizes. For this reason, the LSB of the architected page size encoding is assumed to be zero always and is not implemented in A2.

6.4 Access Control

After a matching TLB entry has been identified and the address has been translated, the access control mechanism determines whether the program has execute, read, and/or write access to the page referenced by the address, as described in the following sections.

6.4.1 Execute Access

The User State Execute Enable (UX) or Supervisor State Execute Enable (SX) bit of a TLB entry controls *execute* access to a page of storage, depending on the operating mode (user or supervisor) of the processor.

- User mode (MSR[PR] = 1)
- Supervisor mode (MSR[PR] = 0)

Instructions can be fetched and executed from a page in storage while in supervisor mode if the SX access control bit for that page is equal to 1. If the SX access control bit is equal to 0, instructions from that page are not fetched and are not placed into any cache as the result of a fetch request to that page while in supervisor mode.

Furthermore, if the sequential execution model calls for the execution in supervisor mode of an instruction from a page that is not enabled for execution in supervisor mode (that is, SX = 0 when MSR[PR] = 0), an execute access control exception type of instruction storage interrupt is taken. See *CPU Interrupts and Exceptions* on page 277 for more information.

6.4.2 Write Access

The User State Write Enable (UW) or Supervisor State Write Enable (SW) bit of a TLB entry controls *write* access to a page, depending on the operating mode (user or supervisor) of the processor.

- User mode (MSR[PR] = 1)

Store operations (including the store-class cache management instruction in *Table 6-3*) are permitted to a page in storage while in user mode if the UW access control bit for that page is equal to 1. If execution of a store operation is attempted in user mode to a page for which the UW access control bit is 0, a write access control exception occurs. If the instruction is an **stswx** with string length 0, no interrupt is taken and no operation is performed (see *Section 6.4.4 Access Control Applied to Cache Management Instructions* on page 178). For all other store operations, execution of the instruction is suppressed and a data storage interrupt is taken.

- Supervisor mode (MSR[PR] = 0)

Store operations (including the store-class cache management instructions in *Table 6-3*) are permitted to a page in storage while in supervisor mode if the SW access control bit for that page is equal to 1. If execution of a store operation is attempted in supervisor mode to a page for which the SW access control bit is 0, a write access control exception occurs. If the instruction is an **stswx** with string length 0, no interrupt is taken and no operation is performed (see *Access Control Applied to Cache Management Instructions* on page 178). For all other store operations, execution of the instruction is suppressed and a data storage interrupt is taken.

6.4.3 Read Access

The User State Read Enable (UR) or Supervisor State Read Enable (SR) bit of a TLB entry controls *read* access to a page, depending on the operating mode (user or supervisor) of the processor.

- User mode (MSR[PR] = 1)

Load operations (including the load-class cache management instructions in *Table 6-3*) are permitted from a page in storage while in user mode if the UR access control bit for that page is equal to 1. If execution of a load operation is attempted in user mode to a page for which the UR access control bit is 0, then a read access control exception occurs. If the instruction is a load (not including **lswx** with string length 0), execution of the instruction is suppressed and a data storage interrupt is taken. On the other hand, if the instruction is an **lswx** with string length 0, no interrupt is taken and no operation is performed (see *Access Control Applied to Cache Management Instructions*).

- Supervisor mode (MSR[PR] = 0)

Load operations (including the load-class cache management instructions in *Table 6-3*) are permitted from a page in storage while in supervisor mode if the SR access control bit for that page is equal to 1. If execution of a load operation is attempted in supervisor mode to a page for which the SR access control bit is 0, a read access control exception occurs. If the instruction is a load (not including **lswx** with string length 0), execution of the instruction is suppressed and a data storage interrupt is taken. On the other hand, if the instruction is an **lswx** with string length 0, no interrupt is taken and no operation is performed (see *Access Control Applied to Cache Management Instructions*).

6.4.4 Access Control Applied to Cache Management Instructions

This section summarizes how each of the cache management instructions is affected by the access control mechanism. Any cache management instruction that causes a protection exception that does not result in a data storage interrupt is treated as a no-op.

Any cache management instruction that is privileged and execution is attempted in user mode causes a privileged instruction exception type of program interrupt to occur instead of a data storage interrupt.

Table 6-3 summarizes the effect of access control on each of the cache management instructions. Unless otherwise noted in this table, a protection exception causes a data storage interrupt.

Table 3. Access Control Applied to Cache Management Instructions (Sheet 1 of 2)

Instruction	Treated as a Read Might Cause a Protection Violation Exception	Treated as a Write Might Cause a Protection Violation Exception	Virtualization Fault
dcba	No	Yes ¹	No
dcbf	Yes	No	Yes
dcbfep	Yes	No	Yes
dcbi	No	Yes	Yes
dcbic	Yes	No	Yes

1. **dcba**, **dcbtst**, and **dcbtstep** can cause a write access control exception but do not result in a data storage interrupt. The instruction is nop'ed.
 2. **dcbt**, **dcbtep**, or **icbt** can cause a read access control exception but do not result in a data storage interrupt. The instruction is nop'ed.
 3. **icbtls** and **icblc** require either execute or read access.

Table 3. Access Control Applied to Cache Management Instructions (Sheet 2 of 2)

Instruction	Treated as a Read Might Cause a Protection Violation Exception	Treated as a Write Might Cause a Protection Violation Exception	Virtualization Fault
dcbst	Yes	No	Yes
dcbstep	Yes	No	Yes
dcbt	Yes ²	No	No
dcbtep	Yes ²	No	No
dcbtls	Yes	No	Yes
dcbtst	No	Yes ¹	No
dcbtstep	No	Yes ¹	No
dcbtstls	No	Yes	Yes
dcbz	No	Yes	Yes
dcbzep	No	Yes	Yes
dci	No	No	No
icbi	Yes	No	Yes
icbiep	Yes	No	Yes
icblc	Yes ³	No	Yes
icbt	Yes ²	No	No
icbtls	Yes ³	No	Yes
ici	No	No	No
icswx	No	Yes	Yes
icswepx	No	Yes	Yes

1. **dcbz**, **dcbtst**, and **dcbtstep** can cause a write access control exception but do not result in a data storage interrupt. The instruction is nop'ed.
2. **dcbt**, **dcbtep**, or **icbt** can cause a read access control exception but do not result in a data storage interrupt. The instruction is nop'ed.
3. **icbtls** and **icblc** require either execute or read access.

6.5 Storage Attributes

Each TLB entry specifies a number of storage attributes for the memory page with which it is associated. Storage attributes affect the manner in which storage accesses to a given page are performed. The storage attributes (and their corresponding TLB entry fields) are:

- Write-through (W)
- Caching inhibited (I)
- Memory coherence required (M)
- Guarded (G)
- Endianness (E)
- User-definable (U0, U1, U2, U3)

All combinations of these attributes are supported except combinations that simultaneously specify a region as write-through and caching inhibited.

6.5.1 Write-Through (W)

The A2 core data cache ignores the write-through attribute. The data for all store operations is written to memory, as opposed to only being written into the data cache. If the referenced line also exists in the data cache (that is, the store operation is a “hit”), the data is also written into the data cache. An alignment exception occurs if a **dcbz** instruction targets a memory page that is either write-through required or caching inhibited. A data storage exception occurs if an **lwarx**, **ldarx**, **stwcx.**, or **stdcx.** instruction targets a memory page that is either write-through required or caching inhibited.

See *Instruction and Data Caches* on page 153 for more information about the handling of accesses to write-through storage.

6.5.2 Caching Inhibited (I)

If a memory page is marked as caching inhibited ($I = 1$), all load, store, and instruction fetch operations perform their access in memory, as opposed to in the respective cache. If $I = 0$, the page is cacheable; and the operations can be performed in the cache. An alignment exception occurs if a **dcbz** instruction targets a memory page that is either write-through required or caching inhibited. A data storage exception occurs if an **lwarx**, **ldarx**, **stwcx.**, or **stdcx.** instruction targets a memory page that is either write-through required or caching inhibited.

It is a programming error for the target location of a load, store, **dcbz**, or fetch access to caching inhibited storage to be in the respective cache; the results of such an access are undefined. It is *not* a programming error for the target locations of the other cache management instructions to be in the cache when the caching inhibited storage attribute is set. The behavior of these instructions is defined for both $I = 0$ and $I = 1$ storage.

See *Instruction and Data Caches* on page 153 for more information about the handling of accesses to caching inhibited storage.

6.5.3 Memory Coherence Required (M)

The memory coherence required (M) storage attribute is defined by the architecture to support cache and memory coherency within multiprocessor shared memory systems. If a TLB entry is created with $M = 1$, any storage accesses to the page associated with that TLB entry are indicated, using the corresponding transfer attribute interface signal, as being memory coherence required. However, the setting has no effect on the operation within the A2 core.

6.5.4 Guarded (G)

The guarded storage attribute is provided to control “speculative” access to “non-well-behaved” memory locations. Storage is said to be “well-behaved” if the corresponding real 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. As such, data and instructions can be fetched out-of-order from well-behaved storage without causing undesired side effects.

In general, storage that is not well-behaved should be marked as guarded. Because such storage might represent a control register on an I/O device or might include locations that do not exist, an out-of-order access to such storage might cause an I/O device to perform unintended operations or might result in a machine check exception. For example, if the input buffer of a serial I/O device is memory-mapped, an out-of-order or speculative access to that location might result in the loss of an item of data from the input buffer, if the instruction execution is interrupted and later re-attempted.

A data access to a guarded storage location is performed only if either the access is caused by an instruction that is known to be required by the sequential execution model, or the access is a load and the storage location is already in the data cache. After a guarded data storage access is initiated, if the storage is also caching inhibited, only the bytes specifically requested are accessed in memory, according to the operand size for the instruction type. Data storage accesses to guarded storage that is marked as cacheable can access the entire cache block, either in the cache itself or in memory.

Instruction fetch is not affected by guarded storage. While the architecture does not prohibit instruction fetching from guarded storage, system software should generally prevent such instruction fetching by marking all guarded pages as “no-execute” (UX/SX = 0). Then, if an instruction fetch is attempted from such a page, the memory access will not occur and an execute access control exception type of instruction storage interrupt results if and when execution is attempted for an instruction at any address within the page.

See *Instruction and Data Caches* on page 153 for more information about the handling of accesses to guarded storage. Also see *Partially Executed Instructions* on page 283 for information about the relationship between the guarded storage attribute and instruction restart and partially executed instructions.

6.5.5 Endian (E)

The endian (E) storage attribute controls the byte ordering with which load, store, and fetch operations are performed. Byte ordering refers to the order in which the individual bytes of a multiple-byte scalar operand are arranged in memory. The operands in a memory page with E = 0 are arranged with big-endian byte ordering, which means that the bytes are arranged with the *most*-significant byte at the lowest-numbered memory address. The operands in a memory page with E = 1 are arranged with little-endian byte ordering, which means that the bytes are arranged with the *least*-significant byte at the lowest-numbered address.

See *Byte Ordering* on page 64 for a more detailed explanation of big-endian and little-endian byte ordering.

6.5.6 User-Definable (U0–U3)

The A2 core provides four user-definable (U0–U3) storage attributes that can be used to control system-dependent behavior of the storage system. By default, these storage attributes do not have any effect on the operation of the A2 core, although all storage accesses indicate to the memory subsystem the values of U0–U3 using the corresponding transfer attribute interface signals. The specific system design can then take advantage of these attributes to control some system-level behaviors.

6.5.7 Supported Storage Attribute Combinations

Storage modes where both W = 1 and I = 1 (which would represent write-through but caching inhibited storage) are not supported. For all supported combinations of the W and I storage attributes, the G, E, and U0-U3 storage attributes can be used in any combination.

6.5.8 Aliasing

For multiple pages that are mapped to the same real address, the following rules apply:

1. If the multiple pages exist on a single processor, then:
 - The I bits must match the corresponding I bits on all pages (see note below).
 - The W bits do not need to match on all pages.

- The M bits do not need to match on all pages; however, it is then software's responsibility to maintain data coherency.
2. If the multiple pages exist on multiple processors, then:
 - The I bits do not need to match on all pages.
 - The W bit must match on all pages (Book III-E requirement).
 - The M bits do not need to match on all pages; however, it is then software's responsibility to maintain data coherency.

Note: For multiple pages that exist on a single processor which map to the same real address, the I bits do not need to match under the following conditions, which must be guaranteed by software:

1. For those pages where the I bit is zero, the page must be marked as Guarded and no execute to prevent speculative accesses.
2. For those addresses where the calculability attributes are different, software must ensure that only those pages where all I bits are the same access the overlapped real address. (Alternatively, software could manage the cache appropriately between different calculability accesses to guarantee that an access to any I = 1 is not found in the associated cache. When the architected I bit is a one, the data must not be in any level of cache.)

For example, consider a cacheable 64 K page and a noncacheable 4 K page that both map to the same real address (that is, the 4 K page maps to the last 4 K of real addresses that the 64 K page maps to). In this case, the 64 K page is marked as guarded as well as cacheable. In addition, software must ensure that, when operating in the 64 K page, no accesses are performed to the last 4 K addresses.

6.6 Translation Lookaside Buffer

The translation lookaside buffer (TLB) is the hardware resource that controls page identification and address translation, and contains protection and storage attributes. A single unified 512-entry, 4-way set-associative TLB is used for both instruction and data accesses. In addition, the A2O core implements two separate, fully-associative, smaller "shadow" TLB arrays: one for instruction fetch accesses and one for data accesses. These shadow TLBs are also referred to as "ERATs" (effective to real address translation arrays). The shadow TLBs, or ERATs, improve performance by lowering the latency for address translation and by reducing contention for the main unified TLB between instruction fetching and data storage accesses. See *Effective to Real Address Translation Arrays* on page 187 for additional information about the operation of the shadow TLB arrays.

Maintenance of TLB entries is under software control. System software determines the TLB entry replacement strategy and the format and use of any page table information (that is, in the absence of Category Embedded Page Table, or E.PT, which infers hardware-based page table walking). A TLB entry contains all of the information required to identify the page, to specify the address translation, to control the access permissions, and to designate the storage attributes.

A TLB entry is written by copying information from the MAS register fields, using the **tlbwe** instruction with MAS0[ATSEL] = 0. (That is, the **tlbwe** instructions are targeting the TLB array.) A TLB entry is read by copying the information into the MAS register fields using the **tlbre** instruction. Software can also search for specific TLB entries using the **tlbsx[.]** instruction. See *TLB Management Instructions (Architected)* on page 196 for more information about these instructions.

Each TLB entry identifies a page and defines its translation, access controls, and storage attributes. Accordingly, fields in the TLB entry fall into four categories:

- Page identification fields (information required to identify the page to the hardware translation mechanism)
- Address translation fields
- Access control fields
- Storage attribute fields

Table 4. TLB Entry Fields (Sheet 1 of 5)

TLB Word ¹	Bit ²	Field	Description
Page Identification Fields			
0	0:51	EPN	Effective Page Number (variable size, from 34 - 52 bits) Bits 0:n-1 of the EPN field are compared to bits 0:n-1 of the EA of the storage access (where n = 64-log ₂ (page size in bytes), and page size is specified by the SIZE field of the TLB entry).
0	52:53	Class ⁵	Class (2 bits) This field is used to uniquely identify entries for invalidation. The local tlbilx instruction contains extended settings for the “T” type field that are conditionally used to match against this field to select entries to be invalidated.
0	54	V	Valid (1 bit) This bit indicates that this TLB entry is valid and can be used for translation. The Valid bit for a given entry can be set or cleared with a tlbwe instruction, and can be cleared by a tlbivax instruction.
0	55	X ⁵	Exclusion Range Enable (1 bit) This bit enables the creation of a variable sized “hole” at the base of large page sizes (> 4 KB). For large pages, the unused LSBs of the EPN field are ordinarily set to zero. When the X bit is set, a subset of LSBs of the EPN can be set to ‘1’ to define an exclusion range that prevents an effective address match for this entry. For a more detailed description of this function, see <i>Section 6.2.3 Exclusion Range (X-bit) Operation</i> .
0	56:59	SIZE	Page Size (4 bits) The SIZE field specifies the size of the page associated with the TLB entry as 4 ^{SIZE} KB, where SIZE = 1, 3, 5, 7, or 10. This field is recoded to a 3-bit field in the ERAT shadow copies.
0	60:61	ThdID ⁵	Thread ID (2 bits) These bits indicate for which threads this TLB entry is valid; one bit for each processing thread. Bit 60 corresponds to thread 0, and bit 61 to thread 1. The ThdID bit for a given thread can be set or cleared with the tlbwe instruction.
0	62:63	Reserved	
0	64	ExtClass ⁵	Extended Class (1 bit) This field is used as an extension to the Class field to uniquely identify entries for invalidations. This field will be set to a zero value when tlbwe completes with MAS1[IPROT] = 0. For a more detailed description of this function, see the ECL field description in <i>Section 6.18.4 Memory Management Unit Control Register 3 (MMUCR3)</i> .
0	65	TID_NZ ⁵	Translation ID Non-Zero (1 bit) This field is used to denote when the TID field is nonzero. This field is set to a ‘1’ value when tlbwe completes with MAS1[TID] != 0; otherwise, it is cleared. It is used by TLB shadow copies that can contain less than the full 14-bit TID value. For a more detailed description of this function, see the TID_NZ field description in <i>Section 6.18.1 Memory Management Unit Control Register 0 (MMUCR0)</i> .
0	66	TGS	Translation Guest State (1 bit) This bit differentiates between guest operating system and hypervisor state translations.

Table 4. TLB Entry Fields (Sheet 2 of 5)

TLB Word ¹	Bit ²	Field	Description
0	67	TS	Translation Address Space (1 bit) This bit indicates the address space with which this TLB entry is associated. For instruction storage accesses, MSR[IS] must match the value of TS in the TLB entry for that TLB entry to provide the translation. Likewise, for data storage accesses, MSR[DS] must match the value of TS in the TLB entry. For the tbsx [.] instruction, the MMUCR0[TS] field must match the value of TS.
0	68:81	TID ³	Translation ID (14 bits) Field used to identify a globally shared page (TID = 0) or the process ID of the owner of a private page (TID <> 0).
0	82:89	TLPID ⁴	Translation Logical Partition ID (8 bits) Field used to identify a nonguest page (TLPID = 0) or the logical partition ID of a guest page (TLPID <> 0).
0	90	IPROT ⁶	Invalidate Protection (1 bit) This bit protects the TLB entry from local or global invalidations. This bit also influences the determination of the LRU algorithm for the TLB.
0	91	IND ⁷	Indirect (1 bit) When set, this bit indicates that this entry is an indirect virtual linear page table (VLPT) pointer entry used for hardware page table walking (used with category E.PT implementations only).
Storage Attribute Fields			
1	0:7	-	Reserved (8 bits) Set to 0.
1	8:9	WLC ⁵	D-Cache Way Locking Class Attribute (2 bits) L1 data-cache way locking class attribute bit used in conjunction with XUCR0[WLK] enable bit to determine the D-cache replacement management table entry to use.
1	10	ResvAttr ⁵	Reserved Attribute (1 bit) Extended page attribute bit with function reserved by the A2 core.
1	11	-	Reserved (1 bit) Set to 0.
1	12	U0	User-Definable Storage Attribute 0 (1 bit) Specifies the U0 storage attribute for the page associated with the TLB entry. The function of this storage attribute is system-dependent and has no effect within the A2O Core.
1	13	U1	User-Definable Storage Attribute 1 (1 bit) Specifies the U1 storage attribute for the page associated with the TLB entry. The function of this storage attribute is system-dependent and has no effect within the A2O Core.
1	14	U2	User-Definable Storage Attribute 2 (1 bit) Specifies the U2 storage attribute for the page associated with the TLB entry. The function of this storage attribute is system-dependent and has no effect within the A2O Core.
1	15	U3	User-Definable Storage Attribute 3 (1 bit) Specifies the U3 storage attribute for the page associated with the TLB entry. The function of this storage attribute is system-dependent and has no effect within the A2O Core.
1	16	R ⁵	Reference (1 bit) Software-managed page referenced (was accessed) attribute bit. See <i>Section 6.12 Page Reference and Change Status Management</i> for more details.
1	17	C ⁵	Change (1 bit) Software-managed page changed (was updated) attribute bit. See <i>Section 6.12 Page Reference and Change Status Management</i> for more details.

Table 4. TLB Entry Fields (Sheet 3 of 5)

TLB Word ¹	Bit ²	Field	Description
Address Translation Fields			
1	18:21	-	Reserved (4 bits) Reserved for real page number extension.
1	22:51	RPN ⁹	Real Page Number (variable size, from 18 - 30 bits) Bits 22:n-1 of the RPN field are used to replace bits 0:n-1 of the effective address to produce a portion of the real address for the storage access (where n = 64-log ₂ (page size in bytes) and page size is specified by the SIZE field of the TLB entry). Software must set unused low-order RPN bits (that is, bits n:51) to 0.
Additional Storage Attribute Fields			
1	52	W	Write-Through (1 bit) (See also the WL1 attribute.) 0 The page is not write-through (that is, the page is copy-back) in the L2 cache, and the L1 cache is always write-through. 1 The page is write-through in both the L1 and L2 cache.
1	53	I	Caching Inhibited When this bit is set, bits 12, 13, 14, and 15 of word 2 can be set. 0 The page is not caching inhibited (that is, the page might be cacheable). 1 The page is caching inhibited in all cache levels.
1	54	M	Memory Coherence Required (1 bit) 0 The page is not memory coherence required. 1 The page is memory coherence required. Note: L2 cache provides MESI support; I = 0 (architecturally cacheable), and M = 1 (Memory Coherence Required).]
1	55	G	Guarded (1 bit) 0 The page is not guarded. 1 The page is guarded.
1	56	E	Endian (1 bit). 0 All accesses to the page are performed with big-endian byte ordering, which means that the byte at the effective address is considered the most-significant byte of a multibyte scalar. 1 All accesses to the page are performed with little-endian byte ordering, which means that the byte at the effective address is considered the least-significant byte of a multibyte scalar.
Access Control Fields			
1	57	VF ⁸	Virtualization Fault (1 bit) 0 The page access is controlled by the user and supervisory access control bits. 1 Load, store, or cache management accesses to this page always result in a virtualization fault exception (which can lead to a data storage interrupt).
1	58	UX (IND = 0) SPSIZE ₀ (IND = 1)	User State Execute Enable (IND = 0) or SPSIZE₀ (IND = 1)⁹ (1 bit). 0 (IND = 0) Instruction fetch is not permitted from this page while MSR[PR] = 1, and the attempt to execute an instruction from this page while MSR[PR] = 1 will cause an execute access control exception type of instruction storage interrupt. 1 (IND = 0) Instruction fetch and execution is permitted from this page while MSR[PR] = 1.

Table 4. TLB Entry Fields (Sheet 4 of 5)

TLB Word ¹	Bit ²	Field	Description
1	59	SX (IND = 0) SPSIZE ₁ (IND = 1)	Supervisor State Execute Enable (IND = 0) or SPSIZE₁ (IND = 1)⁹ (1 bit)
			0 (IND = 0) Instruction fetch is not permitted from this page while MSR[PR] = 0, and the attempt to execute an instruction from this page while MSR[PR] = 0 will cause an execute access control exception type of instruction storage interrupt.
			1 (IND = 0) Instruction fetch and execution is permitted from this page while MSR[PR] = 0.
1	60	UW (IND = 0) SPSIZE ₂ (IND = 1)	User State Write Enable (IND = 0) or SPSIZE₂ (IND = 1)⁹ (1 bit)
			0 (IND = 0) Store operations and the dcbz instruction are not permitted to this page when MSR[PR] = 1 and will cause a write access control exception type of data storage interrupt.
			1 (IND = 0) Store operations and the dcbz instruction are permitted to this page when MSR[PR] = 1.
1	61	SW (IND = 0) SPSIZE ₃ (IND = 1)	Supervisor State Write Enable (IND = 0) or SPSIZE₃ (IND = 1)⁹ (1 bit)
			0 (IND = 0) Store operations and the dcbz and dcbi instructions are not permitted to this page when MSR[PR] = 0 and will cause a write access control exception type of data storage interrupt.
			1 (IND = 0) Store operations and the dcbz and dcbi instructions are permitted to this page when MSR[PR] = 0.
1	62	UR (IND = 0) SPSIZE ₄ (IND = 1)	User State Read Enable (IND = 0) or SPSIZE₄ (IND = 1)^{9, 10} (1 bit)
			0 (IND = 0) Load operations and the dcbt , dcbtst , dcbst , dcbf , icbt , and icbi instructions are not permitted from this page when MSR[PR] = 1 and will cause a read access control exception. Except for the dcbt , dcbtst , and icbt instructions, a data storage interrupt will occur.
			1 (IND = 0) Load operations and the dcbt , dcbtst , dcbst , dcbf , icbt , and icbi instructions are permitted from this page when MSR[PR] = 1.

Table 4. TLB Entry Fields (Sheet 5 of 5)

TLB Word ¹	Bit ²	Field	Description
1	63	SR (IND = 0) RPN ₅₂ (IND = 1)	Supervisor State Read Enable (IND = 0) or RPN₅₂ (IND = 1)^{9, 11} (1 bit)
			0 1

Notes:

1. “TLB Word” for TLB entries refers to a functional grouping of the fields into page identification fields. For ERAT entries, this refers to not only a functional grouping of fields, but also to the **eratwe** and **eratre** instruction word select (WS) field in 64-bit mode.
2. The “Bit” value in this column for TLB entries is for reference only, because these fields are transferred via **tlbwe** and **tlbre** instructions using the MMU Assist Registers (MAS), and the values in this column have no correlation to the bit numbering of the MAS registers. For ERAT entries, the contents of this column indicate bit assignments in the source (RS) or target register (RT) for the **eratwe** and **eratre** instructions respectively, in 64-bit mode.
3. The TID field contains 14 bits in TLB entries to fully and uniquely identify them with respect to the current process ID or the EPLC or EPSC registers EPID values. For ERAT entries, only 8 bits of the TID field are implemented (for certain settings of the MMUCR1 register). When an ERAT miss is resolved from the TLB, the least significant 8 bits of the TLB TID field are stored into the ERAT TID field. Supervisory software must guarantee uniqueness in the 8-bit TID field in the ERAT arrays to the extent necessary to avoid multi-hit scenarios. Refer also to *Section 6.18.2 Memory Management Unit Control Register 1 (MMUCR1)* for a description of the ICTID, ITTID, DCTID, and DTTID bits that affect this ERAT function.
4. The TLPID field does not exist in the ERAT entries (that is, ERAT entries are not tagged with the logical partition ID). Supervisory software must guarantee that the ERAT entries contain translations from only one logical partition at a time.
5. These fields are implementation specific (nonarchitected) fields.
6. The IPROT bit exists in addition to the ExtClass field in TLB entries. ERAT entries are protected by using the ExtClass field only (that is, the IPROT bit is not implemented in shadow copies).
7. The IND bit exists only in TLB entries; it is not implemented in ERAT shadow copies.
8. The Virtualization Fault (VF) bit exists only in TLB and D-ERAT entries; it is not implemented in the I-ERAT.
9. The function of these bits is dependent on if this entry is a direct (IND = 0) or indirect (IND = 1) type entry.
10. The SPSIZE₄ function for indirect (IND = 1) is treated as reserved for this implementation because sub-page sizes are a power of 4 subset of the architected power of 2 sub-page sizes.
11. This bit is used to store the RPN₅₂ LSB for indirect (IND = 1) entries in this implementation, which correlates to the MAS_{3RPNL[52]} field.

6.7 Effective to Real Address Translation Arrays

The A2 core implements two fully-associative effective to real address translation (ERAT) arrays also called shadow TLB arrays): one for instruction fetches and one for data accesses. These arrays “shadow” the value of a subset of the entries in the main, unified TLB (the UTLB in the context of this discussion). This subset of TLB entries contained in the ERAT arrays is referred to as “TLB lookaside information” in the architecture. The purpose of the ERAT arrays is to reduce the latency of the address translation operation and to avoid contention for the UTLB array between instruction fetches and data accesses.

The instruction ERAT (I-ERAT) contains 16 entries, while the data ERAT (D-ERAT) contains 32 entries, and all entries are shared between the four A2 processing threads. There is no latency associated with accessing the ERAT arrays, and instruction execution continues in a pipelined fashion as long as the requested address is found in the ERAT. If the requested address is not found in the ERAT, the instruction fetch or data storage access is automatically stalled while the address is looked up in the UTLB. If the address is found in a direct entry (IND = 0) residing in the UTLB, the penalty associated with the miss in the I-ERAT shadow array is 12 cycles, and the penalty associated with a miss in the D-ERAT shadow array is 19 cycles. If the address

lookup finds no direct entries, but does find an indirect entry ($IND = 1$), then the address is forwarded to the hardware page table walker for page table lookup. If the address also misses the indirect entry lookup in the UTLB, an instruction or data TLB miss exception is reported.

When operating in MMU mode, the on-demand replacement of entries in the ERATs is managed by hardware in a pseudo least-recently-used (LRU) fashion. Upon an ERAT miss that leads to a UTLB hit, the hardware automatically casts-out the oldest entry in the ERAT and replaces it with the new translation. 

An ERAT entry can be written directly by hypervisor level software by copying information from a [GPR](#) and the MMUCR0 fields, using a series of two **eratwe** instructions (assuming 64-bit operation). An ERAT entry is read by copying the information into a GPR and the MMUCR0 fields, using a series of two **eratre** instructions. Software can also search for specific ERAT entries using the **eratsx[.]** instruction. See *TLB Management Instructions (Architected)* on page 196 for more information about these instructions.

The **eratwe** instruction with the $WS = 3$ setting is used in the A2 implementation to set a hardware LRU watermark register for each of the ERAT facilities. This can be leveraged directly in certain kernel applications to “reserve” some number of translation entries for the kernel to be “immune” to replacement, especially with a backing hardware MMU TLB replacement scheme. This feature allows for “pinning” of some number of ERAT entries above the watermark value that are managed by software. The entries at or below the watermark are candidates for hardware replacement via normal LRU selection. See *Section 6.7.5 ERAT LRU Replacement Watermark* for more details on this feature.

6.7.1 ERAT Context Synchronization

In MMU mode ($CCR2[NOTLB] = 0$) where the ERATs are backed by the unified TLB, the hardware can conditionally invalidate all entries of extended class zero ($ExtClass = 0$) in both of the ERATs upon execution of certain ERAT context-altering instructions. This set of instructions includes: **sc**, **ehpriv**, **mtmsr**, **mtpidr**, **mtlpidr**, **rfi**, **rfdi**, **rfmci**, **rfgi**, and **isync**.

In MMU mode ($CCR2[NOTLB] = 0$), the conditional invalidation of ERAT entries (with $ExtClass = 0$) in the event of ERAT context-altering instructions is controlled by the configuration bits $MMUCR1[CSINV]$. Software can choose to prevent any context-altering invalidations, allow all ERAT context-altering instructions to flush all nonprotected entries, or can allow for the **isync** instruction to be excluded from the set of events that flush nonprotected entries. In ERAT-only mode ($CCR2[NOTLB] = 1$), the ERAT entries are not invalidated as the result of these instructions, and the $MMUCR1[CSINV]$ field is effectively disabled. See *Section 6.18.2 Memory Management Unit Control Register 1 (MMUCR1)* on page 264 for a detailed description of these register bits.

This context-altering invalidation disable feature was added to A2 because, unlike some previous generation processors, the ERAT entries are tagged with certain context-specific information (GS, AS, and 8 PID bits, but not LPID), and therefore overly generous invalidations of all nonprotected ERAT entries due to context synchronizing events might not be required by the system software.

Note that there are other “context changing” operations that do not cause automatic context synchronization in the hardware. For example, execution of a **tlbwe** instruction can change the UTLB contents, but is not one of the ERAT context-altering instructions listed above and does not necessarily invalidate or otherwise update the ERAT entries. (A **tlbwe** instruction *might*, however, back-invalidate certain ERAT entries; see *Section 6.7.6 ERAT (TLB Lookaside Information) Coherency and Back-Invalidation* for details.) For changes to the entries in the UTLB (or to other address-related resources such as the PID) to be definitely reflected in the ERATs, software must ensure that either a context-synchronizing operation that leads to ERAT invalidation occurs before any attempt to use any address associated with the updated UTLB entries (either the old or new contents of those entries), or that the corresponding ERAT entries are invalidated by use of the **tlbilx** or

eratilx instructions. By invalidating the ERAT arrays, a context-altering instruction forces the hardware to refresh the ERAT entries with the updated information in the UTLB as each memory page is accessed (when enabled by the appropriate MMUCR1[CSINV] setting).

Programming Note: Of the items in the preceding list of ERAT invalidating operations, the machine check interrupt is not architecturally required to be context synchronizing, and thus is not guaranteed to cause invalidation of any ERAT arrays on implementations other than the A2O Core. Consequently, software that is intended to be portable to other implementations should not depend on this behavior and should insert the appropriate architecturally-defined context synchronizing operation as necessary for desired operation.

6.7.2 ERAT Reset Behavior

During reset, the instruction and data ERATs are first flushed and then loaded with two initial entries that perform the following functions:

- Map the reset code and data page and set I = 1, G = 1.
- Map the first 4 K page of effective address space (“page 0”) that contains the initial exception vector addresses and set ExtClass = 1.

The LRU watermark registers for both ERATs are also loaded with an initial value just below the two boot entries. See *Section 4.2 A2 Core State After Reset* on page 149 for details regarding the boot entry contents, and so forth

After reset, both of the ERAT LRU pointers are set to 0 (the first nonvalid entry). The LRU pointer (or simply “LRU”) is physically different from the LRU watermark register. The LRU value is subsequently incremented toward the watermark value until there are no nonvalid slots left in the ERAT. At this point, and as long as all slots at or below the watermark value in the ERAT are occupied by valid entries, normal pseudo-LRU replacement policy takes effect for those entry numbers at or below the watermark.

6.7.3 Atomic Update of ERAT Entries

In previous embedded implementations, carefully planned software sequences and/or software locking was required when updating TLB entries because of the partial updates to the entries that occur when writing two or more parts of the entry. In the A2 design, each of the ERAT caches include four (1 per thread) 64-bit RPN registers that are updated upon **eratwe** of the RPN portion (WS = 1). Both halves of the ERAT entry are then updated atomically when **eratwe** is executed with WS = 0 (EPN portion). The value written into the RPN portion of the entry is the data most recently written to the RPN holding register. This allows two or more processing threads to update ERAT entries simultaneously (as long as the entry indexes are different, or when the round-robin increment mode described below is enabled).

6.7.4 ERAT LRU Round-Robin Replacement Mode

Both of the ERAT entities contain a physical LRU mechanism for hardware replacement from the optional MMU TLB. Two configuration mode bits (MMUCR1[IRRE] and MMUCR1[DRRE]) are used to change the behavior of the I-ERAT and D-ERAT LRUs, the **eratwe** instruction, and the TLB reloads in the round-robin replacement mode. In this mode, the LRU behaves as an atomically incrementing entry index for the **eratwe** instruction, rather than using the RA register as the entry index. The ERAT LRU index number is incremented (the mod number of entries below the watermark is described in *Section 6.7.5*) in a round-robin fashion each time the effective (WS = 0) portion of the entry is written. In this way, multiple threads can update ERAT entries in the same hardware structure without the need for software locking between threads for this shared resource. Likewise, the LRU behaves as an atomically incrementing entry index for TLB reload events that

occur in MMU mode. The ERAT LRU index number is incremented (the mod number of entries below the watermark is described *Section 6.7.5*) in a round-robin fashion each time an entry is written as the result of a TLB reload.

6.7.5 ERAT LRU Replacement Watermark

The **eratwe** instruction with a $WS = 3$ setting is used in the A2 implementation to set a hardware LRU watermark register for each of the ERAT facilities. This can be leveraged directly in certain kernel applications to reserve some number of translation entries for the kernel to be immune to replacement, especially with a backing hardware MMU TLB replacement scheme. This feature allows for pinning of some number of ERAT entries above the watermark value, which are managed by software and are subject to the atomic update property of the threadwise RPN holding registers mentioned in *Section 6.7.3*. The entries at or below the watermark are candidates for hardware replacement through normal LRU selection or through round-robin replacement as described in *Section 6.7.4*. For example, hypervisor software can choose to set up a shared entry for all threads above the watermark for the interrupt vector code that remains resident in the ERATs.

All unprotected entries in the ERAT arrays can be invalidated as the result of an ERAT context-altering instruction (such as an **isync**, and so forth, when enabled by $MMUCR1[CSINV]$) or as the result of a local **eratilx** with the $T = 0$ (invalidate all in the partition) setting. Note that the ERAT watermark register values themselves are not affected by context-synchronizing events. In addition, certain groups of entries can be invalidated using other settings for the **eratilx** or **tlbilx** instruction "T" field or as the result of other invalidation events dependent on ERAT-only versus MMU mode of operation. For the entries to be truly pinned above the watermark such that they are immune to such invalidations, the entries must be created with a nonzero extended class setting (using $MAS1[IPROT] = 1$ and $MMUCR3[ECL] = 1$ for **tlbwe** insertion or $MMUCR0[ECL] = 1$ for **eratwe** insertion).

Programming Note: It is recommended that the watermark value never be set lower than the value of n , where n equals the number of threads allowed to run software on this implementation, minus one. Normally, with four threads running, the watermark value should not be set below a value of 3. This allows for all threads to make forward progress when running in disjoint pages. Setting the watermark to a value too low results in multiple threads contending for too few hardware resources (particularly in TLB mode where TLB reloads into the ERATs are targeting entries below the watermark) and can result in poor system performance and/or livelock (no forward progress).

Programming Note: Writing the ERAT LRU replacement watermark value by executing **eratwe** with the $WS = 3$ setting has the effect of clearing the LRU replacement algorithm such that the LRU points to entry zero. This guarantees that the LRU points to an entry number less than or equal to the new watermark value. After this, the LRU resumes normal operation and is limited to values at or below the new watermark value.

6.7.6 ERAT (TLB Lookaside Information) Coherency and Back-Invalidation

There is considerable flexibility in establishing ERAT entries that are immune to back-invalidations caused by TLB state modifications. ERAT entries created with $ExtClass = 1$ are generally immune to such back-invalidations (except in the specific **tlbwe** back-invalidate scenario described below). Such entries can be installed directly by executing an **eratwe** instruction with $MMUCR0[ECL] = 1$ or can be installed by a reload from the TLB of an entry that has both $IPROT = 1$ and $ExtClass = 1$. The latter creates "sticky" ERAT entries with $ExtClass = 1$ at or below the watermark pointer that are immune to **tlbilx**, **tlbivax**, **eratilx**, and **erativax** invalidations, but that can be overwritten by new TLB reloads (as determined by the ERAT LRU) or by **eratwe** execution. If volatility of the ERAT entries installed via TLB reloads is desired, the corresponding TLB entries should be created via **tlbwe** instructions with $MMUCR3[ECL] = 0$. This ensures that ERAT entries are always

installed with ExtClass = 0 regardless of the value of the TLB entry IPROT bit. TLB entries created via page table translation (that is, by the hardware page table walker) are always created with IPROT = 0 and ExtClass = 0; hence, the resulting ERAT entries are created with ExtClass = 0.

The subset of TLB entries contained in the ERAT arrays is referred to as “TLB lookaside information” in the architecture. There are certain architectural requirements regarding the coherency of ERAT entries with respect to the TLB entries that they shadow. This coherency largely depends on the value of the TLBnCFG[HES] bit associated with a given implementation’s TLB. If TLBnCFG[HES] = 0, lookaside information for the associated TLB is kept coherent with the TLB and is invisible to software. If TLBnCFG[HES] = 1, lookaside information is not required to be kept coherent with the TLB.

In the case of the A2 processor, TLB0CFG[HES] = 1, and the ERAT lookaside information is not necessarily kept coherent with the entries residing in the TLB. Only under the following conditions is the corresponding ERAT lookaside information kept coherent with the TLB:

1. Writing the MMUCSR0[TLB0_FI] to a ‘1’ value invalidates all unprotected (ExtClass = 0) lookaside information (in addition to all IPROT = 0 TLB entries).
2. Executing **tlbilx** or **tlbivax** instructions invalidates unprotected (ExtClass = 0) lookaside entries corresponding to the TLB entry values that they are specified to invalidate, as well as those TLB entries that would have been invalidated except for their IPROT = 1 value (when the corresponding ERAT entry was created with ExtClass = 0).
3. Executing a **tlbwe** instruction in hypervisor state (MSR[PR] = 0 and MSR[GS] = 0) that does not result in an exception back-invalidates a corresponding lookaside ERAT entry (regardless of the state of the ERAT entry ExtClass field) when all of the following conditions are true:
 - a. MMUCR1[TLBWE_BINV] = 1.
 - b. MAS0[HES] = 0.
 - c. MAS0[ESEL] is selecting a target TLB entry to be overwritten that is currently valid (V = 1) and that is a direct TLB entry (IND = 0). Note that only direct TLB entries have shadow ERAT copies.
 - d. Either of the following conditions are met:
 - (1) The MAS0[WQ] used by the **tlbwe** instruction is 0b00 or 0b11 (write always).
 - (2) The MAS0[WQ] used by the **tlbwe** instruction is 0b01 (TLB write conditional) and the TLB reservation for the thread executing the **tlbwe** exists.
 - e. The ERAT entry TGS and TS values match those of the TLB entry being overwritten.
 - f. The ERAT entry EPN[0:m] matches EPN[0:m] of the TLB entry being overwritten, where $m = 63 - \log_2(\text{ERAT entry page size in bytes})$.
 - g. The ERAT entry X = 0, or the EPN[n:51] of the TLB entry being overwritten, is greater than the ERAT entry EPN[n:51], where $n = 64 - \log_2(\text{ERAT entry page size in bytes})$.
 - h. The ERAT entry TID field matches the TID[6:13] value of the TLB entry being overwritten.
 - i. The ERAT entry THDID field matches the TID[2:3] value of the TLB entry being overwritten, or MMUCR1[I/DTTID] = 0.
 - j. The ERAT entry CLASS field matches the TID[0:1] value of the TLB entry being overwritten, or MMUCR1[I/DCTID] = 0.
 - k. The ERAT entry TID_NZ field equals the logical OR of all of the TID[0:13] bits of the TLB entry being overwritten.

The **tlbwe** with MAS0[HES] = 0 back-invalidate scenario above is intended to represent software overwriting a specific, valid TLB entry whose virtual address matches that of a shadow ERAT copy. This is particularly useful when invalidating a TLB entry with IPROT = 1 by executing **tlbwe** with MAS1[V] = 0. However, context synchronizing invalidation of the entire ERAT is not desirable by system software because the shadow ERAT copy might have been created by a TLB reload with ExtClass = 1. This function is controlled by MMUCR1[TLBWE_BINV]. Setting this bit low allows for modification of a specific TLB entry with an installed, valid ERAT shadow copy representing a page from which instructions are currently being fetched (that is, modifying the TLB page where software is executing) without back-invalidating the ERAT copy. This is sometimes done at system software initialization time (a hand off from firmware to the hypervisor, for example).

6.7.7 ERAT External PID (EPID) Context and Instruction Dependencies

There are certain virtual address ambiguities associated with the A2 ERAT entries that can lead to unintended translation results (including multi-hit error scenarios and unintended sharing of entries), particularly when using the external PID load and store context registers (EPLC and EPSC) and the associated external PID instruction set. The A2 ERAT entries do not contain the TLPID (logical partition ID) and, under certain conditions, might contain only a subset of the TID value from the associated UTLB entries (see *Section 6.18.2 Memory Management Unit Control Register 1 (MMUCR1)* for descriptions of the ITTID, DTTID, ICTID, and DCTID bits). Because of these ambiguities, it is possible for the ERAT structures to contain alias entries when the TLPID and/or the upper bits of the TID are ignored. To mitigate these types of ERAT ambiguities, the Class field is used in the D-ERAT to differentiate entries that were created as the result of external PID loads and stores versus those created as the result of normal, nonexternal PID loads and stores. Because of this, the use of external PID loads and stores by software is considered to be a mutually exclusive function to software using the Class field as part of the TID (by setting MMUCR1[DCTID] = 1).

When an ERAT entry is created by using the **eratwe** instruction, software has full control over setting the value of the ERAT entry Class field. When an ERAT entry is created due to a reload from the UTLB, the hardware sets the ERAT Class value depending on the type of operation that caused the reload. The MMUCR1 register ICTID, DCTID, and DCCD bits also affect this behavior. *Table 6-5* summarizes the UTLB to ERAT reload Class field values as a function of operation type and MMUCR1 configuration bits.

Table 5. ERAT Class Field Reload Value For UTLB Hits

Operation Type ¹	MMUCR1 [ICTID]	MMUCR1 [DCTID]	MMUCR1 [DCCD]	I-ERAT/D-ERAT Class Reload Value ²
I-ERAT fetch	0	-	-	TLBE.Class[0:1]
I-ERAT fetch	1	-	-	TLBE.TID[0:1]
D-ERAT non-EPID load/store	-	0	0	0b0 TLBE.Class[1]
D-ERAT non-EPID load/store	-	0	1	TLBE.Class[0:1]
D-ERAT non-EPID load/store	-	1	-	TLBE.TID[0:1]
D-ERAT EPID load	-	0	0	0b10
D-ERAT EPID store	-	0	0	0b11
D-ERAT EPID load/store	-	0	1	TLBE.Class[0:1]
D-ERAT EPID load/store	-	1	-	TLBE.TID[0:1]

1. This is the original operation type that caused an ERAT miss request to the UTLB, resulting in a UTLB hit reload.
2. This is the value loaded into the appropriate ERAT structure Class field as the result of a UTLB hit reload. Values written to the UTLB entry Class field as the result of page table translation might be different from that shown here. See *Table 6-14 TLB Update After Page Table Translation* on page 226 for UTLB entry values after a page table translation.

When translations occur in the I-ERAT due to instruction fetches, the Class field is not used as part of the compare function (assuming MMUCR1[ICTID] = 0). When translations occur in the D-ERAT, however, the Class field *is* used as part of the compare function (assuming MMUCR1[DCCD] = 0). When a D-ERAT translation occurs due to a normal, non-EPID load or store, the Class field compare value is set to 0b0x (where x = don't care). When a D-ERAT translation occurs due to an EPID load execution, the Class field compare value is set to 0b10. Finally, when a D-ERAT translation occurs due to an EPID store execution, the Class field compare value is set to 0b11. In this way, translations in the D-ERAT avoid aliasing across entries intended for normal load/stores versus those intended for external PID loads versus those for external PID stores.

Updating the EPLC and EPSC registers via **mtspr** instructions have an associated impact on the D-ERAT contents as well. Updating the EPLC or EPSC registers has the side affect of generating an immediate class-based invalidate to the D-ERAT structure. Updating the EPLC register generates a Class = 2 invalidate of all nonprotected (ExtClass = 0) D-ERAT entries containing a Class value of 2. Likewise, updating the EPSC register generates a Class = 3 invalidate of all nonprotected (ExtClass = 0) D-ERAT entries containing a Class value of 3. For both of these D-ERAT invalidate events, the ThdID field of each respective D-ERAT entry is compared against the thread ID of the processing thread executing the **mteplc** or **mtepsc** instruction.

Even with these hardware mechanisms in place, D-ERAT entry aliases are still possible without certain software restrictions and the use of programming techniques to avoid such aliases. An ERAT alias scenario can occur in the D-ERAT, for example, when two different processing threads execute an external PID load instruction and their respective EPLC registers contain similar virtual addresses, differing only by the ELPID value. The first thread to execute its external PID load could result in a UTLB reload of a D-ERAT entry (with Class = 2, and ThdID=0b11) that happens to be alias for the second thread's external PID load instruction, resulting in an unintended translation of the second thread's load instruction.

It is considered a programming error to install two or more identical UTLB entries with respect to the virtual address. It is also the responsibility of system software to avoid the ERAT aliasing described above. Some possible software solutions to the ERAT aliasing problem include (but are not limited to) the following:

1. Software locking between threads when using external PID load and stores, such that it is impossible for two or more threads to unintentionally share an EPID load or store entry.
2. Ensuring uniqueness in the virtual address components of the threadwise EPLC and/or EPSC registers outside of the ELPID and nonrepresented bits of the EPID.
3. Enforcing a policy of intentional sameness in all EPLC[ELPID] fields, and/or all EPSC[ELPID] fields (so that sharing of Class = 2 or Class = 3 entries between threads is an intentional phenomenon, and so forth).
4. Using software installed UTLB entries with unique ThdID fields intended specifically for external PID load or store instructions issued by a specific thread or group of threads (assuming MMUCR1[DTTID] = 0).

It should be noted that supervisory software has the availability of the **eratilx** T = 4, 5, 6, or 7 instruction to assist in removing ERAT entries containing a particular Class field while using the techniques listed above, or for other appropriate applications.

6.8 Logical to Real Address Translation Array (Category E.HV.LRAT)

This processor supports the Power ISA category Embedded.Page Table (E.PT) and the embedded MMU Architecture Version 2.0 (MAV 2.0). Because this processor also supports the Embedded.Hypervisor (E.HV) category, the Embedded.Hypervisor.LRAT (E.HV.LRAT) category is also required and supported. Because of

this, hypervisor software must always ensure that at least one valid logical to real address translation (LRAT) entry exists. The A2 core implements an 8-entry, fully-associative logical to LRAT array in support of E.HV.LRAT.

When an implementation supports Category Embedded.Hypervisor (as the A2 does), only the hypervisor knows about the actual real address allocation in the system, and the guest operating system view of real addresses becomes an intermediate level of translation termed “logical” addresses. The purpose of the LRAT array is to provide a structure in which the hypervisor can assign mappings from these “logical” addresses to the actual real addresses. The LRAT structure is the primary implementation component in support of the architecture’s Category Embedded.Hypervisor.LRAT.

The LRAT array allows a guest operating system to avoid the performance penalty of always having to trap to the hypervisor when the guest needs to install guest-specific translation entries into the unified TLB array. In some cases, the logical to real translation happens when guest supervisory code executes a **tlbwe** instruction. The MAS3[RPNL] and MAS7[RPNU] fields that were setup by the guest are interpreted by hardware as a logical page number (LPN) and used to initiate a fully-associative lookup in the LRAT entries (previously setup by the hypervisor). If the logical page is found to be resident in the LRAT, the logical page number is translated to a real page number (using the RPN contents of the matching LRAT entry) before finally being written into the TLB entry. If the logical page is not found in the LRAT, or if multiple matching LRAT entries are found, an LRAT miss exception occurs.

The logical to real translation can also happen when a page table entry (PTE) translation occurs as the result of a guest installed (TGS = 1) indirect entry (IND = 1). With the LRAT, the guest operating system can directly manage its own page table. In this case, if the associated indirect TLB entry contains TGS = 1, a translation that finds a matching PTE results in the RPN field of the PTE being treated as an LPN, and the LPN is translated via the LRAT. In this case, the resulting RPN from the LRAT is loaded into the TLB in place of the LPN from the page table. If there is an LRAT miss on this LPN translation, or if multiple matching LRAT entries are found, an LRAT miss exception occurs. When this exception occurs, the LPN and the associated size information from the PTE are saved in the Logical Page Exception Register (LPER). With this LPER information, the hypervisor can load the missing LRAT entry and the instruction that caused the exception can be re-executed.

The logical page number taken from MAS7[RPNU] and MAS3[RPNL] for **tlbwe** instructions, or from the PTE[ARPN] for page table translations, matches an LRAT entry if the following conditions are met:

- The valid (V) field of the LRAT entry is 1, and
- Either the value of the logical partition identifier is equal to the value of the LPID field of the LRAT entry (partition entry), or the value of the LRAT entry LPID field is 0 (shared entry), and
- Either the value of the LRAT entry X-bit is 0, or the value of bits n:43 of the logical page number (where $n = 64 - \log_2(\text{page size in bytes})$ and page size is specified by the value of the SIZE field of the LRAT entry) is greater than the value of bits n:43 of the LPN field in the LRAT entry that are set to 1 (that is, the logical address is “above” the exclusion region of the logical page), and
- The value of bits 22:n-1 of the logical page number is equal to the value of bits 22:n-1 of the LPN field of the LRAT entry (where $n = 64 - \log_2(\text{page size in bytes})$ and page size is specified by the value of the SIZE field of the LRAT entry).

The LRAT entries are configured by hypervisor level software using **tlbwe** instructions with MAS0[ATSEL] = 1 (that is, the **tlbwe** instructions are targeting the LRAT array). The MAS registers are used in much the same manner as they are when writing TLB entries, with the exception that MAS2[EPN] is interpreted as the LPN field of the LRAT entries. There is also a different set of page sizes associated with the LRAT entries than those used for the TLB (or ERAT) entries. These logical page sizes, also termed logical “sector” sizes,

include: 1 MB, 16 MB, 256 MB, 1 GB, 4 GB, 16 GB, 256 GB, and 1 TB. These logical pages, or sectors, can be sized by the hypervisor to encompass a single large effective page or many smaller effective pages mapped within the larger logical sector.

The page access control and storage attributes associated with the TLB and effective address pages are not contained in the LRAT entries (the LRAT performs only logical page identification and real page address translation at the logical page or “sector” size). The LRAT entries do, however, implement an exclusion range (X-bit) feature, similar to that of the TLB and ERAT entries.

Table 6. LRAT Entry Fields (Sheet 1 of 2)

LRAT Word ¹	Bit ²	Field	Description
Logical Page Identification Fields			
0	0:21	—	Reserved (22 bits) Not used in the A2 implementation.
0	22:43	LPN	Logical Page Number (variable size, from 12- 22 bits) Bits 22:n–1 of the LPN field are compared to bits 22:n–1 of the LPN contained in MAS3.RPNL and MAS7.RPNL for tlbwe instructions, or contained in the page table entry for page table translations (where n = 64–log ₂ (page size in bytes, and page size is specified by the SIZE field of the LRAT entry).
0	44:53	—	Reserved (10 bits) Not used in the A2 implementation.
0	54	V	Valid (1 bit) This bit indicates that this LRAT entry is valid and can be used for translation. The Valid bit for a given entry can be set or cleared with a tlbwe instruction with MAS0[ATSEL] = 1.
0	55	X ³	Exclusion Range Enable (1 bit) This bit enables the creation of a variable sized “hole” at the base of large page sizes (> 1 MB). For large pages, the unused LSBs of the LPN field are ordinarily set to zero. When the X bit is set, a subset of LSBs of the LPN can be set to ‘1’ to define an exclusion range that prevents a logical address match for this entry. For a more detailed description of this function, see <i>Section 6.2.3 Exclusion Range (X-bit) Operation</i> .
0	56:59	SIZE	Page Size (4 bits) The SIZE field specifies the size of the page associated with the LRAT entry as 4 ^{SIZE} KB, where SIZE = 5, 7, 9, 10, 11, 12, 14, or 15.
0	60:81	—	Reserved (22 bits) Not used in the A2 implementation.
0	82:89	LPID	Logical Partition ID (8 bits) Field used to identify the logical partition ID of this LRAT entry. A value of 0 in this field provides for a “wildcard” match of any LPID value for a given tlbwe or page table translation.
0	90:91	—	Reserved (2 bits) Not used in the A2 implementation.

Notes:

1. “LRAT Word” for LRAT entries refers to a functional grouping of the fields into page identification fields.
2. The “Bit” value in this column for LRAT entries is for reference only because these fields are transferred via **tlbwe** and **tlbre** instructions using the MMU Assist Registers (MAS), and the values in this column have no correlation to the bit numbering of the MAS registers.
3. These fields are implementation specific (nonarchitected) fields.

Table 6. LRAT Entry Fields (Sheet 2 of 2)

LRAT Word ¹	Bit ²	Field	Description
Address Translation Fields			
1	0:21	—	Reserved (22 bits) Not used in the A2 implementation.
1	22:43	RPN	Real Page Number (variable size, from 12 - 22 bits) Bits 22:n-1 of the RPN field are used to replace bits 22:n-1 of the LPN to produce a portion of the real address for the storage access (where n = 64-log ₂ (page size in bytes) and page size is specified by the SIZE field of the TLB entry). Software must set unused low-order RPN bits (that is, bits n:43) to 0.
1	44:63	—	Reserved (20 bits) Not used in the A2 implementation.

Notes:

1. "LRAT Word" for LRAT entries refers to a functional grouping of the fields into page identification fields.
2. The "Bit" value in this column for LRAT entries is for reference only because these fields are transferred via **tlbre** and **tlbre** instructions using the MMU Assist Registers (MAS), and the values in this column have no correlation to the bit numbering of the MAS registers.
3. These fields are implementation specific (nonarchitected) fields.

6.9 TLB Management Instructions (Architected)

To enable software to manage the TLB, a set of TLB management instructions is implemented within the A2 core. These instructions are described briefly in the sections that follow, and in detail in *Section 12 Implementation Dependent Instructions* on page 475. In addition, the interrupt mechanism provides resources to assist with software handling of TLB-related exceptions. One such resource is Save/Restore Register 0 (SRR0), which provides the exception-causing address for instruction TLB error and instruction storage interrupts. Another resource is the Data Exception Address Register (DEAR), which provides the exception-causing address for data TLB error and data storage interrupts. Finally, the Exception Syndrome Register (ESR) provides bits to differentiate amongst the various exception types that can cause a particular interrupt type. See *CPU Interrupts and Exceptions* on page 277 for more information about these mechanisms.

All of the TLB management instructions are supervisor or hypervisor privileged to prevent user mode programs from affecting the address translation and access control mechanisms. *Table 6-7* on page 196 shows the privilege levels of the various TLB management instructions.

Table 7. TLB Management Instruction Privilege Levels (Sheet 1 of 2)

Instruction	Privilege ¹	Notes
tlbre	hypervisor	TLB entry real addresses that can be returned are hypervisor privileged data.
tlbre	supervisor	Guest operating system supervisory code can install guest TLB entries that "hit" in the LRAT facility without hypervisor intervention.
tlbsx[.]	hypervisor	TLB entry real addresses that can be returned in the event of a search "hit" are hypervisor privileged data.

1. Because these instructions depend on data in the MAS registers, these instructions are executable only in MMU mode (CCR[NOTLB] = 0). Any attempt to execute one of these instructions while in ERAT-only mode (CCR2[NOTLB] = 1) results in an illegal instruction exception.

Table 7. TLB Management Instruction Privilege Levels (Sheet 2 of 2)

Instruction	Privilege ¹	Notes
tlbsrx.	supervisor	Guest operating system supervisor code can receive search results without entry contents and set a TLB reservation before installing entries.
tlbivax	hypervisor	Only hypervisor code can invalidate global entries across processors or logical partitions.
tlbilx	supervisor	Guest operating system supervisor code can invalidate guest state entries. An attempt to invalidate a hypervisor state entry while MSR[GS] = 1 results in an embedded hypervisor privilege exception.

1. Because these instructions depend on data in the MAS registers, these instructions are executable only in MMU mode (CCR[NOTLB] = 0). Any attempt to execute one of these instructions while in ERAT-only mode (CCR2[NOTLB] = 1) results in an illegal instruction exception.

This processor is compliant with Category E.PT, and the size and location of the hardware page tables and format of the associated hardware page table entries are well defined. Aside from this, this processor can also maintain software-managed page tables, and this document does not imply any format for these software page tables or the associated page table entries. Software has significant flexibility in organizing the size, location, and format of software page tables, and in implementing a custom TLB entry replacement strategy. For example, software can lock TLB entries that correspond to frequently used storage, so that those entries are never cast out of the TLB, and TLB miss exceptions to those pages never occur.

Note: The descriptions in the following sections are based on 64-bit mode of operation. See *Section 6.11 32-Bit Mode Memory Management Behavior* on page 208 for a description of how these instructions behave differently for 32-bit mode operation.

6.9.1 TLB Read and Write Instructions (tlbre and tlbwe)

TLB entries can be read and written by the **tlbre** and **tlbwe** instructions, respectively, using the MMU Assist (MAS) Registers for entry data transfer. Certain implementation-specific fields in the TLB entries are transferred via the MMUCR3 register. (See *Section 6.18.4 Memory Management Unit Control Register 3 (MMUCR3)* on page 274 for a description of these fields). Because a TLB entry contains more than 64 bits, multiple **mtspr** and **mfspr** instructions must be executed to and from the MAS registers (and MMUCR3 register) to transfer all of the TLB entry information.

In previous embedded implementations, carefully planned software sequences and/or software locking was required when updating TLB entries because of the partial updates to the entries that occur when writing two or more parts of the entry. In the A2 design, the TLB includes four sets of MAS registers (one per thread) and four MMUCR3 registers (one per thread) that are updated upon **tlbre**. Both halves of a TLB entry (the effective and real portions) are updated atomically with the data most recently written to the MAS and MMUCR3 registers when **tlbwe** is executed.

When targeting the TLB (with MAS0[ATSEL] = 0), the **tlbre** instruction uses the MAS1[TID], MAS1[TSIZE], and MAS2[EPN] fields to define the hashed congruence class of the TLB entry. The ESEL field of MAS0 designates which way of the 4-way TLB array is to be read. Finally, the contents of the selected TLB entry are transferred to the appropriate MAS registers and MMUCR3.

When targeting the TLB (with MAS0[ATSEL] = 0), the **tlbwe** instruction uses the MAS1[TID], MAS1[TSIZE], and MAS2[EPN] fields to define the hashed congruence class of the TLB entry. The ESEL field of MAS0 can designate which way of the 4-way TLB array is to be written. Alternately, the hardware can automatically calculate the way location of the targeted TLB entry. This feature is controlled by the MAS0[HES] bit. When MAS0[HES] = 0, the ESEL field of MAS0 designates into which way of the 4-way TLB array is to be trans-

ferred. When MAS0[HES] = 1, the entry way is defined by the hardware LRU mechanism (which always excludes entries with IPROT = 1). Finally, the contents of the selected TLB entry are transferred from the appropriate MAS registers and MMUCR3 when the **tlbwe** completes.

The TLB 7-bit congruence class hash function is shown in *Table 6-8*. The table describes how each individual TLB index bit (that is, congruence class address bit) is formed by XORing different sets of EPN bits (and possibly PID bits) based on the page size. This function is used whenever an entry is being searched for (either due to servicing an ERAT miss, or for **tlbsx[.]**, a specific virtual address based invalidation, or when either the **tlbre** or **tlbwe** instructions are executed). For example, when searching for a 4 K page containing a nonzero TID value, TLB congruence class index bit 6 is determined by the following function: $TLB_C_C_INDEX(6) = EPN(51) \text{ XOR } EPN(44) \text{ XOR } EPN(37) \text{ XOR } PID(15)$.

Table 8. TLB Congruence Class Hashing Function (of EPN Address Bits)

TID1	TLB CC Index Bit	PID Bits ²	Page Size												
			4 KB			64 KB			1 MB ³		16 MB		256 MB ³		1 GB
TID ≠ 0 Entries	6	13	51	44	37	47		37	43	36	39		35		33
	5	12	50	43	36	46		36	42	35	38		34		32
	4	11	49	42	35	45		35	41	34	37		33		31
	3	10	48	41	34	44		34	40	33	36	32	32		30
	2	9	47	40	33	43	40	33	39	32	35	31	31		29
	1	8	46	39	32	42	39	32	38	31	34	30	30	28	28
	0	7	45	38	31	41	38	31	37	30	33	29	29	27	27
TID = 0 Entries	6	-	51	44	37	47		37	43	36	39		35		33
	5	-	50	43	36	46		36	42	35	38		34		32
	4	-	49	42	35	45		35	41	34	37		33		31
	3	-	48	41	34	44		34	40	33	36	32	32		30
	2	-	47	40	33	43	40	33	39	32	35	31	31		29
	1	-	46	39	32	42	39	32	38	31	34	30	30	28	28
	0	-	45	38	31	41	38	31	37	30	33	29	29	27	27

1. The entry's translation ID used to match PID can be either nonzero, in which case the PID is included in the hash, or can be zero, which is a wildcard match for any PID value, and therefore PID is not included in the hash.
2. For all page sizes, when the ENTRY.TID value is known or assumed to be nonzero, the 7 LSBs of the 14-bit PID value (bits 7:13 of PID bits 0:13) are XORed with the resulting EPN hash function to determine the final congruence class (CC).
3. Only these page sizes are valid for indirect (IND = 1) entries. All page sizes are valid for direct entries, except for 256 MB.

Writing TLB entries with **tlbwe** can also be made conditional (that is, dependent on the existence of a reservation held by the executing thread) by using the write qualifier field MAS0[WQ]. The reservation, which is tagged with a specific virtual address, can be established by the **tlbsrx.** instruction. The TLB write conditional form is enabled by setting MAS0[WQ] = 01. Alternately, the **tlbwe** can be made to ignore the reservation and always write (MAS0[WQ] = 00) or can simply clear the reservation without writing the TLB contents (MAS0[WQ] = 10). See *Section 6.15 TLB Reservations and TLB Write Conditional (Category E.TWC)* for more details.

Writing TLB entries with **tlbwe** is supervisory privileged and is executable by either the hypervisor or a guest operating system ($MSR[GS] = 1$). The guest's view of real addresses are actually termed "logical addresses" and must be converted to the actual system real addresses (that the hypervisor controls). This conversion is controlled by the LRAT facility (Category E.HV.LRAT). When a **tlbwe** is executed in guest mode, the fully associative LRAT entries are searched for a matching logical address, and, if found, the guest's logical address is converted to a real address before being written into the TLB. If a match is not found in the LRAT, an embedded hypervisor exception is raised. See *Section 6.8 Logical to Real Address Translation Array (Category E.HV.LRAT)* on page 193 for a detailed description of this facility.

When executing a **tlbwe**, the hardware calculates the parity to be recorded in the entry using the contents of the MAS registers and MMUCR3. If the parity bits stored for the particular entry that is read by the **tlbre** indicate a parity error, the parity error machine check exception is generated. See *Section 6.13.1 Parity Errors Generated from **tlbre** or **eratre*** for more information about the parity operation.

6.9.2 TLB Search Instruction (**tlbsx[.]**)

The **tlbsx[.]** instruction can be used to locate an entry in the TLB that is associated with a particular virtual address. This instruction forms an effective address for which the TLB is to be searched, in the same way data storage access instructions perform their address calculation, by adding the contents of registers RA (or the value 0 if $RA = 0$) and RB together. The MAS5[SGS], MAS5[SLPID], MAS6[SAS], and MAS6[SPID] fields then provide the guest state, logical partition ID, address space, and process ID portions of the virtual address, respectively. The MAS6[SIND] is also used to differentiate between direct versus indirect TLB entries. Next, the TLB is searched for this virtual address; the search process disables the comparison to the process ID if the TID field of a given TLB entry is 0 (see *Section 6.2.4 TLB Match Process* on page 173). Finally, the EPN, TID, TSIZE, and TLB way index of the matching entry are written into the MAS2[EPN], MAS1[TID], MAS1[TSIZE], and MAS0[ESEL] fields respectively. These values can then serve as the source values to calculate the congruence class index hash for a subsequent **tlbre** or **tlbwe** instruction, to read or update the entry. If no matching entry is found, the target MAS register contents are set to default values (see *Section 6.17.28 MAS Register Update Summary* on page 259).

The "record form" of the instruction (**tlbsx.**) updates $CR[CR0]_2$ with the result of the search. If a match is found, $CR[CR0]_2$ is set to 1; otherwise, it is set to 0.

When the TLB is searched using a **tlbsx** instruction, if a matching entry is found, the parity calculated for the tag (EPN portion) is compared to the parity stored in the entry. A mismatch causes a parity error exception. Parity errors in word 1 (the RPN portion) of the entry do not cause parity error exceptions when executing a **tlbsx** instruction.

6.9.3 TLB Search and Reserve Instruction (**tlbsrx.**)

The **tlbsrx.** instruction can be used to search for the existence of an entry in the TLB that is associated with a particular virtual address. This instruction forms a virtual address for which the TLB is to be searched, in the same manner as the **tlbsx[.]** instruction. However, the **tlbsrx.** instruction does not return the entry contents to the MAS registers, but rather sets a reservation associated with the virtual address that can be used to qualify (enable) subsequent **tlbwe** instructions. This TLB write conditional property of the **tlbwe** instruction is controlled by the MAS0[WQ] field. See *Section 6.15 TLB Reservations and TLB Write Conditional (Category E.TWC)* for more details.

Finally, an indication is provided in $CR[CR0]_2$ as to the search results so that software can decide how to proceed if a duplicate entry already exists for the virtual address associated with the search. This can occur due to multiple threads racing to install the same TLB entry for a similar page miss.

6.9.4 TLB Invalidate Virtual Address (Indexed) Instruction (tlbivax)

The **tlbivax** instruction is used to invalidate TLB and ERAT entries that contain the virtual page number associated with the effective address of this instruction. This is a global version of invalidation that affects all processors containing TLB entries tagged with the same logical partition ID. The MAS6[ISIZE] field contains the invalidation page size. The MAS5[SGS] and MAS5[SLPID] fields are used as the invalidation guest state and target logical partition ID. The MAS6[SPID] and MAS6[SAS] fields are used as the invalidation process ID and address space. The MAS6[SIND] is also used to differentiate between direct versus indirect TLB entries.

The IPROT value (and ExtClass value for the ERATs) of the **tlbivax** instruction is always assumed to be "0". TLB entries that have been established with IPROT = 1 (and ERAT entries with ExtClass = 1) are not invalidated by the **tlbivax** instruction. This allows privileged software running on the local core to establish an additional level of masking or "immunization" against invalidations for a unique set of entries.

Programming Note: Only one processing thread per core can issue a **tlbivax** and/or a **tlbsync** at a time. Failure to observe these limitations might result in an unrecoverable system hang. This is usually accomplished via software locking.

Engineering Note: It is assumed that two or more A2 processor cores (including the local core) can issue simultaneous **tlbivax** operations targeting the same or different logical partitions. It is a requirement of this processor, however, that the memory subsystem that provides the back invalidate snoops for these transactions must serialize the invalidate snoops such that the A2 core receives only one snoop at a time (that is, until the required core-sourced handshaking operation is sent for the current snoop operation). It is also a requirement that the memory subsystem provides a locally sourced versus remotely sourced indication to the core as part of the invalidation snoop transaction. This is necessary for the local core to differentiate between a snoop that is the result of a **tlbivax** instruction issued by a thread on this core (that is stalled at issue and waiting for a locally sourced snoop to complete) and one that originated from a remote core (which does not release a locally stalled thread).

These restrictions are not assumed for the local version **tlbilx** instruction. Because a local **tlbilx** instruction does not go to the bus, it does not formally require software locking. Therefore, the MMU needs to know the difference between a locally originated invalidate operation and a nonlocally generated invalidate.

All TLB entries are tagged with the logical partition ID (LPID), and the **tlbivax** invalidation snoops from the bus contain a target LPID value. The handling of the invalidation snoops based on this LPID value is dependent on the configured mode of the receiving core. Although a heterogeneous MMU mode system is not envisioned (that is, one in which some cores are configured for MMU mode, while others are configured as ERAT-only mode), it might be possible for some system configurations and is therefore described.

When this core is operating in MMU mode (CCR2[NOTLB] = 0), the MMUCR1[TLBI_REJ] bit controls the behavior of the snoop handling hardware based on the LPID value. If MMUCR1[TLBI_REJ] = '0', the MMU accepts all incoming invalidation snoop operations regardless of the current LPIDR register contents, and it includes the incoming LPID snoop value in its TLB entry invalidation match criteria. There is no rejection of the transaction by the MMU in this case, and a TLBI_COMPLETE is issued to the memory subsystem after the TLB and ERAT copies have been invalidated. Conversely, when MMUCR1[TLBI_REJ] = '1' and an incoming invalidation snoop operation is targeted for a different partition from that contained in the local LPIDR register, the MMU issues an immediate rejection of the transaction to the memory subsystem, and no TLBI_COMPLETE is issued.

When this core is operating in ERAT-only mode (CCR2[NOTLB] = 1) and an incoming invalidation snoop operation is targeted for a different partition from that contained in the local LPIDR register, the MMU issues an immediate rejection of the transaction to the memory subsystem and no TLBI_COMPLETE is issued. If the

incoming invalidation snoop LPID value matches the current value of the LPIDR[LPID] field, there is no rejection of the transaction by the MMU; and a TLBI_COMPLETE is issued to the memory subsystem as after the appropriate ERAT entries have been invalidated.

For this description, the term “AS” refers to the current addressing space as determined by MSR[IS] for instruction fetches and by MSR[DS] for data accesses. The term “GS” refers to the current guest state determined by MSR[GS], and LPID is the value of the LPIDR register. This implementation supports an 88-bit virtual address (1-bit GS || 8-bit LPID || 1-bit AS || 14-bit PID || 64-bit EA). For the **tlbivax** instruction, the MAS5[SGS], MAS5[SLPID], MAS6[SAS], and MAS6[SPID] register fields are concatenated with bits [0:(63-p), where $p = \log_2(\text{page size})$] of the effective address (EA) and are used to form the VPN to match. This results in a selective invalidation in the TLB and (and shadow copies in the ERATs) based on the VPN, the page size, and potentially other attributes. In other words, any entry in the TLB with matching TGS, TLPID, TS, TID, and effective page number (EPN) is invalidated.

When global **tlbivax** operations are sent over certain system bus structures (such as the PBus), some of the information associated with the invalidate transaction needs to be condensed to conform to the bus width. This processor condenses EPN[27:51], the TS and TID, and the page size into a single 42-bit physical address bus ($w = 27$, the MSb of the EPN encoding on the A2 core downbound request address bus). The TGS, IND, and L parameters, along with the targeted LPID value, are sent in the data payload as part of the downbound invalidation request from the core. This can lead to aliasing when using global invalidations. The ability to transfer a complete virtual address and other information for global **tlbivax** transactions is dependent on the manner in which a particular system bus encodes this information and the number of EPN bits necessary to support a given application and/or logical partition. The PBus, for example, uses this information for transmission to remote processors as a TLBI_OP bus transaction using a 46-bit physical address bus that includes the LPID value.

The GS, IND, and L parameters from the **tlbivax** instruction are extracted from the downbound request data payload sent from the core to the memory subsystem and transported in the secondary type field of the PBus TLBI_OP transaction. When the page size is greater than 4 KB ($L = 1$), the page size is placed in the least significant 4 bits of the EPN field being sent on the address bus to the system (EPN[48:51]). The EA[27:30] bits are placed on unused EPN[44:47] bits on the address bus to the system for certain larger page sizes (used in the hardware hashing function to determine the targeted TLB’s congruence class).

The target TLB and ERAT effective address comparison ($\text{entry}[\text{EPN}_{w:63-p}] = ? \text{EPN}_{w:63-p}$, where $p = \log_2(\text{page size})$) is subject to the limitations imposed by the physical core to memory subsystem address bus width (and potentially other physical system address bus bottlenecks). The A2 core supports two configurations options for determination of the MSB of the **tlbivax** bus transaction EPN field (based on the difference between the limited EPN width supported by the PBus transaction and other possible bus structures that could support the full A2 core request bus interface EPN width). MMUCR1[TLBI_MSB] controls this selection.

The EPN encoding of the A2 core request bus interface supports a variable value for “w” based on supported page sizes and MMUCR1[TLBI_MSB]. This value of w is used directly for the EPN comparison for TLB entries. For the ERAT shadow copies, a constant value of $w = 31$ is used that results in potentially more generous invalidations than those of the TLB depending on page size.

For the downbound core generated **tlbivax** request to the memory subsystem, certain large page sizes contain unused LSBs of the EPN field. These unused bits are overlaid with more significant EPN bits in certain cases. The downbound request EPN field encoding is implemented as shown in *Table 6-9*.

Table 9. Supported EPN[27:51] Field Values in Downbound TLBIVAX Request

Page Size	MMUCR1 [TLBI_MSB]	EPN[27:30] ²	EPN[31:33]	EPN[34:35]	EPN[36:39]	EPN[40:43]	EPN[44:47]	EPN[48:51]	TLB "w" Value ³
4 KB (L = 0)	0	EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[40:43]	EA[44:47]	EA[48:51]	31
64 KB (L = 1)		EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[40:43]	EA[44:47]	0b0011	31
1 MB (L = 1)		EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[40:43]	EA[27:30]	0b0101	27
16 MB (L = 1)		EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[23:26]	EA[27:30]	0b0111	23
256 MB ¹ (L = 1)		EA[27:30]	EA[31:33]	EA[34:35]	EA[19:22]	EA[23:26]	EA[27:30]	0b1001	19
1 GB (L = 1)		EA[27:30]	EA[31:33]	EA[17:18]	EA[19:22]	EA[23:26]	EA[27:30]	0b1010	17
4 KB (L = 0)		1	EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[40:43]	EA[44:47]	EA[48:51]
64 KB (L = 1)	EA[27:30]		EA[31:33]	EA[34:35]	EA[36:39]	EA[40:43]	EA[44:47]	0b0011	27
1 MB (L = 1)	EA[27:30]		EA[31:33]	EA[34:35]	EA[36:39]	EA[40:43]	EA[23:26]	0b0101	23
16 MB (L = 1)	EA[27:30]		EA[31:33]	EA[34:35]	EA[36:39]	EA[19:22]	EA[23:26]	0b0111	19
256 MB ¹ (L = 1)	EA[27:30]		EA[31:33]	EA[34:35]	EA[15:18]	EA[19:22]	EA[23:26]	0b1001	15
1 GB (L = 1)	EA[27:30]		EA[31:33]	EA[13:14]	EA[15:18]	EA[19:22]	EA[23:26]	0b1010	13

1. This page size is supported only for IND = 1 invalidations.

2. These EPN bits are not supported in the PBus TLBIVAX_OP address definition (Category B.E.TWC), but might be supported by other bus structures.

3. The "w" value for ERAT shadow copies in this implementation is always 31 (that is, ERAT arrays do not benefit from additional EA bits being transported and are subject to more generous EPN aliasing).

6.9.5 TLB Invalidate Local (Indexed) Instruction (tlbilx)

The **tlbilx** instruction is used to invalidate TLB and ERAT entries that contain the virtual page EPN number associated with the effective address of this instruction. This is a local version of invalidation that affects only the local processor's TLB and ERAT entries. The "T" parameter of this instruction dictates how selective (or specific) the invalidation is to be. The MAS6[ISIZE] field contains the invalidation page size. The MAS6[SPID] and MAS6[SAS] fields are used as the invalidation process ID and address space. The MAS6[SIND] is used to differentiate between direct versus indirect TLB entries for T = 1 and T = 3 invalidations. MAS6[SIND] is used as a literal IND bit match value for T = 3 (invalidate by VA), or alternately it is used as the IND bit match enable (when set to '1') with the match value being '0' for T = 1 (invalidate by PID). This second usage allows for invalidation of only direct entries when invalidating based on process ID.

The IPROT value (and ExtClass value for the ERATs) of the **tlbilx** instruction is always assumed to be "0". TLB entries that have been established with IPROT = 1 (and ERAT entries with ExtClass = 1) are not invalidated by the **tlbilx** instruction. This allows privileged software running on the local core to establish an additional level of masking, or "immunization" against invalidations for a unique set of entries. This includes requests to invalidate all entries or all entries of a certain process ID or class.

6.9.6 TLB Sync Instruction (tlbsync)

The **tlbsync** instruction is used to synchronize software TLB management operations in a multiprocessor environment with hardware-enforced coherency. It is provided in support of software compatibility between Power ISA-based systems.

The **tlbsync** instruction can be used (in memory subsystems that support this behavior) to ensure that the effects of global **tlbivax** and **erativax** operations have been made globally visible. Generally, this behavior depends on the processor waiting for the memory subsystem to deliver a **sync** acknowledgment after the **tlbsync** has been completed on the bus fabric. In A2, this behavior is controlled by a bit in the XUCR0 register.

6.10 ERAT Management Instructions (Non-Architected)

To enable hypervisor (or “bare-metal” operating system) software to manipulate the ERAT entries directly, a set of nonarchitected ERAT management instructions is implemented within the A2 core. These instructions are described briefly in the sections that follow and in detail in *Section 12 Implementation Dependent Instructions* on page 475. All of the ERAT management instructions are hypervisor privileged to prevent user and guest mode programs from affecting the shadow TLB address translation and access control mechanisms.

The processor does not imply any format for the page tables or the page table entries. Software has significant flexibility in organizing the size, location, and format of the page table, and in implementing a custom ERAT entry replacement strategy. For example, software can lock certain ERAT entries that correspond to frequently used storage so that those entries are never cast out of the ERATs and TLB miss exceptions to those pages never occur.

Note: The descriptions in the following sections are based on 64-bit mode of operation. See *Section 6.11 32-Bit Mode Memory Management Behavior* on page 208 for a description of how these instructions behave differently for 32-bit mode operation.

All of the ERAT management instructions are embedded hypervisor privileged to prevent user mode programs and guest operating system code from directly affecting the ERAT shadow copies. *Table 6-10* shows the privilege levels of the various ERAT management instructions.

Table 10. ERAT Management Instruction Privilege Levels

Instruction	Privilege ¹	Notes
eratre	hypervisor	ERAT entry real addresses that can be returned are hypervisor privileged data.
eratwe	hypervisor	Installing ERAT entries is not conditioned by the LRAT facility without hypervisor intervention.
eratsx[.]	hypervisor	ERAT entry real address contents that can be returned in the event of a search “hit” are hypervisor privileged data.
erativax	hypervisor ²	Only hypervisor code can invalidate global entries across processors or logical partitions.
eratilx	hypervisor	Hypervisor code can invalidate guest state ERAT entries when re-assigning logical partitions.

1. Because these instructions do not depend on data in the MAS registers, these instructions (with the exception of **erativax**) can be executed in either MMU mode (CCR[NOTLB] = 0) or ERAT-only mode (CCR[NOTLB] = 1).
 2. Any attempt to execute the **erativax** instruction while in MMU mode (CCR2[NOTLB] = 0) results in an illegal instruction exception.

6.10.1 ERAT Read and Write Instructions (eratre and eratwe)

ERAT entries can be read and written by the **eratre** and **eratwe** instructions, respectively. Because an ERAT entry contains more than 64 bits, multiple **eratre/eratwe** instructions must be executed to transfer all of the ERAT entry information. An ERAT entry is divided into two portions: ERAT word 0 and ERAT word 1. The RA field of the **eratre** and **eratwe** instructions designates a GPR from which the low-order bits are used to specify the index of the ERAT entry to be read or written. An immediate field (WS) designates which word of the

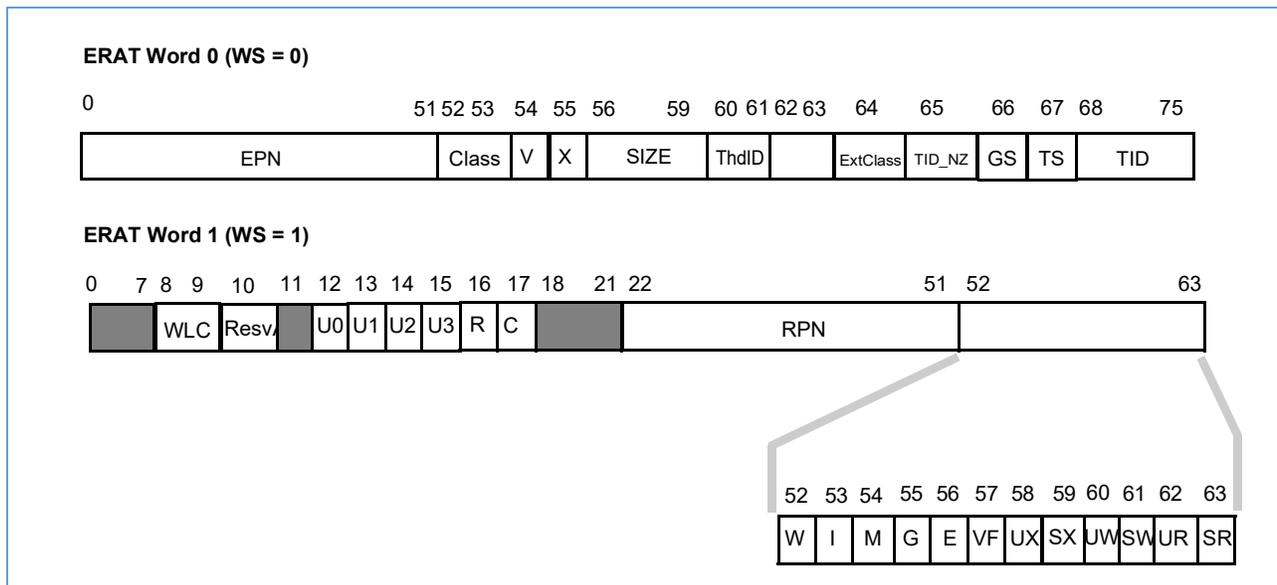
ERAT entry is to be transferred (that is, $WS = 0$ specifies ERAT word 0, and so on). Finally, the contents of the selected ERAT word are transferred to or from a designated target or source GPR (and the MMUCR0 GS, TS, TID, ExtClass, and TID_NZ fields, for TLB word 0; see *Figure 6-3*), respectively.

In previous embedded implementations, carefully planned software sequences and/or software locking was required when updating entries because of the partial updates to the entries that occur when writing two or more parts of the entry. In the A2 design, each of the ERAT caches includes four (one per thread) 64-bit RPN registers that are updated upon **eratwe** of the RPN portion ($WS = 1$). Both halves of the ERAT entry are then updated atomically when **eratwe** is executed with $WS = 0$ (EPN portion). The value written into the RPN portion of the entry is the data most recently written to the RPN holding register.

The fields in each ERAT word are illustrated in *Figure 6-3*. The bit numbers indicate which bits of the target or source GPR correspond to each ERAT field. Note that the GS, TS, TID, ExtClass, and TID_NZ fields of ERAT word 0 are transferred to or from the MMUCR0[TGS], [TS], [TID], [ECL], and [TID_NZ] fields respectively, rather than to or from the target or source GPR.

When executing an **eratwe**, the hardware calculates the parity to be recorded in the entry. If the parity bits stored for the particular word that is read by the **eratre** indicate a parity error, the parity error machine check exception is generated. See *Section 6.13.1 Parity Errors Generated from tlbre or eratre* on page 214 for more information about parity operation.

Figure 3. ERAT Entry Word Definitions



6.10.2 ERAT Search Instruction (eratsx[.])

The **eratsx[.]** instruction can be used to locate an entry in the I-ERAT or D-ERAT that is associated with a particular virtual address. This instruction forms an effective address for which the selected ERAT is to be searched, in the same way data storage access instructions perform their address calculation, by adding the contents of registers RA (or the value 0 if RA = 0) and RB together. The MMUCR0[TGS], MMUCR0[TS], and MMUCR0[TID] fields then provide the guest state, address space, and process ID portions of the virtual address, respectively (the logical partition ID portion is not contained in the ERATs in the A2 implementation). Next, the ERAT is searched for this virtual address; the search process disables the comparison to the process ID if the TID_NZ field of a given ERAT entry is 0 (see *Section 6.2.4 TLB Match Process* on

page 173). Finally, the index of the matching entry is written into the target register (RT). This index value can then serve as the source value for a subsequent **eratre** or **eratwe** instruction, to read or update the entry. If no matching entry is found, the target register contents are undefined.

Note: Only the RA = 0 variation of this instruction is supported for searching the I-ERAT array (that is, EA is calculated as RA = 0 + (RB) when MMUCR0[TLBSEL] = 2).

The “record form” of the instruction (**eratsx.**) updates CR[CR0]₂ with the result of the search: if a match is found, CR[CR0]₂ is set to 1; otherwise it is set to 0.

When the ERAT is searched using an **eratsx[.]** instruction, if a matching entry is found, the parity calculated for the tag (EPN portion) is compared to the parity stored in the entry. A mismatch causes a parity error exception. Parity errors in word 1 (RPN portion) of the entry will not cause parity error exceptions when executing an **eratsx[.]** instruction (that is, the search operation only relies on the integrity of the EPN portion).

6.10.3 ERAT Invalidate Virtual Address (Indexed) Instruction (**erativax**)

The **erativax** instruction is used to invalidate ERAT entries that contain the virtual page number associated with the effective address of this instruction. The **erativax** instruction can be executed in “ERAT-only” mode (CCR2[NOTLB] = ‘1’). Execution of this instruction in MMU mode results in an illegal instruction exception. This is a global version of invalidation that affects all processors in the same logical partition. The RS[ISIZE] field contains the invalidation page size. The MMUCR0[TGS] and LPIDR[LPID] fields are used as the invalidation guest state and logical partition ID. The MMUCR0[TID] and MMUCR0[TS] fields are used as the invalidation process ID and address space.

The extended class (ExtClass) of the **erativax** instruction is always assumed to be “0”. Entries that have been established with ExtClass = 1 are not invalidated by this instruction. This allows privileged software running on the local core to establish an additional level of masking, or “immunization” against invalidations for a unique set of entries. This includes requests to invalidate all entries or all entries of a certain process ID or class.

This implementation requires that only *one processor at a time can issue an erativax that targets a specific logical partition* (that is, a unique LPIDR[LPID] value) and that only *one processing thread per core* can issue an **erativax** at a time. Failure to observe these limitations might result in an unrecoverable system hang. This is usually accomplished via software locking. Therefore, it is not possible to have a locally originated **erativax** and an **erativax** from the bus for the same logical partition simultaneously. It is assumed that two or more A2 processor cores (including the local core) can issue simultaneous **erativax** operations targeting different logical partitions. It is a requirement of this processor, however, that the memory subsystem that provides the back invalidate snoops for these transactions *must serialize the invalidate snoops such that the A2 core receives only one snoop at a time* (that is, until the required core-sourced handshaking operation is sent for the current snoop operation). It is also a requirement that the memory subsystem provides a *locally sourced versus remotely sourced indication* to the core as part of the invalidation snoop transaction. This is necessary for the local core to differentiate between an invalidation snoop that is the result of an instruction issued by a thread on this core (that can be stalled at issue) or that originated from a remote core. Because the ERAT entries are not tagged with the translation logical partition ID (TLPID), all entries are assumed to reside in the same logical partition at any one time. If an incoming **erativax** operation is targeted for a different logical partition (as determined by the LPIDR register), the MMU rejects it back to the bus. These restrictions are not assumed for the local version **eratilx** instruction. Because a local **eratilx** instruction does not go to the bus, it does not formally require software locking. Therefore, the MMU needs to know the difference between a locally originated invalidate operation and a nonlocally generated invalidate.

The **erativax** invalidation snoops from the bus contain a target LPID value. The handling of the invalidation snoops based on this LPID value is dependent on the configured mode of the receiving core. While a heterogeneous MMU mode system is not envisioned (that is, one in which some cores are configured for MMU mode, while others are configured as ERAT-only mode), it might be possible for some system configurations and is therefore described.

When the receiving core is operating in MMU mode (CCR2[NOTLB] = 0), the MMUCR1[TLBI_REJ] bit controls the behavior of the snoop handling hardware based on the LPID value. If MMUCR1[TLBI_REJ] = '0', the MMU accepts all incoming invalidation snoop operations, regardless of the current LPIDR register contents, and includes the incoming LPID snoop value in its TLB entry invalidation match criteria. There is no rejection of the transaction by the MMU in this case, and a TLBI_COMPLETE is issued to the memory subsystem after the TLB and ERAT copies have been invalidated. Conversely, when MMUCR1[TLBI_REJ] = '1' and an incoming invalidation snoop operation is targeted for a different partition from that contained in the local LPIDR register, the MMU issues an immediate rejection of the transaction to the memory subsystem, and no TLBI_COMPLETE is issued.

When the receiving core is operating in ERAT-only mode (CCR2[NOTLB] = 1) and an incoming invalidation snoop operation is targeted for a different partition from that contained in the local LPIDR register, the MMU issues an immediate rejection of the transaction to the memory subsystem, and no TLBI_COMPLETE is issued. If the incoming invalidation snoop LPID value matches the current value of the LPIDR[LPID] field, there is no rejection of the transaction by the MMU, and a TLBI_COMPLETE is issued to the memory subsystem after the appropriate ERAT entries have been invalidated.

For this discussion, the term "AS" refers to the current addressing space as determined by MSR[IS] for instruction fetches and by MSR[DS] for data accesses. This implementation supports a 88-bit virtual address (1-bit GS || 8-bit LPID || 1-bit AS || 14-bit PID || 64-bit EA). For the **erativax** instruction, the MMUCR0[TGS], [TS], [TID], and the LPIDR[LPID] register fields (TGS || LPID || TS || TID) are concatenated with bits [0:(63-p)] of the EA, where $p = \log_2(\text{page size})$, and are used to form the VPN to match. In other words, any nonprotected entry in an ERAT in the same logical partition with matching TGS, TS, TID, effective page number (EPN), and page size, or conditionally matching only page class, or conditionally matching only page TID, or conditionally all non-protected entries is invalidated. The IS (invalidation select) field controls this, and is provided in RS[56:57] of the **erativax** instruction. *Table 6-11* gives details of the implementation of the IS field.

Table 11. Summary of Supported IS Field Values in ERATIVAX

MMU Mode	IS Field	Local/Global	Behavior
ERAT-only	00	Global	INVAL_ALL All nonprotected entries associated with the logical partition "lpid" are invalidated.
ERAT-only	01	Global	INVAL_TID All nonprotected entries associated with logical partition "lpid" that match TID are invalidated.
ERAT-only	10	Global	INVAL_CLASS All nonprotected entries associated with logical partition "lpid" that match CLASS are invalidated.
ERAT-only	11	Global	The logic is as selective as possible when invalidating nonprotected entries associated with logical partition "lpid". The invalidation match criteria is EPN[31:(63-p)], TGS, TS, TID, and SIZE.
TLB mode	-	-	Not supported; illegal instruction exception

When global **erativax** operations are sent over certain system bus structures (such as the PBus), some of the information associated with the invalidate transaction needs to be condensed to conform to the bus width. This processor condenses EPN[27:51], the TS and TID, and the page size into a single 42-bit physical address bus ($w = 27$, the MSb of the EPN encoding on the A2 core downbound request address bus). The TGS, IND, and L parameters, along with the targeted LPID value, are sent in the data payload as part of the downbound invalidation request from the core. This can lead to aliasing when using global invalidations. The ability to transfer complete virtual address and other information for global **erativax** transactions is dependent on the manner in which a particular system bus encodes this information and the number of EPN bits necessary to support a given application and/or logical partition. The PBus, for example, uses this information for transmission to remote processors as a TLBI_OP bus transaction using a 46-bit physical address bus that includes the LPID value.

The GS, IND, and L parameters from the **erativax** instruction (of course, $IND = 0$ always for this instruction) are extracted from the downbound request data payload sent from the core to the memory subsystem and transported in the secondary type field of the PBus TLBI_OP transaction. When the page size is greater than 4 KB ($L = 1$), the page size is placed in the least significant 4 bits of the EPN field being sent on the address bus to the system (EPN[48:51]). The EA[27:30] bits are placed on unused EPN[44:47] bits on the address bus to the system for certain larger page sizes (to be used to complete the hardware hashing function to determine the targeted TLBs congruence class).

The target ERAT effective address comparison ($entry[EPN_{w:63-p}] = ? EPN_{w:63-p}$, where $p = \log_2(\text{page size})$) is subject to the limitations imposed by the physical core to memory subsystem address bus width (and potentially other physical system address bus bottlenecks). The A2 core supports two configurations options for determination of the MSB of the **erativax** bus transaction EPN field (based on the difference between the limited EPN width supported by the PBus transaction and other possible bus structures that could support the full A2 core request bus interface EPN width). MMUCR1[TLBI_MSB] controls this selection.

The EPN encoding of the A2 core request bus interface supports a variable value for “w” based on supported page sizes and MMUCR1[TLBI_MSB]. This value of w is used directly for the EPN comparison for TLB entries (for example, in MMU mode **tlbivax** instructions). For the ERAT shadow copies, a constant value of $w = 31$ is used, which results in potentially more generous invalidations than those of the TLB-based hardware configurations depending on page size. The constant value $w = 31$ is a limitation imposed by the ERAT array hardware in this implementation.

For the downbound core generated **erativax** request to the memory subsystem, certain large page sizes contain unused LSBs of the EPN field. These unused bits are overlaid with more significant EPN bits in certain cases. The downbound request EPN field encoding is implemented as shown in *Table 6-12*.

Table 12. Supported EPN[27:51] Field Values in Downbound *erativax* Request

Page Size	MMUCR1 [TLBI_MSB]	EPN[27:30] ¹	EPN[31:33]	EPN[34:35]	EPN[36:39]	EPN[40:43] ³	EPN[44:47]	EPN[48:51]	ERAT "w" Value ²
4 KB (L = 0)	0	EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[40:43]	EA[44:47]	EA[48:51]	31
64 KB (L = 1)		EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[40:43]	EA[44:47]	0b0011	31
1 MB (L = 1)		EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[40:43]	EA[27:30]	0b0101	31
16 MB (L = 1)		EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[23:26]	EA[27:30]	0b0111	31
1 GB (L = 1)		EA[27:30]	EA[31:33]	EA[17:18]	EA[19:22]	EA[23:26]	EA[27:30]	0b1010	31
4 KB (L = 0)	1	EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[40:43]	EA[44:47]	EA[48:51]	31
64 KB (L = 1)		EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[40:43]	EA[44:47]	0b0011	31
1 MB (L = 1)		EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[40:43]	EA[23:26]	0b0101	31
16 MB (L = 1)		EA[27:30]	EA[31:33]	EA[34:35]	EA[36:39]	EA[19:22]	EA[23:26]	0b0111	31
1 GB (L = 1)		EA[27:30]	EA[31:33]	EA[13:14]	EA[15:18]	EA[19:22]	EA[23:26]	0b1010	31

1. These EPN bits are not supported in the PBus TLBIVAX_OP address definition (Category B.E.TWC), but might be supported by other bus structures.
2. The "w" value for ERAT shadow copies in this implementation is always 31 (that is, ERAT arrays do not benefit from additional EA bits being transported and are subject to more generous EPN aliasing).
3. For IS = '10' (invalidate by class), EPN bits 42:43 contain the class value to be targeted, which is derived from RS(58:59).

6.10.4 ERAT Invalidate Local (Indexed) Instruction (*eratilx*)

The *eratilx* instruction is used to invalidate local ERAT entries that contain the virtual page number associated with the effective address of this instruction, or alternately, that contain certain specific values of parameters such as process ID, class, and so forth. This instruction can be executed in either ERAT-only mode or MMU mode; it has no effect on the underlying TLB structure (if it exists in a particular MMU implementation). The RS source register contains the page size and class of the invalidation, along with an invalidation select (IS) field that determines how specific (or selective) the invalidation transaction is to be.

The extended class (ExtClass) of the *eratilx* instruction is always assumed to be "0". Entries that have been established with ExtClass = 1 are not invalidated by this instruction. This allows privileged software running on the local core to establish an additional level of masking, or "immunization" against invalidations for a unique set of entries. This includes requests to invalidate all entries, or all entries of a certain class or process ID.

This instruction is not globally broadcast to other processors in the system.

6.11 32-Bit Mode Memory Management Behavior

When this processor's machine state is changed to operate in 32-bit mode, the virtual address translation mechanism operates essentially the same as described elsewhere in this document for 64-bit mode, but there are nuances that users need to be aware of. The effective addresses for data loads and stores that are seen by the D-ERAT, and the instruction fetch addresses as seen by the I-ERAT, are modified such that the upper 32-bits of the effective address are zeroed before being compared in the ERAT logic. This means that entries installed for 32-bit processes either directly by management instructions in ERAT-only mode, or by

TLB hit reloads in MMU mode, need to have the upper 32 bits of the effective page number zeroed at the time of installation to have effective addresses compare successfully on these entries. This is done automatically in hardware to the source operand when using the **tlbwe** instruction in 32-bit mode.

The upper 32 bits of the A2 processor's 64-bit GPR hardware structures are undefined in 32-bit mode (that is, the upper 32 bits can contain undefined data left over from a 64-bit to 32-bit state transition). Because the TLB management instructions rely on GPRs as source and target registers, these instructions operate somewhat differently in 32-bit mode. These differences are described below.

6.11.1 32-Bit Mode TLB Read and Write Instructions (**tlbre** and **tlbwe**)

The TLB entries are read from and written to using the **tlbre** and **tlbwe** instructions, respectively. The MMU Assist Registers (MAS) are used as destination or source for these instructions. However, in 32-bit mode operation, the upper 32 bits of EPN(0:31) are treated as zeros. This means that the TLB entry EPN(0:31) bits are set to zeros when **tlbwe** is executed in 32-bit mode. Likewise, the upper 32 bits of MAS2[EPN] are set to zeros when **tlbre** is executed in 32-bit mode. Because the real page number is broken up into disjoint 32-bit registers (MAS3[RPNL] and MAS7[RPNU]), the full real page number (RPN) is transferred between the TLB entry and the appropriate MAS registers.

Note: The A2 hardware structures that embody the TLB and ERATs physically contain 52 bits that represent the EPN in 64-bit mode. In 32-bit mode, the upper 32 bits of the EPN (EPN[0:31]) are zeroed in the source operand of the **tlbwe** instruction (source data from MAS2[EPN] in this case). This is not the case when creating entries via the **tlbwe** instruction in 64-bit mode. When 64-bit mode supervisory software creates entries intended to be used in the 32-bit machine state, it must create such entries with the upper 32-bits of the EPN zeroed for 32-bit mode translation compares to succeed.

6.11.2 32-Bit Mode TLB Search Instruction (**tlbsx[.]**)

The **tlbsx[.]** instruction is used to locate an entry in the TLB that is associated with a particular virtual address. This instruction forms an effective address for which the TLB is to be searched, in the same way data storage access instructions perform their address calculation, by adding the contents of registers RA (or the value 0 if RA = 0) and RB together. In 32-bit mode, this instruction operates essentially the same as the 64-bit mode version (that is, bits 0 to 51 of the EPN are still compared to the contents of the ERAT or TLB entry's EPN, assuming a 4 KB page size). However, in 32-bit mode, address bits 0-31 of the effective address are forced to zero before comparison. This implies that the effective page number (EPN) in the TLB entries that pertain to the current 32-bit process need to have been created with zeros in the upper 32-bits for a search EPN compare to succeed.

6.11.3 32-Bit Mode TLB Search and Reserve Instruction (**tlbsrx.**)

The **tlbsrx.** instruction can be used to search for the existence of an entry in the TLB that is associated with a particular virtual address. This instruction operates essentially the same as the 64-bit mode version. However, in 32-bit mode, address bits 0-31 of the effective address are forced to zero before comparison. This implies that the effective page number (EPN) in the TLB entries that pertain to the current 32-bit process needs to have been created with zeros in the upper 32-bits for a search compare to succeed and a reservation to be established. Assuming a reservation is successfully established, the upper 32-bits of the reservation EPN are set to zeros. Any subsequent **tlbwe** conditional (with MAS0[WQ] = '01') succeeds only if MAS2[EPN] bits 0:31 appear to be zero as well (either set to zeros via a 64-bit mode **mtspr**, or the **tlbwe** conditional is also executed in 32-bit mode).

6.11.4 32-Bit Mode TLB Invalidate Virtual Address (Indexed) Instruction (tlbivax)

The **tlbivax** instruction is used to invalidate TLB (and ERAT) entries that contain the virtual page number associated with the effective address of this instruction. This instruction operates essentially the same as the 64-bit mode version. However, in 32-bit mode, address bits 0:31 of the effective address are forced to zero before forwarding the invalidation snoops to target processors. This implies that the effective page number (EPN) in the TLB (and ERAT) entries that pertain to the current 32-bit process need to have been created with zeros in the upper 32-bits for an invalidate EPN compare to succeed and invalidation to occur.

6.11.5 32-Bit Mode TLB Invalidate Local (Indexed) Instruction (tlbilx)

The **tlbilx** instruction is used to invalidate local TLB (and ERAT) entries that contain the virtual page number associated with the effective address of this instruction or, alternately, that contain certain specific values of parameters such as process ID.

This instruction operates essentially the same as the 64-bit mode version. However, in 32-bit mode, address bits 0:31 of the effective address are forced to zero before comparison. This implies that the effective page number (EPN) in the TLB (and ERAT) entries that pertain to the current 32-bit process need to have been created with zeros in the upper 32-bits for an invalidate EPN compare to succeed and invalidation to occur.

6.11.6 32-Bit Mode TLB Sync Instruction (tlbsync)

The 32-bit mode **tlbsync** instruction behavior is identical to 64-bit mode.

6.11.7 32-Bit Mode ERAT Read and Write Instructions (eratre and eratwe)

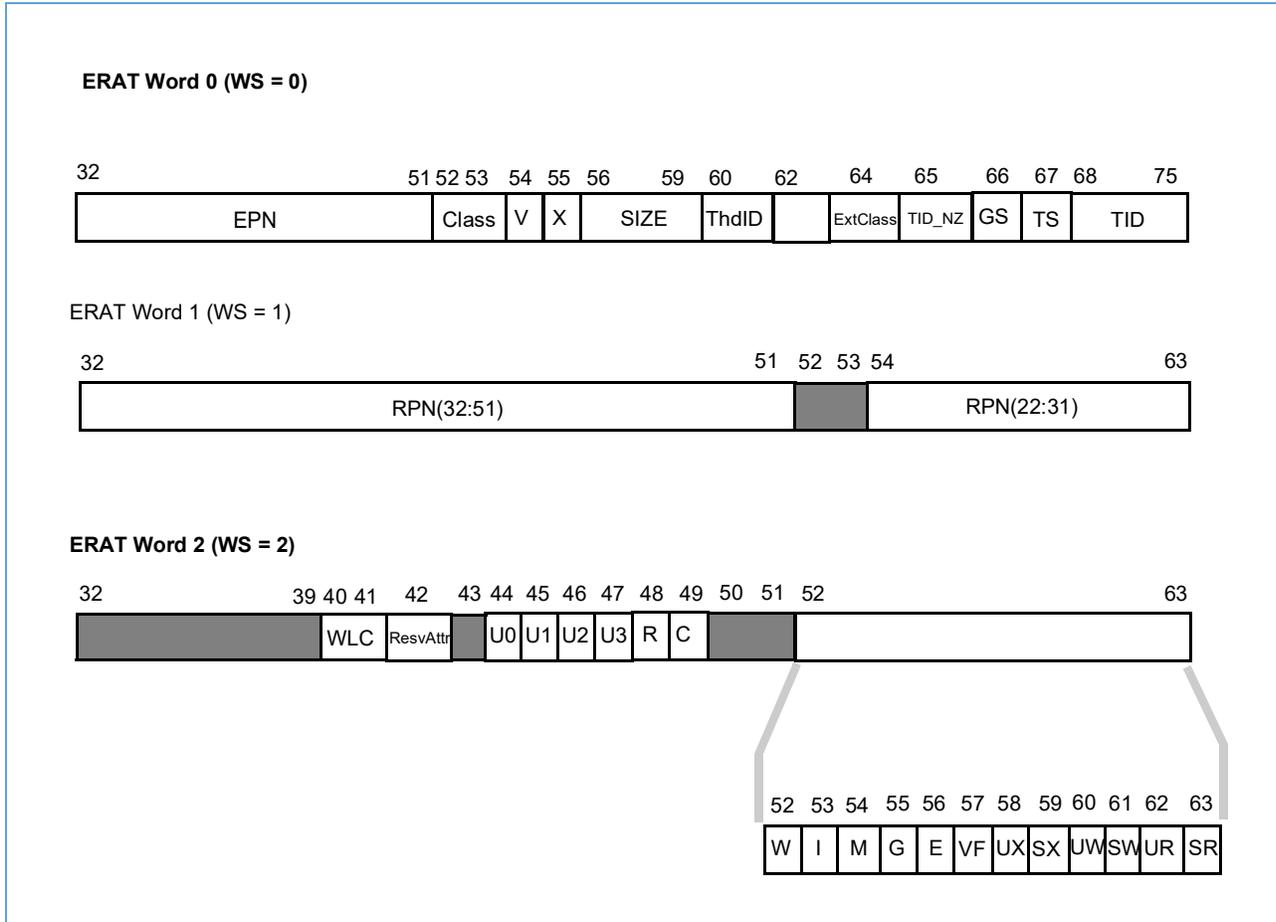
In 32-bit mode operation, the ERAT entries are read from and written to using the **eratre** and **eratwe** instructions, respectively. However, because an ERAT entry contains more than 32 bits, multiple **eratre/eratwe** instructions must be executed to transfer all of the entry information. In 32-bit mode, an ERAT entry is divided into three portions: ERAT word 0, word 1, and word 2. The immediate field (WS), which designates which word of the ERAT entry is to be transferred (that is, WS = 0 specifies ERAT word 0, and so on), is modified to use three values instead of only two in 64-bit mode.

In the A2 design, each of the ERAT caches includes four (one per thread) 64-bit RPN registers that are updated upon **eratwe** of the RPN or attribute portion (WS = 1 or WS = 2). All three portions of the ERAT entry are then updated atomically when **eratwe** is executed with WS = 0 (EPN portion). The value written into the RPN portion of the entry is the data most recently written to the RPN holding register.

The fields in each ERAT word are illustrated in *Figure 6-4*. The bit numbers indicate which bits of the target or source GPR correspond to each ERAT field. Note that the GS, TS, TID, and ExtClass fields of ERAT word 0 are transferred to or from the MMUCR0[TGS], [TS], [TID], and [ECL] fields respectively, rather than to or from the target/source GPR.

Note: The A2 hardware structures that embody the ERATs physically contain 52 bits that represent the EPN in 64-bit mode. In 32-bit mode, the upper 32 bits of the EPN (EPN[0:31]) are zeroed in the source operand of the **eratwe** instruction. This is not the case when creating entries via the **eratwe** instruction in 64-bit mode. When 64-bit mode supervisory software creates entries intended to be used in 32-bit machine state, it must create such entries with the upper 32-bits of the EPN zeroed for 32-bit mode translation compares to succeed.

Figure 4. ERAT Entry Word Definitions for 32-Bit Mode



6.11.8 32-Bit Mode ERAT Search Instruction (eratsx[.])

In 32-bit mode, the **eratsx[.]** instruction is used to locate an entry in the I-ERAT or D-ERAT that is associated with a particular virtual address. This instruction forms an effective address for which an ERAT is to be searched, in the same way data storage access instructions perform their address calculation, by adding the contents of registers RA (or the value 0 if RA = 0) and RB together. This instruction operates essentially the same as the 64-bit mode version (that is, bits 0 to 51 of the EPN are still compared to the contents of the ERAT entry's EPN, assuming a 4 KB page size). However, in 32-bit mode, address bits 0:31 of the effective address are forced to zero before comparison. This implies that the effective page number (EPN) in the ERAT entries that pertain to the current 32-bit process need to have been created with zeros in the upper 32-bits for a search EPN compare to succeed.

6.11.9 32-Bit Mode ERAT Invalidate Virtual Address (Indexed) Instruction (erativax)

The **erativax** instruction is used to invalidate ERAT entries that contain the virtual page number associated with the effective address of this instruction. This instruction operates essentially the same as the 64-bit mode version. However, in 32-bit mode, address bits 0-31 of the effective address are forced to zero before

forwarding invalidation snoops to target processors. This implies that the effective page number (EPN) in the ERAT entries that pertain to the current 32-bit process need to have been created with zeros in the upper 32-bits for an invalidate EPN compare to succeed and invalidation to occur.

6.11.10 32-Bit Mode ERAT Invalidate Local (Indexed) Instruction (*eratlx*)

The *eratlx* instruction is used to invalidate local ERAT entries that contain the virtual page number associated with the effective address of this instruction or, alternately, that contain certain specific values of parameters such as process ID, class, and so forth. This instruction has no effect on the underlying TLB structure (if it exists in a particular MMU implementation).

This instruction operates essentially the same as the 64-bit mode version. However, in 32-bit mode, address bits 0:31 of the effective address are forced to zero before comparison. This implies that the effective page number (EPN) in the ERAT entries that pertain to the current 32-bit process needs to have been created with zeros in the upper 32-bits for an invalidate EPN compare to succeed and invalidation to occur.

6.12 Page Reference and Change Status Management

When performing page management, it is useful to know whether a given memory page has been referenced, and whether its contents have been modified. Note that this might be more involved than determining whether a given TLB entry has been used to reference or change memory, because multiple TLB entries can translate to the same memory page. If it is necessary to replace the contents of some memory page with other contents, a page that has been referenced (accessed for any purpose) is more likely to be maintained than a page that has never been referenced. If the contents of a given memory page are to be replaced and the contents of that page have been changed, the current contents of that page must be written to backup physical storage (such as a hard disk) before replacement.

Similarly, when performing TLB management, it is useful to know whether a given TLB entry has been referenced. When making a decision about which entry of the TLB to replace to make room for a new entry, an entry that has never been referenced is a more likely candidate to be replaced.

The A2O core does not automatically record references or changes to a page or TLB entry. Instead, the interrupt mechanism can be used by system software to maintain reference and change information for TLB entries and their associated pages, respectively.

Execute, read and write access control exceptions can be used to allow software to maintain reference and change information for a TLB entry and for its associated memory page. The following description explains one way in which system software can maintain such reference and change information.

The TLB entry is originally written into the TLB with its access control bits (UX, SX, UR, SR, UW, and SW) off. The first attempt of application code to use the page therefore causes an access control exception and a corresponding instruction or data storage interrupt. The interrupt handler can choose to record the reference to the TLB entry and to the associated memory page in a software table, and then turns on the appropriate access control bit and referenced bit, thereby indicating that the particular TLB entry has been referenced. An initial read from the page is handled by turning on the appropriate UR or SR access control bit and setting the R bit, leaving the page “read-only” and referenced. Subsequent read accesses to the page via that TLB entry proceed normally.

If a write access is later attempted, a write access control exception type of data storage interrupt occurs. The interrupt handler can choose to record the change status to the memory page in a software table, and then turns on the appropriate UW or SW access control bit and the C bit, thereby indicating that the memory page associated with the particular TLB entry has been changed. Subsequent write accesses to the page via that TLB entry proceed normally.

Alternately, software can choose to generate access control exceptions based additionally on the status of the reference and change bits by setting the MMUCR1[REE] (reference exception enable) and/or MMUCR1[CEE] (change exception enable) configuration bits. Software can then initialize TLB entries with appropriate execute, read, and write access controls enabled ahead of time with the reference and change bits set to 0. The first attempt to execute, load, or store into the associated memory page then generates the appropriate instruction or data storage interrupt. The handler then needs only to update the reference or change bits for the entry. Certain touch and lock set and clear instructions are excluded from the set of load and store instructions that can cause the REE and CEE related storage interrupts. This is done to maintain consistency with architectural special cases of access-control exception generation exclusion. The excluded set of instructions is: **dcbt**, **dcbtstp**, **dcbtst**, **dcbtstep**, and **icbt**, and **dcbtls**, **dcbtstls**, **dcblic**, **icbtls**, and **icblic** when CT \neq 0 or 2.

Because the reference and change (R and C) bits are maintained in the TLB entries, recording page management changes to a software table in memory is optional. The R and C bits are transferred to and from TLB entries using the MMUCR3[R,C] fields. The **tlbsx[.]** instruction can be used by page management software to find a TLB entry matching a particular virtual address, followed by a **tlbre** of that entry. Then the R and C bits can be inspected to determine if the page had been previously referenced or changed.

6.13 TLB and ERAT Parity Operations

The TLB and ERAT devices are parity protected against soft errors in TLB and ERAT memory arrays that are caused by alpha particle impacts. If a parity error is detected, the CPU can be configured to vector to the machine check interrupt handler, where software can restore the corrupted state of the TLB (or ERAT) from the page tables in system memory. Alternately, under certain conditions, the parity error detection can be treated as a miss and may be resolved by hardware actions as described below.

The TLB is a 512-entry 4-way set associative RAM with 92 tag bits, 56 data bits, and 20 parity bits (168 bits) per entry. Tag bits are parity protected with 13 parity bits for the 92-bit tag, and 7 parity bits for 56 bits of data. The parity bits are stored in the TLB entries in fields not accessible to software.

The ERATs are 16-entry and 32 -entry, fully associative CAMs with 75 tag bits, 51 data bits, and 17 parity bits (143 bits) per entry. Tag bits are parity protected with 10 parity bits for the 75-bit tag (that is, those read and written as word 0 by the **eratre** and **eratwe** instructions) and 7 parity bits for 51 bits of data (that is, those read and written as word 1 by the **eratre** and **eratwe** instructions). The parity bits are stored in the ERAT entries in fields not accessible to software. The ERAT entry width does not necessarily match the TLB entry width due to the abbreviated form of tag and data contents, and the absence of reserved bits, in the ERATs.

TLB parity bits are set any time the TLB is updated, which is always done either via a **tlbwe** instruction, or by hardware page table translation. TLB parity is checked each time the TLB is searched or read; that is, either refilling the I-ERAT or D-ERAT, or as a result of a **tlbsx[.]**, **tlbsrx.**, or **tlbre** instruction. When executing an I-ERAT or D-ERAT refill, parity is checked for the tag and data words. When executing **tlbsx[.]** or **tlbsrx.**, the data side outputs (RPN, attribute, and access protection bits) are not used for the search translation, so only the tag parity is checked. When executing a **tlbre**, parity is checked for both tag and data components.

Similarly, ERAT parity bits are set any time the ERAT is updated, which is done via either loading an ERAT entry from the TLB, or as the result of an **eratwe** instruction (for software-managed ERAT entries). ERAT parity is checked each time the ERAT is searched or read (that is, either during an address translation, or as a result of an **eratsx** or **eratre** instruction). When executing an I-ERAT or D-ERAT translation, parity is checked for the tag and data words. When executing an **eratsx**, only the tag parity is checked. When executing an **eratre**, parity is checked only for the word specified in the WS field of the **eratre** instruction.

When a parity error is detected during an address translation for fetches, loads, and stores, the subsequent actions taken by hardware are dependent on the setting of the CCR2[NOTLB] and XUCR4[MMU_MCHK] configuration bits. When configured as CCR2[NOTLB] = 0 and XUCR4[MMU_MCHK] = 0, the structure that detects a translation parity error is flash invalidated (including protected entries) for all entries (in the case of ERATs) or all entries in the congruence class (in the case of the TLB). The offending instruction is flushed and replayed. The TLB and ERAT caches subsequently treat the instruction replay like a miss and proceed to either reload the entry with correct parity (in the case of an ERAT miss, TLB direct entry hit), perform a PTE translation via hardware tablewalk (in the case of TLB direct entry miss, indirect entry hit), or generate a TLB miss exception where software can take appropriate action (TLB indirect entry miss). When this processor is configured as CCR2[NOTLB] = 0 and XUCR4[MMU_MCHK] = 1, or when in ERAT-only mode with CCR2[NOTLB] = 1, translation parity error detection causes a machine check exception. If MSR[ME] is set (which is the usual case), the processor takes a machine check interrupt. Similarly, detection of a parity error as the result of a **tlbre**, **tlbsx[.]**, **tlbsrx.**, **eratre**, or **eratsx[.]** instruction also causes a machine check exception; if MSR[ME] is set, the processor takes a machine check interrupt. See *Section 14.5.5 CCR2 - Core Configuration Register 2* and *Section 14.5.142 XUCR0 - Execution Unit Configuration Register 0* for a detailed description of the NOTLB and MMU_MCHK configuration bits.

Certain touch and lock set and clear instructions are excluded from the set of load and store instructions that can cause storage access interrupts or TLB error interrupts, and are treated as a no-op when they detect such exceptions. The excluded set of instructions is: **dcbt**, **dcbtstep**, **dcbtst**, **dcbtstep**, **icbt**, and **dcbtlls**, **dcbtstls**, **dcblic**, **icbtlls**, and **icblic** when CT \neq 0 or 2. However, it should be noted that these instructions still report exceptions due to parity error detection, and may generate resulting machine check interrupts.

6.13.1 Parity Errors Generated from **tlbre** or **eratre**

Because a **tlbre** or **eratre** that detects a parity error causes a machine check exception, the target data facility can only be updated with the TLB or ERAT entry data if the MSR[ME] bit is cleared, preventing the machine check interrupt. Thus, the usual flow of code that detects a parity error in the TLB (or ERAT) and then finds out which entry is erroneous proceeds as follows:

1. A **tlbre** or **eratre** instruction is executed from hypervisor code, resulting in a parity exception. The exception sets **MCSR[TLBPE]**, **MCSR[IEPE]**, or **MCSR[DEPE]**, depending on the source of the parity error (TLB, I-ERAT, or D-ERAT).
2. MSR[ME] = 1, so the CPU vectors to the machine check handler (that is, takes the machine check interrupt) and resets the MSR[ME] bit. Note that even though the parity error causes an *asynchronous* interrupt, that interrupt is guaranteed to be taken before the **tlbre** or **ieratre** instruction completes, and so the MAS registers (in the case of **tlbre**) or the target register (RT in the case of **eratre**) are not updated.
3. The machine check handler code includes a series of **tlbre** (or **eratre**) instructions to query the state of the TLB (or ERAT) and find the erroneous entry. When a **tlbre** (or **eratre**) encounters an erroneous entry and MSR[ME] = 0, the parity exception still happens, setting the **MCSR[TLBPE]** bits (or **MCSR[I/DEPE]** bits). Finally, the instruction completes, since no interrupt is taken because MSR[ME] = 0, updating the target register with data from the TLB (or ERAT).

As is the case for any machine check interrupt, after vectoring to the machine check handler, the MCSRR0 contains the value of the oldest “uncommitted” instruction in the pipeline at the time of the exception and MCSRR1 contains the old (MSR) context. The interrupt handler is able to query the Machine Check Status Register (MCSR) to find out that it was called due to a TLB (or ERAT) parity exception, and then use **tlbre** (or **eratre**) instructions to find the error in the TLB (or ERAT) and restore it from a known good copy in main memory.

Engineering Note: A parity error on the TLB entry *that maps the machine check exception handler* prevents recovery. In effect, one of the 512 TLB entries is unprotected, in that the machine cannot recover from an error in that entry. It is possible to add logic to get around this problem, but the reduction in **SER** achieved by protecting 511 out of 512 TLB entries is sufficient. Further, the software technique of simply dedicating a TLB entry to the page that contains the machine check handler and periodically refreshing that entry from a known good copy can reduce the probability that the entry will be used with a parity error to near zero.

Programming Note: In ERAT-only mode (CCR2[NOTLB] = 1), because an ERAT parity error during translation is treated as a machine check exception, a parity error on the I-ERAT entry *that maps the machine check error exception handler code* can prevent recovery. In effect, one of the 16 I-ERAT entries is unprotected such that the machine cannot recover from an error in that entry. The machine check error handler can experience recursive re-entrance due to this error. Continued looping in this scenario should result in a watchdog timeout critical exception, which exits the recursive looping *if the critical handler is mapped via a separate 4 K page ERAT entry*. A similar re-entrance error can occur for **eratre** generated parity errors in the ERAT entry that maps the machine check handler code. Furthermore, the software technique of simply dedicating a “pinned” ERAT entry to the page that contains either the miss handler or machine check handler, or both, and periodically refreshing that entry from a known good copy can reduce the probability that the entry will be used with a parity error to near zero.

Programming Note: As mentioned above, any **tlbre**, **tlbsx[.]**, **tlbsrx.**, **eratre**, or **eratsx[.]** instruction that causes a machine check interrupt is flushed from the pipeline before it completes. Furthermore, any instruction that causes a D-ERAT or I-ERAT reload that causes a TLB parity error is flushed before it completes, and is retried after the miss has been resolved.

6.13.2 Simulating TLB and ERAT Parity Errors for Software Testing

Because parity errors occur in the TLB and ERATs infrequently and unpredictably, it is desirable to provide users with a way to simulate the effect of a TLB or ERAT parity error so that interrupt handling software can be exercised. This is exactly the purpose of the 6-bit MMUCR1[PEI] parity error inject field.

Usually, parity is calculated as the even parity for each set of bits to be protected, which the checking hardware expects. This calculation is done as the TLB or ERAT data is stored with a **tlbwe** and **eratwe** instructions, when the TLB is reloaded by hardware page table translation, or when the ERAT is reloaded from the TLB. However, **tlbwe** and **eratwe** instruction execution contains an optional parity error injection feature. If the appropriate MMUCR1[PEI] bit is set prior to the **tlbwe** or **eratwe** execution, the calculated parity for the corresponding bits of the data being stored are inverted and stored as odd parity. When the data stored with odd parity is subsequently used for a load, store, or fetch translation, or is used to refill the D-ERAT or I-ERAT from the TLB, or is accessed by a **tlbsx**, **tlbsrx.**, **tlbre**, **eratsx**, or **eratre** instruction, it causes a parity error exception type of machine check interrupt (assuming MSR[ME]=1) and exercises the interrupt handling software. The following pseudo-code is an example of how to use the MMUCR1[PEI] field to simulate a parity error on a TLB entry:

```

mtspr MASn, Rx           ; Setup MAS for EPN, ESEL, and so forth.
mtspr MMUCR1, Rx        ; Set some MMUCR1[PEI] bits.
isync                   ; Wait for the MMUCR1 context to update.
    
```

```

tlbwe                ; Transfer MAS data to TLB data.
isync                ; Wait for the tlbwe to finish.
mfspr MMUCR1, Rz    ; Reset MMUCR1[PEI].
isync                ; Wait for the MMUCR1 context to update.
tlbre                ; tlbre with bad parity causes interrupt.
...
mfspr Rt,MMUCR1    ; Handler reads MMUCR1[TEEN] entry with bad parity.

```

The following pseudo-code is an example of how to use the MMUCR1[PEI] field to simulate a parity error on an ERAT entry:

```

mfspr MMUCR1,Rx    ; Set some MMUCR1[PEI] bits, and [CSINV]=11.
isync              ; Wait for the MMUCR1 context to update.
eratwe Rs,Ra,1    ; Set up real portion word 1 of ERAT data.
eratwe Rs,Ra,0    ; Write some data to the ERAT with bad parity.
isync              ; Wait for the eratwes to finish.
mfspr MMUCR1, Rz  ; Reset MMUCR1[PEI].
isync              ; Wait for the MMUCR1 context to update.
eratre Rt,Ra,WS   ; eratre with bad parity causes interrupt.
...
mfspr Rt,MMUCR1   ; Handler reads MMUCR1[I/DEEN] entry with bad parity.

```

6.14 TLB and ERAT Multi-hit Operations

The TLB and ERAT device compare logic includes detection for matching more than one entry for the same virtual address. This condition is called a “multi-hit” error. These multi-hit errors in the TLB and ERAT memory arrays may be caused by alpha particle impacts or software errors (software installation of more than one entry that matches the same virtual address is considered a programming error). If a multi-hit error is detected, the CPU can be configured to vector to the machine check interrupt handler, where software can restore the corrupted state of the TLB (or ERAT) from the page tables in system memory. Alternately, under certain conditions, the multi-hit detection can be treated as a miss and may be resolved by hardware actions as described below.

TLB multi-hits are *checked* each time the TLB is searched; that is, either refilling the I-ERAT or D-ERAT, or as the result of a **tlbsx[.]** or **tlbsrx**. instruction. The virtual address for these operations is checked for a multi-hit condition across the hashed TLB congruence class. When executing a **tlbre**, multi-hit checking is not defined because this instruction targets a single, particular TLB entry.

ERAT multi-hits are checked each time the ERAT is searched; that is, either during an address translation for fetches, loads, and stores, or as the result of an **eratsx[.]** instruction. When executing an I-ERAT or D-ERAT translation, the virtual address is checked for multi-hits across the entire ERAT contents (in a fully associative manner). Because of the ambiguities that exist in the ERAT's partial virtual address contents, multi-hit detection for **eratsx[.]** instructions is not explicitly defined as an error condition, but rather is reported to software with a status bit. When executing an **eratsx[.]** instruction, multi-hit checking is performed and, if detected, a bit is set in the RT returned data to indicate more than one entry matched the virtual address. When executing an **eratre**, multi-hit checking is not defined because this instruction targets a single, particular

ERAT entry. It should also be noted that multi-hit detection for invalidate snoops caused by **tlbivax**, **tlbilx**, **erativax**, and **eratilx** instructions are not considered errors in this implementation, but rather all entries matching the associated virtual address and size parameters are invalidated.

When a multi-hit error is detected during an address translation for fetches, loads, and stores, the subsequent actions taken by hardware are dependent on the setting of the CCR2[NOTLB] and XUCR4[MMU_MCHK] configuration bits. When configured as CCR2[NOTLB] = 0 and XUCR4[MMU_MCHK] = 0, the structure that detects a translation multi-hit error is flash invalidated (including protected entries) for all entries (in the case of ERATs) or all entries in the congruence class (in the case of the TLB). The offending instruction is flushed and replayed. The TLB and ERAT caches subsequently treat the instruction replay like a miss and proceed to either reload the entry (in the case of an ERAT miss, TLB direct entry hit), perform a PTE translation via hardware tablewalk (in the case of TLB direct entry miss, indirect entry hit), or generate a TLB exception where software can take appropriate action (TLB indirect entry miss). When this processor is configured as CCR2[NOTLB] = 0 and XUCR4[MMU_MCHK] = 1, or when in ERAT-only mode with CCR2[NOTLB] = 1, address translation multi-hit error detection causes a machine check exception. If MSR[ME] is set (which is the usual case), the processor takes a machine check interrupt. Similarly, detection of a multi-hit error as the result of a **tlbsx[.]** or **tlbsrx** instruction also causes a machine check exception; if MSR[ME] is set, the processor takes a machine check interrupt. As described above, execution of the **eratsx[.]** instruction does not generate multi-hit machine check exceptions. See *Section 14.5.5 CCR2 - Core Configuration Register 2* and *Section 14.5.142 XUCR0 - Execution Unit Configuration Register 0* for a detailed description of the NOTLB and MMU_MCHK configuration bits.

Certain touch and lock set and clear instructions are excluded from the set of load and store instructions that can cause storage access interrupts or TLB error interrupts, and are treated as a no-op when they detect such exceptions. The excluded set of instructions is: **dcbt**, **dcbtstp**, **dcbtst**, **dcbtstep**, **icbt**, and **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, and **icblc** when CT \neq 0 or 2. However, it should be noted that these instructions still report exceptions due to multi-hit detection, and may generate resulting machine check interrupts.

6.15 ERAT-Only Mode Operation

Two modes of operation are possible for the A2 address translation depending on whether or not a hardware MMU (TLB based) is present to realize a second-level translation facility. These two modes are termed “MMU mode” and “ERAT-only mode”. This mode controlled by the CCR2[NOTLB] bit. The MMU mode assumes an underlying hardware MMU containing a software-managed TLB. The ERAT-only mode assumes no underlying MMU TLB and relies solely on the ERAT contents (or shadow TLB arrays) for translation. In either mode, minimal hardware MMU support exists on the A2 processor for interprocessor invalidation snooping, assuming the memory subsystem supports these operations.

In MMU mode, the instruction and data ERAT entries (or shadow TLB entries) are ordinarily maintained by hardware via substitution of aging entries with recently requested replacement entries from the unified TLB (UTLB, or unified for both instruction and data translations) in the MMU. In ERAT-only mode, the ERAT entries are directly managed by software, and there is no hardware source for replacement entries (that is, there is no backing UTLB structure assumed).

The TLB management instructions (which use the MAS registers) are dependent on the existence of a hardware UTLB. The ERAT management instructions are not dependent on the existence of a hardware UTLB. These instructions have a direct effect on the ERAT entries in ERAT-only mode of operation. The Power ISA

embedded TLB management instructions are implemented to support software management of cached TLB entries in MMU mode. These instructions include: **tlbre**, **tlbwe**, **tlbsx[.]**, **tlbsrx.**, **tlbivax**, and **tlbilx**. See *Section 6.9 TLB Management Instructions (Architected)* for a detailed description of these instructions.

A set of ERAT management instructions are implemented to support hypervisor software management of cached ERAT entries in either the ERAT-only mode or MMU mode. These instructions rely on the MMUCR0[TLBSEL] selection field to determine the targeted hardware structure for the software management operations (I-ERAT or D-ERAT). These instructions include: **eratre**, **eratwe**, **eratsx[.]**, **erativax**, and **eratilx**. See *Section 6.10 ERAT Management Instructions (Non-Architected)* for a detailed description of these instructions.

6.16 TLB Reservations and TLB Write Conditional (Category E.TWC)

A TLB write conditional facility exists on the A2 processor to improve performance of TLB miss handling in a multiprocessor or multithreaded case. Without the TLB write conditional facility, software must hold a software lock to prevent other processors or threads from updating a shared TLB or invalidating a TLB entry. A TLB reservation can be set by a new **tlbsrx.** instruction before software searches the software page table for the entry to translate the virtual address that got the TLB miss. Because the page size is not known before the page table search, the TLB reservation is defined in terms of a VA, not a VPN. A TLB write is conditional based on this TLB reservation. Because TLB writes by other threads reset the reservation and the **tlbsrx.** instruction can be used to detect TLB entries created by other threads, there is protection against duplicate entries. Also, TLB invalidations by other processors and threads reset the reservation, thereby ensuring that a stale, invalid TLB entry is not created. Because the TLB reservation is set without the page size information, the **tlbivax** and **tlbilx** instructions are defined to specify the page size when the TLB write conditional facility is supported. The **tlbsrx.** instruction is executable by a guest operating system. A guest operating system that uses this instruction on an implementation that supports a logical-to-real-address-translation capability (as the A2 processor does) can handle a TLB miss without trapping to the hypervisor.

The **tlbsrx.** instruction and **tlbwe** instruction with $MAS0_{WQ} = 0b01$ (termed a “TLB write conditional” operation) together allow for software to write a TLB entry while ensuring that the entry is not a duplicate entry and is not a stale or invalid entry. The **tlbsrx.** instruction has two side effects that occur at the same time:

1. A TLB reservation is established for the virtual address and the associated IND value, and
2. A search of the TLB array is performed for the virtual address.

The TLB reservation is used by a subsequent **tlbwe** instruction that writes a TLB entry (that is, $MAS0_{ATSEL} = 0$ or $MSR_{GS} = 1$) with $MAS0_{WQ} = 0b01$. The TLB is only written by this **tlbwe** if the TLB reservation still exists at the instant the TLB is written. A **tlbwe** that writes the TLB is said to “succeed.” TLB write conditional cannot be used for the LRAT.

In this processor, a TLB reservation is composed of a hardware lookaside latch that captures the virtual address and the IND value associated with the **tlbsrx.** instruction. The contents of the TLB reservation latch are shown in *Table 6-13*. There is a separate TLB reservation latch for each hardware processing thread on the A2 processor (that is, a total of four TLB reservation latches are implemented on A2).

Table 13. TLB Reservation Fields

Field	Width (Bits)	Notes
GS	1	Guest State Set to the value of MAS8 _{TGS} when the reservation is established. ¹
LPID	8	Logical Partition Identifier Set to the value of MAS8 _{TLPID} when the reservation is established. ¹
AS	1	Address Space Set to the value of MAS1 _{TS} when the reservation is established. ¹
PID	14	Process Identifier Set to the value of MAS1 _{TID} when the reservation is established. ¹
EPN	52	Effective Page Number Set to the value of tlbsrx . instruction's EA _{0:51} when the reservation is established. ¹
IND	1	Indirect Set to the value of MAS1 _{IND} when the reservation is established. ¹
Class ³	2	Class Set to the value of MMUCR3 _{CLASS} when the reservation is established. ¹
V	1	Valid Set to '1' when the reservation is established. ¹ Set to '0' when the reservation is cleared. ²

1. A TLB reservation is established only by execution of the **tlbsrx**. instruction.
2. A TLB reservation can be cleared by several events described elsewhere in this section.
3. These fields are nonarchitected, implementation-specific fields.

A TLB reservation is established or set (the reservation latch fields are updated and the valid bit is set to '1'), only by execution of the **tlbsrx**. instruction. The result of the search of the TLB is irrelevant with respect to the establishment of the reservation. There is no specific page size associated with the TLB reservation. The TLB reservation applies to any virtual page that contains the virtual address.

A TLB reservation is cleared by any of the following events:

1. The thread holding the TLB reservation executes another **tlbsrx**. This clears the first TLB reservation and establishes a new one.
2. A **tlbivax** is executed by any thread in the system, and all the following conditions are met:
 - a. The MAS5_{SGS} and MAS5_{SLPID} values used by the **tlbivax** match the GS and LPID values associated with the TLB reservation.
 - b. The MAS6_{SPID} and MAS6_{SAS} values used by the **tlbivax** match the PID and AS values associated with the TLB reservation.
 - c. The EA_{31:n-1} values of the **tlbivax** match the EPN_{31:n-1} values associated with the TLB reservation, where $n = 64 - \log_2(\text{page size in bytes})$ and *page size* is specified by the MAS6_{ISIZE}. The subset of EA_{31:n-1} (not EA_{0:n-1}) is used because only this subset of EA bits is transferred with **tlbivax** transactions for all supported page sizes over the system bus structure.

MAS6_{SIND} is part of the **tlbivax** invalidation criteria so that unnecessary invalidation of entries, especially indirect entries, can be avoided. However, this bit is not part of the criteria for a **tlbivax** clearing a TLB reservation. A matching TLB reservation needs to be cleared by **tlbivax** regardless of the IND bit to facilitate hypervisor searches of a guest page table for some LRAT miss cases.

3. The thread holding the TLB reservation, or another thread that shares the TLB with this thread, executes an **mtspr** to MMUCSR0 that performs a TLB invalidate all operation; and the LPIDR contents of the thread executing the **mtspr** matches the LPID value associated with the TLB reservation.
4. If a **tlbilx** with $T = 0$ (invalidate all in logical partition) is executed by the thread holding the TLB reservation, or by a thread that shares the TLB with this thread, and the $MAS5_{SLPID}$ value used by the **tlbilx** matches the LPID value associated with the TLB reservation.
5. If a **tlbilx** with $T = 1$ (invalidate by PID in logical partition) is executed by the thread holding the TLB reservation, or by a thread that shares the TLB with this thread, and the $MAS5_{SLPID}$ and $MAS6_{SPID}$ values used by the **tlbilx** match the LPID and PID values associated with the TLB reservation.
6. If a **tlbilx** with $T = 3$ (invalidate by VA in logical partition) is executed by the thread holding the TLB reservation, or by a thread that shares the TLB with this thread, the $MAS5_{SGS}$, $MAS5_{SLPID}$, $MAS6_{SPID}$, and $MAS6_{SAS}$ values used by the **tlbilx** match the GS, LPID, PID, and AS values associated with the TLB reservation, and $EA_{0:n-1}$ values of the **tlbilx** match the $EPN_{0:n-1}$ values associated with the TLB reservation, where $n = 64 - \log_2(\text{page size in bytes})$ and *page size* is specified by the $MAS6_{SIZE}$.
7. A **tlbwe** instruction is executed by the thread holding the TLB reservation, or by a thread that shares the TLB with this thread, and all the following are true:
 - a. An interrupt does not occur as a result of the **tlbwe** instruction.
 - b. The $MAS8_{TLPID}$ value used by the **tlbwe** matches the LPID value associated with the TLB reservation.
 - c. The $MAS8_{TGS}$ value used by the **tlbwe** matches the GS value associated with the TLB reservation.
 - d. The $MAS1_{TID}$ value used by the **tlbwe** matches the PID value associated with the TLB reservation.
 - e. The $MAS1_{IND}$ value used by the **tlbwe** matches the IND value associated with the TLB reservation.
 - f. The $MAS1_{TS}$ value used by the **tlbwe** matches the AS value associated with the TLB reservation.
 - g. Bits $0:(n-1)$ of $MAS2_{EPN}$ used by the **tlbwe** match the $EPN_{0:n-1}$ values associated with the TLB reservation, where $n = 64 - \log_2(\text{page size in bytes})$ and *page size* is specified by the $MAS1_{TSIZE}$ used by the **tlbwe**.
 - h. Either of the following conditions are met:
 - (1) The $MAS0_{WQ}$ used by the **tlbwe** instruction is 0b00 (write always).
 - (2) The $MAS0_{WQ}$ used by the **tlbwe** instruction is 0b01 (TLB write conditional), and the TLB reservation for the thread executing the **tlbwe** exists.
8. The thread that has the TLB reservation or another thread that shares the TLB with this thread, as a result of a Page Table translation, writes a TLB entry; and all the following conditions are met:
 - a. The TS and $EPN_{0:n-1}$ values for the new TLB entry match the corresponding values associated with the TLB reservation, where $n = 64 - \log_2(\text{page size in bytes})$ and *page size* is specified by the SIZE value written to the TLB entry.
 - b. The TLPID for the new TLB entry matches the LPID associated with the TLB reservation.
 - c. The TGS for the new TLB entry matches the GS associated with the TLB reservation.
 - d. The TID for the new TLB entry matches the PID associated with the TLB reservation.
 - e. The Valid bit for the new TLB entry is 1.
 - f. The IND value associated with the TLB reservation is 0 (that is, page table translations do not affect established "indirect" reservations).

Implementations are allowed to clear a TLB reservation for conditions other than those specified above. The architecture ensures that a TLB reservation is cleared when required per the above requirements, but does not guarantee that these are the only conditions for clearing a TLB reservation. However, the occurrence of an interrupt does not clear a TLB reservation.

Aside from the $EA_{31:n-1}$ aliases that occur for **tlbivax** operations, the A2 processor defines the following additional, implementation-specific TLB reservation clear events:

1. A **tlbwe** instruction is executed by the thread holding the TLB reservation or by a thread that shares the TLB with this thread, and all the following are true.
 - a. An interrupt does not occur as a result of the **tlbwe** instruction.
 - b. The $MAS8_{TLPID}$ value used by the **tlbwe** matches the LPID value associated with the TLB reservation.
 - c. The $MAS8_{TGS}$ value used by the **tlbwe** matches the GS value associated with the TLB reservation.
 - d. The $MAS1_{TID}$ value used by the **tlbwe** matches the PID value associated with the TLB reservation.
 - e. The $MAS1_{IND}$ value used by the **tlbwe** matches the IND value associated with the TLB reservation.
 - f. The $MAS1_{TS}$ value used by the **tlbwe** matches the AS value associated with the TLB reservation.
 - g. Bits $0:(n-1)$ of $MAS2_{EPN}$ used by the **tlbwe** match the $EPN_{0:n-1}$ values associated with the TLB reservation, where $n = 64 - \log_2(\text{page size in bytes})$ and *page size* is specified by the $MAS1_{TSIZE}$ used by the **tlbwe**.
 - h. Either of the following conditions are met.
 - (1) The $MAS0_{WQ}$ used by the **tlbwe** instruction is 0b10 (clear reservation without writing TLB).
 - (2) The $MAS0_{WQ}$ used by the **tlbwe** instruction is 0b11 (this $MAS0_{WQ}$ reserved setting is treated the same as the setting of 0b00, or write TLB always).
2. A **tlbilx** instruction is executed by the thread holding the TLB reservation or by a thread that shares the TLB with this thread, and any of the following are true:
 - a. $T = 4$ and the Class value associated with the TLB reservation equals 0.
 - b. $T = 5$ and the Class value associated with the TLB reservation equals 1.
 - c. $T = 6$ and the Class value associated with the TLB reservation equals 2.
 - d. $T = 7$ and the Class value associated with the TLB reservation equals 3.

6.17 Hardware Page Table Walking (Category E.PT)

This processor supports the Power ISA Category Embedded.Page Table (E.PT) and the embedded MMU Architecture Version 2.0 (MAV 2.0). Because this processor also supports the Embedded.Hypervisor (E.HV) category, the Embedded.Hypervisor.LRAT (E.HV.LRAT) category is also required and supported. Because of this, hypervisor software must always ensure that at least one valid logical to real address translation (LRAT) entry exists.

Software can manage translation directly by installing TLB entries, and indirectly by setting up hardware page tables in memory, which the TLB hardware page table walker can subsequently fetch and cache in the TLB array. A hardware page table is a variable-sized data structure that specifies the mapping between virtual page numbers and real page numbers. There can be many hardware page tables. Each page table is defined by an indirect TLB entry. An indirect TLB entry is an entry that contains the parameter $IND = 1$.

6.17.1 Searching the TLB for Direct and Indirect Entries

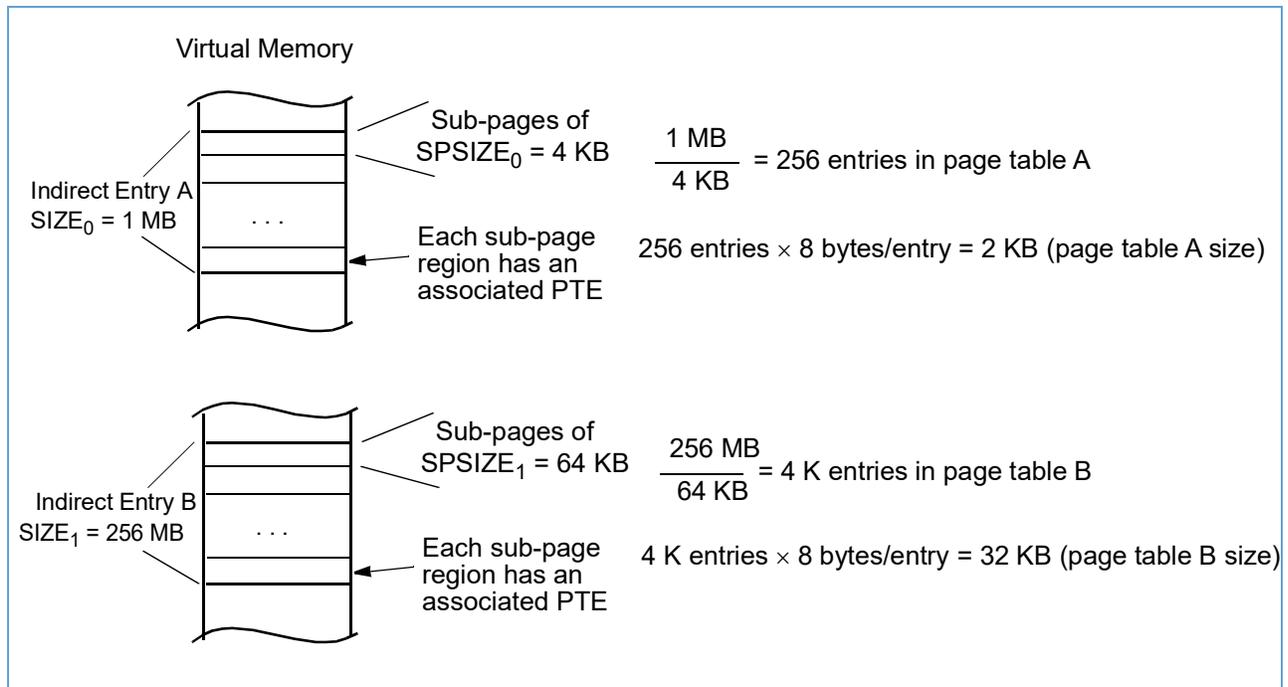
A direct ($IND = 0$) or an indirect TLB entry ($IND = 1$) matches the virtual address if all fields match per *Section 6.2.4 TLB Match Process*. The TLB is searched for matching direct entries first according to the page size order dictated by MMUCR2. If no matching direct TLB entries are found, the TLB is then searched for 1 MB indirect entries, followed by 256 MB indirect entries. If a valid 1 MB indirect entry is found, the search process is discontinued. If there is one and only one matching indirect entry in the associated TLB congruence class being searched, the indirect entry is used to access a page table entry (PTE). If the PTE is valid (V bit equals "1"), the PTE is installed in the TLB and used to translate the virtual address. The PTE entry format is described in *Section 6.16.3 Hardware Page Table Entry Format*. The abbreviated real page number (ARN) from the PTE is treated as a logical page number (LPN), and this LPN is subsequently translated by the LRAT into an RPN before being installed into the TLB. If there is more than one matching direct TLB entry or more than one matching indirect TLB entry for any calculated TLB congruence class, a machine check exception is generated.

6.17.2 Indirect TLB Entry Page and Sub-Page Sizes

Each indirect TLB entry represents a hardware page table in memory, and there can be many disjoint page tables existing in various areas of real memory. Each indirect entry has an associated page size (the size of the virtual address area covered by this indirect entry, or the entry TSIZE field) and a sub-page size (denoting smaller, same-sized “chunks” of the parent indirect entry page size). This processor supports two combinations of page and sub-page sizes: 1 MB page size with 4 KB sub-page size and 256 MB page size with 64 KB sub-page sizes. These combinations are indicated by the read-only EPTCFG register.

The overall size of each hardware page table is determined by the associated indirect entry page size and sub-page size. Each of the page table entries (PTEs) of a particular hardware page table is 8 bytes (64 bits) in length. For this processor, hardware page tables are either 2 KB bytes in length or 32 KB bytes in length. This calculation is shown in Figure 6-5.

Figure 3. Indirect Entry to Page Table Size Calculation



The starting address of a given page table must be aligned to the page table’s size. This virtual linear page table placement with a given page size and sub-page size results in the page table’s starting real address LSB being determined by a certain bit of the indirect entry’s RPN field. For the 1 MB/4 KB combination (a 2 KB page table size), the indirect entry RPN[52] is used as the LSB of the base real address of this page table (that is, RA[53:60] is a given PTE’s offset within that page table). For the 256 MB/64 KB combination (a 32 KB page table size), the indirect entry RPN[48] is used as the LSB of the base real address of this page table (that is, RA[49:60] is a given PTE’s offset within that page table). For this implementation, the indirect entry RPN[53] is not required and is therefore treated as a reserved bit in both the TLB and MAS3.

Note: Even though each of the sub-page regions of a given indirect entry are represented by a backing PTE, each PTE within a page table contains a variable page size field (see for the PTE format). This means that a given PTE page size does not necessarily have to match the sub-page size of the indirect entry used to find this PTE (that is, IND = 1 Entry[SPSIZE] does not necessarily equal PTE[Page Size]). For example, an operating system might choose to use 16 MB pages (a supported direct page size for this processor) in a particu-

BAP[0:5] Field	Base access permission bits. BAP[0] = UX, BAP[1] = SX, BAP[2] = UW, BAP[3] = SW, BAP[4] = UR, BAP[5] = SR. See <i>Section 6.16.5 Hardware Page Table Errors and Exceptions</i> to see how the base access permission bits are modified by the R and C bits to form the storage access control bits that are actually stored into the TLB entry.
V Field	The valid bit. The PTE entry is valid when V = 1.

6.17.4 Calculation of Hardware Page Table Entry Real Address

Although this processor implements only power of 4×1 K page sizes and sub-page sizes, for the sake of this example, TSIZE and SPSIZE are interpreted as a power of 2×1 K page size and sub-page size (as architected). The page table entry that is used to translate the virtual address is selected by a real address formed from a combination of RPN bits from the indirect TLB entry and some EA bits. The low-order m bits of the RPN(0:53) field in the indirect TLB entry must be zeros, where m is the larger of 0 and $(TSIZE + 8 - SPSIZE)$. For this processor, $m = 16$ (TSIZE = 10 for 1 MB indirect entries, and SPSIZE = 2 for 4 KB), or $m = 20$ (TSIZE = 18 for 256 MB indirect entries, and SPSIZE = 6 for 64 KB).

The TSIZE and SPSIZE fields of the indirect TLB entry determine which bits of the RPN and EA are used in the following manner:

1. $EA_{23:51}$ are shifted right q bits, according to a decode of SPSIZE, to produce a 29-bit result S . The value of q is $(SPSIZE - 2)$. Bits shifted out of the rightmost bit position are lost.

For this processor, $q = 0$ or 4 (for SPSIZE of 2 or 6). Therefore, $S_{0:28} = EA_{23:51}$ or $S_{0:28} = {}^4 0 \parallel EA_{23:47}$.

2. A 21-bit EA mask M is formed based on a decode of TSIZE and SPSIZE. The EA mask M is $({}^{29 - (TSIZE - SPSIZE)} 0 \parallel (TSIZE - SPSIZE) - 8 \ 1)$.

For this processor, $M = {}^{21} 0 \parallel {}^0 1$, or $M = {}^{17} 0 \parallel {}^4 1$.

3. The EA mask M from step 2 is ANDed with the high-order 21 bits of the shifted EA result ($S_{0:20}$) from step 1 to form a 21-bit result W .

For this processor, $W = {}^{21} 0$, or $W = {}^{17} 0 \parallel S_{17:20} = {}^{17} 0 \parallel EA_{36:39}$.

4. $RPN_{32:52}$ from the indirect TLB entry is ORed with the 21 bits of the result W from step 3 to form a 21-bit result R .

For this processor, $R = RPN_{32:52}$, or $R = RPN_{32:48} \parallel S_{17:20} = RPN_{32:48} \parallel EA_{36:39}$.

5. The 64-bit real address of the PTE is formed as follows: $RA_{PTE} = TLBE_{RPN[0:31]} \parallel R \parallel S_{21:28} \parallel 0b000$

For this processor, the 42-bit real address truncation of the page table entry real address RA_{PTE} is:

$RA_{PTE} = TLBE_{RPN[22:31]} \parallel TLBE_{RPN[32:52]} \parallel EA_{44:51} \parallel 0b000$ (for 1 MB / 4 KB combination), or

$RA_{PTE} = TLBE_{RPN[22:31]} \parallel TLBE_{RPN[32:48]} \parallel EA_{36:39} \parallel EA_{40:47} \parallel 0b000$ (for 256 MB / 64 KB combination).

The doubleword addressed by the real address result from step 5 is the PTE used to translate the VA if the PTE is valid (valid bit is "1"). The logical address (LA) result is formed by concatenating 0x000 with the $ARPN_{12:51-p}$ from the PTE and with the low-order p bits of EA, where p equals $\log_2(\text{page size specified by } PTE_{PS})$, or $LA = 0x000 \parallel ARPN_{12:51-p} \parallel EA_{64-p:63}$.

For this processor, the 42-bit truncated logical address LA becomes:

$$LA = PTE_{\text{ARPN}[22:51-p]} \parallel EA_{64-p:63}, \text{ where } p = \log_2(\text{page size specified by } PTE_{\text{PS}}).$$

Finally, if the indirect entry's TGS = 1 (a guest page table), this 42-bit logical address is converted to a real address by translation through the LRAT before being stored in the TLB cache. If there is no matching entry in the LRAT for this logical address, an LRAT miss exception occurs. If the page table entry that is accessed is invalid (valid bit V = 0), a page table fault exception occurs.

6.17.5 Hardware Page Table Errors and Exceptions

There are architected and implementation-specific errors associated with hardware page table utilization that can result in certain exceptions being generated. As a result of page table translation, a corresponding TLB entry is created if no exception occurs, and is not created if certain exceptions occur as described below.

A TLB entry is not written as a result of a page table translation if a page table fault exception occurs. A page table fault exception occurs when the hardware page table walker logic encounters a page table entry with $PTE_V = 0$ before writing the TLB entry.

A TLB entry is not written as a result of a page table translation if a TLB ineligible exception occurs. A TLB ineligible exception occurs upon an attempt to translate via the hardware page table if any of the following conditions are met:

1. The TLB array cannot be loaded from the page table (that is, $TLB0CFG_{PT} = 0$) based on a particular system boot configuration setting, or
2. The TLB array does not support the page size specified by PTE_{PS} , or
3. The TLB congruence class chosen for entry insertion contains four valid entries, all with $IPROT = 1$ (that is, all entries are protected and cannot be overwritten).

A TLB entry is not written as a result of a page table translation if an LRAT miss exception occurs. An LRAT miss exception occurs when the hardware page table walker logic requests LRAT translation of the PTE_{ARPN} value before writing the TLB entry, and there is no matching LRAT entry found. This LRAT translation request can happen when a PTE translation occurs as the result of a guest installed (TGS = 1) indirect TLB entry (IND = 1).

A TLB entry is written, assuming none of the above exceptions occurred, regardless of the existence of a virtualization fault exception, or an execute, read, or write access control exception. The PTE entry is installed in the TLB regardless of the state of the associated VF bit, or the derived access control values (UX, SX, UW, SW, UR, and SR). It is the subsequent "replay" of the instruction fetch, or data load or store, or cache management operation that will eventually encounter and generate a virtualization fault exception (when the matching D-ERAT entry's VF bit is set), or an execute, read, or write access control exception (if warranted by the matching ERAT entry's access control bits).

6.17.6 Hardware Page Table Storage Control Attributes

A page table must be located in storage that is big-endian (E = 0), memory coherence required (M = 1), not caching inhibited (I = 0) and not guarded (G = 0). If the translation of a virtual address matches an indirect TLB entry that has its storage control attribute E bit equal to 1, M bit equal to 0, I bit equal to 1, or G bit equal to 1, the transaction is performed without hardware flagging an error. That is, the storage control attributes for the page table that are present in the indirect entry's WIMGE bits are presented unmodified to the memory

subsystem when the hardware walker fetches a PTE entry. It is the responsibility of software installing the indirect TLB entry to ensure that the WIMGE settings are valid. Execution of a **tlbwe** with $MAS1_{IND} = 1$ and an invalid combination of $MAS2_{WIMGE}$ results in an illegal instruction exception.

6.17.7 TLB Update After Hardware Page Table Translation

The architected and implementation-specific fields of the resulting entry that is written to the TLB after hardware page table translation is shown in *Table 6-14*.

Table 14. TLB Update After Page Table Translation (Sheet 1 of 2)

TLB Field	Architected?	New Value after Page Table Translation
EPN _{0:p-1}	Y	EA _{0:p-1} , where $p = 64 - \log_2(\text{page size in bytes as specified by } PTE_{PS})$. Any low-order EPN bits in the TLB entry that correspond to byte offsets within the page are set to zero.
TS	Y	TS from the indirect entry.
TGS	Y	TGS from the indirect entry.
TID	Y	TID from the indirect entry.
TLPID	Y	TLPID from the indirect entry.
V	Y	PTE_V
SIZE ¹	Y	0b0 $PTE_{PS}[52:54]$
IND	Y	0
IPROT	Y	0
VF	Y	0
RPN _{22:p-1}	Y	$LPN_{22:p-1} = PTE_{ARPN}[22:p-1]$, where $p = 64 - \log_2(\text{page size in bytes as specified by } PTE_{PS})$. $RPN_{22:p-1}$ = result of LRAT translation of LPN and PTE_{PS} . Any low-order RPN bits in the TLB entry that correspond to byte offsets within the page are set to zero.
WIMGE	Y	PTE_{WIMGE}
U0:U3	Y	$PTE_{U0:U3}$
UX	Y	Logical AND of $PTE_{BAP[0]}$ and PTE_R (that is, $UX = BAP[0]$ and R).
SX	Y	Logical AND of $PTE_{BAP[1]}$ and PTE_R (that is, $SX = BAP[1]$ and R).
UW	Y	Logical AND of $PTE_{BAP[2]}$ and PTE_R and PTE_C (that is, $UW = BAP[2]$ and R and C).
SW	Y	Logical AND of $PTE_{BAP[3]}$ and PTE_R and PTE_C (that is, $SW = BAP[3]$ and R and C).
UR	Y	Logical AND of $PTE_{BAP[4]}$ and PTE_R (that is, $UR = BAP[4]$ and R).
SR	Y	Logical AND of $PTE_{BAP[5]}$ and PTE_R (that is, $SR = BAP[5]$ and R).
ThdID	N	THDID from the indirect entry.
Class	N	0b10 when the TLB update is caused by an external PID load, or 0b11 when the TLB update is caused by an external PID store; otherwise, 0b00.
ExtClass	N	0
TID_NZ	N	Logical OR of all TID bits from indirect entry ($or_reduce(TID_{0:13})$).
X	N	0
R	N	PTE_R

1. The TLB page size field supported by this implementation is defined as a power of 4×1 KB; hence, the LSB of the PTE_{PS} field (which is a power of 2 based field) is dropped before installing the TLB entry.

Table 14. TLB Update After Page Table Translation (Sheet 2 of 2)

TLB Field	Architected?	New Value after Page Table Translation
C	N	PTE _C
WLC	N	0b00
ResvAttr	N	0

1. The TLB page size field supported by this implementation is defined as a power of 4×1 KB; hence, the LSB of the PTE_{PS} field (which is a power of 2 based field) is dropped before installing the TLB entry.

6.18 Storage Control Registers (Architected)

This section describes the specific implementation of the architected storage control related registers. In addition to the registers described below, the MSR[IS,DS] bits specify which of the two address spaces the respective instruction or data storage accesses are directed towards. Also, the MSR[PR] bit is used by the access control mechanism. See *Machine State Register (MSR)* on page 285 for more detailed information about the MSR and the function of each of its bits.

Note: The A2 reserved, unimplemented fields in architected MMU registers include: MAS0.TLBSEL, MAS0.NV, MAS2.ACM, MAS2.VLE, MAS4.TLBSELD, MAS4.TIDSELD, MAS4.ACMD, MAS4.VLED, MMUCFG.NPIDS (Category: Phased Out), MMUCSR0.TLBn_PS (n = 0 to 3), MMUCSR0.TLBn_FI (n = 1 to 3), TLB0CFG.MINSIZE, TLB0CFG.MAXSIZE, TLB0CFG.AVAIL, TLB1CFG.MINSIZE, TLB1CFG.MAXSIZE, and TLB1CFG.AVAIL.

6.18.1 Process ID Register (PID)

The PID is a 64-bit register, although only the lower 14 bits are defined in the A2 core. The 14-bit PID value is used as a portion of the virtual address for accessing storage (see *Section 6.2.1 Virtual Address Formation* on page 171). The PID value is compared against the TID field of a TLB entry to determine whether or not the entry corresponds to a given virtual address. If an entry's TID field is 0 (signifying that the entry defines a "global" as opposed to "private" page), the PID value is ignored when determining whether the entry corresponds to a given virtual address. See *Section 6.2.4 TLB Match Process* on page 173 for a more detailed description of the use of the PID value in the TLB match process.

The PID is written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**. The following table illustrates the PID register.

Register Short Name:	PID	Read Access:	Priv
Decimal SPR Number:	48	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:49	///	0x0	<u>Reserved</u>
50:63	PID	0x0	<u>Process ID</u> Process ID Register is used by system software to specify which TLB entries are used by the processor to accomplish address translation for loads, stores, and instruction fetches

6.18.2 Logical Partition ID Register (LPIDR)

The LPIDR is written from a GPR using **mtspr** and can be read into a GPR using **mfspr**. This register is shared between all processing threads. Therefore, software locking is recommended to access this register.

The LPIDR register contains the 8-bit logical partition identifier (LPID) for this core. All processing threads on a core are, by default, in the same logical partition at any point in time. The LPID value is forwarded to the memory subsystem for certain off-core transactions. In the case of ERAT-only mode memory management, the LPID value is used to tag outgoing global ERAT invalidation requests and to filter incoming global invalidation snoops (that is, only allow invalidations to be seen by targeted processors if they reside in the same partition as the source processor). In MMU mode (that is, TLB exists), the MAS5[SLPID] value is used to tag outgoing TLB invalidation requests and, normally, all global invalidation snoops are accepted. However, this register's contents is used to filter incoming global invalidation snoops when MMUCR1[TLBI_REJ] = 1. See *Section 6.18.2 Memory Management Unit Control Register 1 (MMUCR1)* for a description of this function.

Register Short Name:	LPIDR	Read Access:	Hypv
Decimal SPR Number:	338	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:55	///	0x0	<u>Reserved</u>
56:63	LPID	0x0	<u>Logical Partition ID</u> The LPID is part of the virtual address and is used during address translation comparing LPID to the TLPID field in the TLB entry to determine a matching TLB entry.

6.18.3 External PID Load Context (EPLC) Register

The EPLC is written from a GPR using **mtspr** and can be read into a GPR using **mfspir**. The EPLC register contains fields that provide the context for external PID load instructions. The external versions of the address space, guest state, logical partition ID, and process ID (EAS, EGS, ELPID, and EPID) are substituted as a portion of the virtual address when accessing storage using external PID load instructions (see *Section 6.2.1 Virtual Address Formation* on page 171). The external problem state bit (EPR) is also substituted for MSR[PR] when using external PID loads.

Register Short Name:	EPLC	Read Access:	Priv
Decimal SPR Number:	947	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	EPR	0b0	<u>External Load Context PR Bit</u> Used in place of MSR[PR] by the storage access control mechanism when an External Process ID Load instruction is executed. 0 Supervisor mode 1 User mode
33	EAS	0b0	<u>External Load Context AS Bit</u> Used in place of MSR[DS] for translation when an External Process ID Load instruction is executed. 0 Address space 0 1 Address space 1
34	EGS ^{HO}	0b0	<u>External Load Context GS Bit</u> ^{HO} Used in place of MSR[GS] for translation when an External Process ID Load instruction is executed. 0 Embedded Hypervisor state. 1 Guest state. This field is only writable in hypervisor state
35:39	///	0x0	<u>Reserved</u>
40:47	ELPID ^{HO}	0x0	<u>External Load Context Logical Process ID Value</u> ^{HO} Used in place of LPID register 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
48:49	///	0b00	<u>Reserved</u>
50:63	EPID	0x0	<u>External Load Context Process ID Value</u> Used in place of all Process ID register values for translation when an external Process ID Load instruction is executed.

6.18.4 External PID Store Context (EPSC) Register

The EPSC is written from a GPR using **mtspr** and can be read into a GPR using **mfspr**. The EPSC register contains fields that provide the context for external PID store instructions. The external versions of the address space, guest state, logical partition ID, and process ID (EAS, EGS, ELPID, and EPID) are substituted as a portion of the virtual address when accessing storage using external PID store instructions (see *Section 6.2.1 Virtual Address Formation* on page 171). The EPR is also substituted for MSR[PR] when using external PID stores.

Register Short Name:	EPSC	Read Access:	Priv
Decimal SPR Number:	948	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	EPR	0b0	<u>External Store Context PR Bit</u> Used in place of MSR[PR] by the storage access control mechanism when an External Process ID Store instruction is executed. 0 Supervisor mode 1 User mode
33	EAS	0b0	<u>External Store Context AS Bit</u> Used in place of MSR[DS] for translation when an External Process ID Store instruction is executed. 0 Address space 0 1 Address space 1
34	EGS ^{HO}	0b0	<u>External Store Context GS Bit^{HO}</u> Used in place of MSR[GS] for translation when an External Process ID Store instruction is executed. 0 Embedded Hypervisor state. 1 Guest state. This field is only writable in hypervisor state
35:39	///	0x0	<u>Reserved</u>
40:47	ELPID ^{HO}	0x0	<u>External Store Context Logical Process ID Value^{HO}</u> Used in place of LPID register value for load translation when an External PID Store instruction is executed. Compared with TLB[TLPID] during translation. This field is only writable in hypervisor state
48:49	///	0b00	<u>Reserved</u>
50:63	EPID	0x0	<u>External Store Context Process ID Value</u> Used in place of all Process ID register values for translation when an external Process ID Store instruction is executed.

6.18.5 MMU Assist Register 0 (MAS0)

The MAS0 register is written from a GPR using **mtspr** and can be read into a GPR using **mfspr**. This register is replicated for all processing threads. MAS0 is used to define which array should be targeted (the TLB or the LRAT) for the TLB management instructions, and it is also used to parameterize and condition certain management instructions. The architected fields MAS0.TLBSEL and MAS0.NV are reserved in this implementation (there is only one TLB, and TLB next victim support is not implemented). The MAS0.ESEL field implements only 3 bits (a 4-way set-associative TLB and a maximum of eight fully-associative LRAT entries are supported in this implementation).

Register Short Name:	MAS0	Read Access:	Priv
Decimal SPR Number:	624	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	ATSEL	0b0	<u>Array Type Select</u> TLB or LRAT structure selection. When in guest mode (MSR[GS]=1), ATSEL is treated as if it were zero such that the TLB is always selected. 0 - TLB 1 - LRAT
33:44	///	0x0	<u>Reserved</u>
45:47	ESEL	0b000	<u>Entry Select</u> TLB and LRAT Entry selection. Identifies an entry in the selected array to be used for tlbwe and tlbre. Valid values for ESEL are from 0 to TLBnCFG[ASSOC] - 1. When MAS0[ATSEL]=0 (TLB), ESEL becomes effectively a TLB way select and valid values are 0-3 (bit 45 is treated as reserved). When MAS0[ATSEL]=1 (LRAT), valid values are 0-7.
48	///	0b0	<u>Reserved</u>
49	HES	0b0	<u>Hardware Entry Select</u> Determines how the TLB entry way is selected by tlbwe when MAS0[ATSEL] = 0 (TLB). This bit must be set to '0' when MAS0[ATSEL]=1 (LRAT), or an illegal instruction exception may occur for tlbwe. 0 - The TLB entry way is selected by MAS0.ESEL[1:2]. 1 - The TLB entry way is selected by the hardware LRU replacement algorithm.
50:51	WQ	0b00	<u>Write Qualifier</u> Qualifies the TLB write operation performed by tlbwe when MAS0.ATSEL = 0 (TLB). This field must be set to "00" or "11" when MAS0.ATSEL = 1 (LRAT), or an illegal instruction exception may occur for tlbwe. 00 - The selected TLB entry is written regardless of the TLB-reservation. 01 - The selected TLB entry is written if and only if the TLB reservation exists (see tlbwrx instruction). A tlbwe with this value is called a TLB Write Conditional. 10 - The TLB-reservation is cleared; no TLB entry is written. 11 - The selected TLB entry is written regardless of the TLB-reservation.
52:63	///	0x0	<u>Reserved</u>

6.18.6 MMU Assist Register 1 (MAS1)

The MAS1 register is written from a GPR using **mtspr** and can be read into a GPR using **mfspr**. This register is replicated for all processing threads. MAS1 is used by certain TLB management instructions to transfer contents to and from TLB or LRAT entries. This register provides for setting the IPROT protection bit of TLB entries. See the programming notes at the end of this section for restrictions regarding this bit.

Register Short Name:	MAS1	Read Access:	Priv
Decimal SPR Number:	625	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	V	0b0	<u>Valid</u> TLB or LRAT Valid bit: 0 - This TLB or LRAT entry is invalid 1 - This TLB or LRAT entry is valid
33	IPROT	0b0	<u>Invalidate Protect</u> Indicates this TLB entry is protected from invalidate operations due to tlbivax or tlbilx instructions, or remote invalidate snoops from other processors. IPROT is not implemented in the LRAT array entries. 0 - This TLB entry is not protected 1 - This TLB entry is protected
34:47	TID	0x0	<u>Translation Identifier</u> This TLB entry's identifier used to compare against the current value of the PID during translations, or against the MAS6.SPID value for searches.
48:49	///	0b00	<u>Reserved</u>
50	IND	0b0	<u>Indirect</u> Designates this TLB entry as an indirect entry which is used by the hardware table walker (HTW) to compute real addresses into the page table. IND is not implemented in the LRAT array entries. 0 - This TLB entry is used to directly translate virtual to real addresses 1 - This TLB entry is used by the HTW as an indirect page table pointer
51	TS	0b0	<u>Translation Space</u> This TLB entry's address space identifier used to compare against the current value of the MSR.IS or DS bit during translations, or against the MAS6.SAS value for searches. TS is not implemented in the LRAT array entries.

Bit(s):	Field Name:	Init	Description
52:55	TSIZE	0b0000	<p><u>Translation Size</u> The selected TLB entry (when MAS0.ATSEL = 0) or LRAT entry (when MAS0.ATSEL = 1) page size value.</p> <p>This implementation supports five page sizes for direct TLB entries (IND=0). All other non-specified page size encodings are treated as reserved. IND=0 direct TLB entries: 0001 = 4KB, 0011 = 64KB, 0101 = 1MB, 0111 = 16MB, 1010 = 1GB, Others = Reserved</p> <p>This implementation supports two page sizes for indirect TLB entries (IND=1). All other non-specified page size encodings are treated as reserved. IND=1 indirect TLB entries: 0101 = 1MB, 1001 = 256MB, Others = Reserved</p> <p>This implementation supports 8 page sizes for LRAT entries. All other non-specified page size encodings are treated as reserved. LRAT entries: 0101 = 1MB, 0111 = 16MB, 1001 = 256MB, 1010 = 1GB, 1011 = 4GB, 1100 = 16GB, 1110 = 256GB, 1111 = 1TB, Others = Reserved</p>
56:63	///	0x0	<u>Reserved</u>

Programming Note 1: This register provides for setting the IPROT protection bit of TLB entries. For this implementation, it is recommended that no more than two entries in any single congruence class of the TLB be written with IPROT = 1. Setting three out of four entries in any TLB congruence class can lead to two or more threads contending to use one unprotected TLB entry to service ERAT misses that map to the same congruence class. This might lead to thrashing of the single unprotected TLB entry and poor system performance. Depending on the timing of these misses, an unrecoverable livelock could occur while the hardware page table translation attempts to resolve two or more outstanding ERAT misses using one available TLB entry. Setting all four entries in a congruence class with IPROT = 1 generally leads to an eventual TLB ineligible exception.

Programming Note 2: This register provides for setting the IPROT protection bit of TLB entries. For this implementation, it is a requirement that protected entries (IPROT = 1) be placed in either way 2 or 3 of the TLB (using **tlbwe** instructions with MAS0[HES] = 0 and MAS0[ESEL] = 2 or 3) to guarantee that they will not be overwritten. This is because of the way the hardware LRU mechanism selects TLB way 0 or 1 by default for certain combinations of valid and nonvalid entries within a TLB congruence class. Protected entries in ways 0 or 1 could be erroneously overwritten by subsequent **tlbwe** instructions with MAS0[HES] = 1 or by hardware page table translation that both use the hardware LRU way selection. This does not pertain to invalidation of TLB entries (that is, protected entries will always be immune to invalidation caused by **tlbivax** and **tlbilx** instructions regardless of where they are placed within a TLB congruence class). As such, when implementing a totally software-managed TLB system, using only **tlbwe** with MAS0[HES] = 0 to install TLB entries and no hardware page table walking, this restriction can be ignored.

6.18.7 MMU Assist Register 2 (MAS2)

The MAS2 register is written from a GPR using **mtspr**, and can be read into a GPR using **mfspir**. This register is replicated for all processing threads. MAS2 is used by certain TLB management instructions to transfer contents to and from TLB or LRAT entries. The architected fields MAS2.ACM and MAS2.VLE are reserved in this implementation.

Register Short Name:	MAS2	Read Access:	Priv
Decimal SPR Number:	626	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:51	EPN	0x0	<p><u>Effective Page Number</u> For TLB entries (MAS0.ATSEL=0), this field is used to transfer the entry effective page number. Only bits associated with a page size boundary are significant. The other bits are treated as an offset within this page and ignored.</p> <p>This field is treated as a logical page number (LPN) for LRAT entries (MAS0.ATSEL=1) and used to transfer the LRAT.LPN value.</p> <p>The upper EPN(0:31) bits are instantiated in the 64-bit A2 implementation.</p>
52:58	///	0x0	<u>Reserved</u>
59	W	0b0	<p><u>Write Through</u> This page's write-through storage attribute 0 - This page is not write-through required storage 1 - This page is write-through required storage</p>
60	I	0b0	<p><u>Caching Inhibited</u> This page's caching inhibited storage attribute 0 - This page is not caching inhibited required storage 1 - This page is caching inhibited required storage</p>
61	M	0b0	<p><u>Memory Coherence Required</u> This page's memory coherence required storage attribute 0 - This page is not memory coherence required storage 1 - This page is memory coherence required storage</p>
62	G	0b0	<p><u>Guarded</u> This page's guarded storage attribute 0 - This page is not guarded storage 1 - This page is guarded storage</p>
63	E	0b0	<p><u>Endianess</u> This page's endianess storage attribute 0 - This page is accessed in big endian byte order 1 - This page is accessed in little endian byte order</p>

6.18.8 MMU Assist Register 2 Upper (MAS2U)

The MAS2U register is written from a GPR using **mtspr** and can be read into a GPR using **mfspr**. This register is replicated for all processing threads. MAS2U is used by certain 32-bit machine state (MSR[CM] = 0) TLB management instructions to transfer to and from TLB or LRAT entries.

Register Short Name:	MAS2U	Read Access:	Priv
Decimal SPR Number:	631	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	EPNU	0x0	<p><u>Effective Page Number (Upper Bits 0:31)</u> This field is an alias to MAS2.EPN(0:31) and is primarily for use by 32b machine state (CM=0) software.</p> <p>For TLB entries (MAS0.ATSEL=0), this field is to transfer the entry effective page number upper bits 0:31.</p> <p>For LRAT entries (MAS0.ATSEL=1), this field is treated as a logical page number (LPN) and used to transfer the LRAT.LPN(22:31) value.</p>

6.18.9 MMU Assist Register 3 (MAS3)

The MAS3 register is written from a GPR using **mtspr**, and can be read into a GPR using **mfspir**. This register is replicated for all processing threads. MAS3 is used by certain TLB management instructions to transfer contents to and from TLB or LRAT entries.

Register Short Name:	MAS3	Read Access:	Priv
Decimal SPR Number:	627	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:52	RPNL	0x0	<p><u>Real Page Number (Lower Bits 32:52)</u> For TLB entries (MAS0.ATSEL=0), this field is used to transfer the lsb's of the entry real page number. RPNL(52) is only significant for indirect TLB entries (IND=1), and is unused for direct TLB entries (IND=0) and LRAT entries (i.e. the TLB entry RPNL(52) bit can only be written to a '1' by a tlbwe instruction when MAS1.IND=1 and MAS0.ATSEL=0, and this bit will always be set to '0' after a tlbre instruction if the chosen TLB entry contains IND=0).</p> <p>RPNL(32:51) is treated as a TLB direct entry logical page number (LPNL) when an LRAT is present and enabled. RPNL(32:51) is treated as a real page number (RPNL) for LRAT entries (MAS0.ATSEL=1) and used to transfer the LRAT.RPN(32:51) value. Only bits associated with a particular TLB or LRAT entry page size boundary are significant. The other bits are treated as an offset within this page and ignored. The upper RPNL(22:31) bits are instantiated in the 64-bit A2 implementation in MAS7.</p>
53	///	0b0	<u>Reserved</u>
54	U0	0b0	<p><u>User Definable Storage Attribute 0</u> Specifies a system-dependent storage attribute for this TLB entry. This field is not implemented in LRAT entries. This field has no effect within the A2 core.</p>
55	U1	0b0	<p><u>User Definable Storage Attribute 1</u> Specifies a system-dependent storage attribute for this TLB entry. This field is not implemented in LRAT entries. This field has no effect within the A2 core.</p>
56	U2	0b0	<p><u>User Definable Storage Attribute 2</u> Specifies a system-dependent storage attribute for this TLB entry. This field is not implemented in LRAT entries. This field has no effect within the A2 core.</p>
57	U3	0b0	<p><u>User Definable Storage Attribute 3</u> Specifies a system-dependent storage attribute for this TLB entry. This field is not implemented in LRAT entries. This field has no effect within the A2 core.</p>
58	UX/SPSIZE0	0b0	<p><u>User Mode Execute Enable (IND=0) / SPSIZE0 (IND=1)</u> For direct TLB (IND=0) entries, specifies user mode (MSR.PR=1) execute access permission. See the appropriate access control section of this document for the definition of execute access. 0 - This page does not have execute access permission in user mode (problem state) 1 - This page has execute access permission in user mode (problem state)</p> <p>For indirect TLB (IND=1) entries, specifies sub-page size bit 0.</p>

Bit(s):	Field Name:	Init	Description
59	SX/SPSIZE1	0b0	<p><u>Supervisor Mode Execute Enable (IND=0) / SPSIZE1 (IND=1)</u> For direct TLB (IND=0) entries, specifies supervisor mode (MSR.PR=0) execute access permission. See the appropriate access control section of this document for the definition of execute access. 0 - This page does not have execute access permission in supervisor mode (privileged state) 1 - This page has execute access permission in supervisor mode (privileged state)</p> <p>For indirect TLB (IND=1) entries, specifies sub-page size bit 1.</p>
60	UW/SPSIZE2	0b0	<p><u>User Mode Execute Enable (IND=0) / SPSIZE2 (IND=1)</u> For direct TLB (IND=0) entries, specifies user mode (MSR.PR=1) write access permission. See the appropriate access control section of this document for the definition of execute access. 0 - This page does not have execute access permission in user mode (problem state) 1 - This page has execute access permission in user mode (problem state)</p> <p>For indirect TLB (IND=1) entries, specifies sub-page size bit 2.</p>
61	SW/SPSIZE3	0b0	<p><u>Supervisor Mode Write Enable (IND=0) / SPSIZE3 (IND=1)</u> For direct TLB (IND=0) entries, specifies supervisor mode (MSR.PR=0) write access permission. See the appropriate access control section of this document for the definition of write access. 0 - This page does not have write access permission in supervisor mode (privileged state) 1 - This page has write access permission in supervisor mode (privileged state)</p> <p>For indirect TLB (IND=1) entries, specifies sub-page size bit 3.</p>
62	UR/SPSIZE4	0b0	<p><u>User Mode Read Enable (IND=0) / SPSIZE4 (IND=1)</u> For direct TLB (IND=0) entries, specifies user mode (MSR.PR=1) read access permission. See the appropriate access control section of this document for the definition of read access. 0 - This page does not have read access permission in user mode (problem state) 1 - This page has read access permission in user mode (problem state)</p> <p>For indirect TLB (IND=1) entries, specifies sub-page size bit 4 (treated as reserved by A2 which implements only power of 4 x 1K sub-page sizes).</p>
63	SR/UND	0b0	<p><u>Supervisor Mode Read Enable (IND=0) / UND (IND=1)</u> For direct TLB (IND=0) entries, specifies supervisor mode (MSR.PR=0) read access permission. See the appropriate access control section of this document for the definition of read access. 0 - This page does not have read access permission in supervisor mode (privileged state) 1 - This page has read access permission in supervisor mode (privileged state)</p> <p>For indirect TLB (IND=1) entries, this bit is undefined (UND).</p>

6.18.10 MMU Assist Register 4 (MAS4)

The MAS4 register is written from a GPR using **mtspr** and can be read into a GPR using **mfspr**. This register is replicated for all processing threads. MAS4 is used by certain events to transfer default contents to other MAS registers. The architected fields MAS4.TLBSELD, MAS4.TIDSELD, MAS4.ACMD, and MAS4.VLED are reserved in this implementation because the associated nondefault fields are not implemented.

Register Short Name:	MAS4	Read Access:	Priv
Decimal SPR Number:	628	Write Access:	Priv
Initial Value:	0x00000000000000100	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:47	///	0x0	<u>Reserved</u>
48	INDD	0b0	<u>Default Indirect Value</u> Specifies the default value loaded into MAS1.IND and MAS6.SIND on a TLB miss exception.
49:51	///	0b000	<u>Reserved</u>
52:55	TSIZED	0b0001	<u>Default Translation Size Value</u> Specifies the default value loaded into MAS1.TSIZE on a TLB miss exception. If MMUCFG.TWC = 1, TSIZED is also the default value loaded into MAS6.ISIZE upon the exception.
56:58	///	0b000	<u>Reserved</u>
59	WD	0b0	<u>Default Write Through Value</u> Specifies the default value loaded into MAS2.W on a TLB miss exception.
60	ID	0b0	<u>Default Caching Inhibited Value</u> Specifies the default value loaded into MAS2.I on a TLB miss exception.
61	MD	0b0	<u>Default Memory Coherence Required Value</u> Specifies the default value loaded into MAS2.M on a TLB miss exception.
62	GD	0b0	<u>Default Guarded Value</u> Specifies the default value loaded into MAS2.G on a TLB miss exception.
63	ED	0b0	<u>Default Endianess Value</u> Specifies the default value loaded into MAS2.E on a TLB miss exception.

6.18.11 MMU Assist Register 5 (MAS5)

The MAS5 register is written from a GPR using **mtspr** and can be read into a GPR using **mfspr**. This register is replicated for all processing threads. MAS5 is used to supply hypervisor-related parameters for certain TLB management instructions.

Register Short Name:	MAS5	Read Access:	Hypv
Decimal SPR Number:	339	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	SGS	0b0	<u>Search Guest State</u> Specifies the GS value used when searching the TLB during execution of tlbisx and tlbisr. and for selecting TLB entries to be invalidated by tlbivax or tlbilx. The SGS field is compared with the TGS field of each TLB entry to find a matching entry.
33:55	///	0x0	<u>Reserved</u>
56:63	SLPID	0x0	<u>Search Logical Partition Identifier</u> Specifies the LPID value used when searching the TLB during execution of tlbisx and tlbisr. and for selecting TLB entries to be invalidated by tlbivax or tlbilx. The SLPID field is compared with the TLPID field of each TLB entry to find a matching entry.

6.18.12 MMU Assist Register 6 (MAS6)

The MAS6 register is written from a GPR using **mtspr** and can be read into a GPR using **mfspir**. This register is replicated for all processing threads. MAS6 is used to supply search and invalidate parameters for certain TLB management instructions.

Register Short Name:	MAS6	Read Access:	Priv
Decimal SPR Number:	630	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	///	0b00	<u>Reserved</u>
34:47	SPID	0x0	<u>Search Process Identifier</u> Specifies the value of PID used when searching the TLB during execution of tlbisx. It also defines the PID of the TLB entry to be invalidated by tlbilx with T=1 or T=3 and tlbivax.
48:51	///	0b0000	<u>Reserved</u>
52:55	ISIZE	0b0000	<u>Invalidate Size</u> Specifies the page size of the TLB entry to be invalidated by tlbilx T=3 or tlbivax.
56:61	///	0x0	<u>Reserved</u>
62	SIND	0b0	<u>Search Indirect</u> Specifies the value of IND used when searching the TLB during execution of tlbisx. It also defines the IND of the TLB entry to be invalidated by tlbilx T=3 and tlbivax.
63	SAS	0b0	<u>Search Address Space</u> Specifies the value of AS used when searching the TLB during execution of tlbisx. It also defines the AS of the TLB entry to be invalidated by tlbilx T=3 and tlbivax.

6.18.13 MMU Assist Register 7 (MAS7)

The MAS7 register is written from a GPR using **mtspr**, and can be read into a GPR using **mfspir**. This register is replicated for all processing threads. MAS7 is used to transfer the MSBs of the real page number to and from the TLB or LRAT entries.

Register Short Name:	MAS7	Read Access:	Priv
Decimal SPR Number:	944	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:53	///	0x0	<u>Reserved</u>
54:63	RPNU	0x0	<u>Real Page Number (Upper Bits 22:31)</u> For TLB entries (MAS0.ATSEL=0), this field is used to transfer the msb's of the entry real page number. This value is treated as a TLB entry logical page number (LPNU) when an LRAT is present and enabled. This field is treated as a real page number (RPNU) for LRAT entries (MAS0.ATSEL=1) and used to transfer the LRAT.RPN(22:31) value.

6.18.14 MMU Assist Register 8 (MAS8)

The MAS8 register is written from a GPR using **mtspr** and can be read into a GPR using **mfspr**. This register is replicated for all processing threads. MAS8 is used to transfer hypervisor-related parameters to and from the TLB entries.

Register Short Name:	MAS8	Read Access:	Hypv
Decimal SPR Number:	341	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	TGS	0b0	<u>Translation Guest Space</u> Specifies the value that is written to the TLB entry TGS bit by the execution of tlbwe. This bit is loaded from the TLB entry TGS by the execution of tlbre and by the execution of tlbxs that finds a matching entry. The TLB entry TGS identifies that value of the Guest State associated with the TLB entry.
33	VF	0b0	<u>Translation Virtualization Fault</u> Specifies the value that is written to TLB entry VF bit by the execution of tlbwe. This bit is loaded from the TLB entry VF bit by the execution of tlbre and by the execution of tlbxs that finds a matching entry. VF identifies that the TLB entry is used to provide virtualized memory mapped I/O. A data access that uses a TLB entry with the VF field equal to 1 causes a Data Storage interrupt, regardless of the settings of the permission bits. The resulting Data Storage interrupt is always directed to the embedded hypervisor state, regardless of the EHCSR.DSIGS value.
34:55	///	0x0	<u>Reserved</u>
56:63	TLPID	0x0	<u>Translation Logical Partition Identifier</u> Specifies the value that is written to the TLB entry TLPID field by the execution of tlbwe. This field is loaded from the TLB entry TLPID by the execution of tlbre and by the execution of tlbxs that finds a matching entry. The TLB entry TLPID identifies the logical partition associated with the TLB entry.

6.18.15 MAS0_MAS1 Register

The MAS0_MAS1 register is written from a 64-bit GPR using **mtspr** and can be read into a 64-bit GPR using **mf spr**. This register is replicated for all processing threads. MAS0_MAS1 is used as a 64-bit register alias for the MAS0 and MAS1 registers combined. It allows software to configure or read both the MAS0 and MAS1 with one 64-bit move to or from operation. The MAS0 and MAS1 field formats of this register are identical to those of the 32-bit versions of these registers (array and entry selection, and entry contents).

Register Short Name:	MAS0_MAS1	Read Access:	Priv
Decimal SPR Number:	373	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	MAS0	0x0	<u>MMU Assist Register 0</u> This field is an alias of MAS0(32:63)
32:63	MAS1	0x0	<u>MMU Assist Register 1</u> This field is an alias of MAS1(32:63)

6.18.16 MAS5_MAS6 Register

The MAS5_MAS6 register is written from a 64-bit GPR using **mtspr** and can be read into a 64-bit GPR using **mf spr**. This register is replicated for all processing threads. MAS5_MAS6 is used as a 64-bit register alias for the MAS5 and MAS6 registers combined. It allows software to configure or read both the MAS5 and MAS6 with one 64-bit move to or from operation. The MAS5 and MAS6 field formats of this register are identical to those of the 32-bit versions of these registers (search and invalidate operation parameters).

Register Short Name:	MAS5_MAS6	Read Access:	Hypv
Decimal SPR Number:	348	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	MAS5	0x0	<u>MMU Assist Register 5</u> This field is an alias of MAS5(32:63)
32:63	MAS6	0x0	<u>MMU Assist Register 6</u> This field is an alias of MAS6(32:63)

6.18.17 MAS7_MAS3 Register

The MAS7_MAS3 register is written from a 64-bit GPR using **mtspr** and can be read into a 64-bit GPR using **mf spr**. This register is replicated for all processing threads. MAS7_MAS3 is used as a 64-bit register alias for the MAS7 and MAS3 registers combined. It allows software to configure or read both the MAS7 and MAS3 with one 64-bit move to or from operation. The MAS7 and MAS3 field formats of this register are identical to those of the 32-bit versions of these registers (real or logical address and attributes).

Register Short Name:	MAS7_MAS3	Read Access:	Priv
Decimal SPR Number:	372	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	MAS7	0x0	<u>MMU Assist Register 7</u> This field is an alias of MAS7(32:63)
32:63	MAS3	0x0	<u>MMU Assist Register 3</u> This field is an alias of MAS3(32:63)

6.18.18 MAS8_MAS1 Register

The MAS8_MAS1 register is written from a 64-bit GPR using **mtspr** and can be read into a 64-bit GPR using **mf spr**. This register is replicated for all processing threads. MAS8_MAS1 is used as a 64-bit register alias for the MAS8 and MAS1 registers combined. It allows software to configure or read both the MAS8 and MAS1 with one 64-bit move to or from operation. The MAS8 and MAS1 field formats of this register are identical to those of the 32-bit versions of these registers (entry contents).

Register Short Name:	MAS8_MAS1	Read Access:	Hypv
Decimal SPR Number:	349	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	MAS8	0x0	<u>MMU Assist Register 8</u> This field is an alias of MAS8(32:63)
32:63	MAS1	0x0	<u>MMU Assist Register 1</u> This field is an alias of MAS1(32:63)

6.18.19 MMU Configuration Register (MMUCFG)

The MMUCFG register is a read-only register that can be read into a GPR using **mf spr**. MMUCFG is used to provide implementation-specific parameters to a guest operating system or hypervisor. The implemented format of this register follows that defined for MAV 2.0.

Register Short Name:	MMUCFG	Read Access:	Hypv
Decimal SPR Number:	1015	Write Access:	None
Initial Value:	0x000000008558341	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:35	///	0b0000	<u>Reserved</u>
36:39	LPIDSIZE	0b1000	<u>Logical Partition Identifier Size</u> Indicates the number of bits in the LPID register that are implemented by the processor. This field will always be set to "1000" for this processor (8 bits).
40:46	RASIZE	0x2a	<u>Real Address Size</u> Indicates the number of real address (RA)bits that are implemented by the processor. This field will always be set to "0101010" for this processor (42 bits).
47	LRAT	0b1	<u>Logical To Real Address Translation</u> Indicates whether the Embedded.Hypervisor.LRAT category is supported by this processor. This bit is part of the boot configuration ring for this processor. 0 = LRAT array is not supported and RPN fields are treated as real page numbers (not logical addresses). 1 = LRAT array is supported and logical address are translated to real addresses as required.
48	TWC	0b1	<u>TLB Write Conditional</u> Indicates whether the Embedded.TLB Write Conditional category is supported by this processor. This bit is part of the boot configuration ring for this processor. 0 = TLB write conditional operations and reservations are not supported. 1 = TLB write conditional operations and reservations are supported.
49:52	///	0b0000	<u>Reserved</u>
53:57	PIDSIZE	0xd	<u>Process Identifier Size</u> Indicates one less than the number of bits in the PID register that are implemented by the processor. This field will always be set to "01101" for this processor (14 bits in PID).
58:59	///	0b00	<u>Reserved</u>
60:61	NTLBS	0b00	<u>Number of TLBS</u> Indicates one less than the number of TLB structures that are implemented by this processor. This field will always be set to "00" for this processor (1 TLB).
62:63	MAVN	0b01	<u>MMU Architecture Version Number</u> Indicates the version number of the architecture of the MMU implemented by the processor. This field will always be set to "01" for this processor (Version 2.0).

6.18.20 MMU Control and Status Register 0 (MMUCSR0)

The MMUCSR0 register is written from a GPR using **mtspr** and can be read into a GPR using **mfspir**. MMUCSR0 is used to provide a register-based invalidate all function for the TLB. The implemented format for this register follows that defined for MAV 2.0. The architected fields MMUCSR0.TLBn_FI (n = 1 to 3) are reserved in this implementation.

Register Short Name:	MMUCSR0	Read Access:	Hypv
Decimal SPR Number:	1012	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:60	///	0x0	<u>Reserved</u>
61	TLB0_FI	0b0	<u>TLB 0 Full Invalidate</u> This bit controls/indicates when an invalidate all function is requested/in progress. 0 - When this bit is read as a '0', there is no invalidate all operation in progress. Writing this bit to a zero while an invalidate all operation is in progress is ignored. 1 - When this bit is read as a '1', there is an invalidate all operation in progress. Hardware will set this bit to a zero when the invalidate all operation is completed. Writing this bit to a '1' initiates the invalidate all operation (unless one is already in progress, in which case writing this bit to a '1' is ignored).
62:63	///	0b00	<u>Reserved</u>

6.18.21 TLB 0 Configuration Register (TLB0CFG)

The TLB0CFG register is a read-only register that can be read into a GPR using **mfspr**. TLB0CFG is used to provide implementation-specific parameters regarding the TLB to a guest operating system or hypervisor. The implemented format of this register follows that defined by MAV 2.0. See the engineering note at the end of this section.

Register Short Name:	TLB0CFG	Read Access:	Hypv
Decimal SPR Number:	688	Write Access:	None
Initial Value:	0x00000000407a200	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:39	ASSOC	0x4	<u>Associativity</u> Indicates the number of ways that are implemented in this processor's TLB0. This field will always be set to "00000100" for this processor (4 ways).
40:44	///	0x0	<u>Reserved</u>
45	PT	0b1	<u>Page Table</u> Indicates whether this TLB can be loaded from the hardware page table. This bit is part of the boot configuration ring for this processor. 0=TLB is not eligible to be loaded from the hardware page table (attempts to install page table entries by hardware walker will result in TLB Ineligible exceptions) 1=TLB can be loaded from the hardware page table.
46	IND	0b1	<u>Indirect</u> Indicates that an indirect entry can be created in this TLB and that there is a corresponding EPTCFG register that defines the page sizes and sub-page sizes. This bit is part of the boot configuration ring for this processor. 0=indirect entries are not supported and this TLB treats the IND bit as reserved (this infers software TLB management only). 1=indirect entries are supported (infers hardware page table walking is supported).
47	GTWE	0b1	<u>Guest TLB Write Enable</u> Indicates that a guest supervisor can write to this TLB because an LRAT array exists for this TLB. This bit is part of the boot configuration ring for this processor. 0=Guest cannot write TLB entries without hypervisor intervention 1=Guest may write TLB entries which will be translated via the LRAT.
48	IPROT	0b1	<u>Invalidate Protect</u> Indicates whether invalidation protection is implemented by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 does support the invalidate protect bit in TLB 0 entries).
49	///	0b0	<u>Reserved</u>
50	HES	0b1	<u>Hardware Entry Select</u> Indicates whether hardware entry selection is supported by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 does support hardware calculation of the entry number for TLB 0 for tlbwe instructions when MAS0.HES=1).
51	///	0b0	<u>Reserved</u>
52:63	NENTRY	0x200	<u>Number of Entries</u> Indicates the number of entries that are implemented in this processor's TLB 0. This field will always be set to "0010_0000_0000" for this processor (512 entries).

Engineering Note: The TLB0CFG[PT] and [IND] bits are both resident on the boot configuration scan chain. Therefore, it is possible to set these bits independently. For A2, because there is only one shared TLB physically resident on this processor, it is recommended that both of these bits be set to the same value (0b00 for software table walking only or 0b11 to support hardware table walking). Setting IND = 0 and PT = 1 has a similar effect as setting both bits low (that is, indirect entries are assumed to not be supported; therefore, no hardware table walking occurs). Setting IND = 1 and PT = 0 has the effect of allowing indirect entry recognition, and hardware table walking can occur. However, when the subsequent attempt to write the page table entry into the TLB occurs, the results depend on the page table entry valid bit setting. When PTE.V = 1 in this mode, it is deemed as a valid attempt to write a TLB entry and a TLB ineligible exception occurs because PT = 0. When PTE.V = 0, it is not deemed as a valid attempt to write a TLB entry; therefore, no exception occurs as the result of PT = 0. In either of these cases, no TLB entry is written.

6.18.22 TLB 0 Page Size Register (TLB0PS)

The TLB0PS register is a read-only register that can be read into a GPR using **mfspr**. TLB0PS is used to provide additional implementation-specific parameters regarding the supported TLB page sizes to a guest operating system or hypervisor. This processor supports only power of 4×1 K page sizes for the TLB, but the architecture defines this register based on power of 2×1 K page sizes.

Register Short Name:	TLB0PS	Read Access:	Hypv
Decimal SPR Number:	344	Write Access:	None
Initial Value:	0x0000000000104444	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:42	///	0x0	<u>Reserved</u>
43	PS20	0b1	<u>Page Size 20</u> Indicates whether a 2^{20} KB (1 GB) page size is supported by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 supports 1 GB page sizes for TLB 0).
44:48	///	0x0	<u>Reserved</u>
49	PS14	0b1	<u>Page Size 14</u> Indicates whether a 2^{14} KB (16 MB) page size is supported by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 supports 16 MB page sizes for TLB 0).
50:52	///	0b000	<u>Reserved</u>
53	PS10	0b1	<u>Page Size 10</u> Indicates whether a 2^{10} KB (1 MB) page size is supported by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 supports 1 MB page sizes for TLB 0).
54:56	///	0b000	<u>Reserved</u>
57	PS6	0b1	<u>Page Size 6</u> Indicates whether a 2^6 KB (64 KB) page size is supported by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 supports 64 KB page sizes for TLB 0).
58:60	///	0b000	<u>Reserved</u>
61	PS2	0b1	<u>Page Size 2</u> Indicates whether a 2^2 KB (4 KB) page size is supported by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 supports 4 KB page sizes for TLB 0).
62:63	///	0b00	<u>Reserved</u>

6.18.23 LRAT Configuration Register (LRATCFG)

The LRATCFG register is a read-only register that can be read into a GPR using **mfspr**. LRATCFG is used to provide implementation-specific parameters regarding the LRAT to the hypervisor.

Register Short Name:	LRATCFG	Read Access:	Hypv
Decimal SPR Number:	342	Write Access:	None
Initial Value:	0x00000000000542008	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:39	ASSOC	0x0	<u>Associativity</u> Indicates the number of ways that are implemented in this processor's LRAT. This field will always be set to "00000000" for this processor (fully associative LRAT).
40:46	LASIZE	0x2a	<u>Logical Address Size</u> Indicates the number of logical address (LA) bits that are implemented by this processor's LRAT. This field will always be set to "0101010" for this processor (42 bits).
47:49	///	0b000	<u>Reserved</u>
50	LPID	0b1	<u>Logical Partition ID</u> Indicates that the LPID field is supported in the LRAT entries. This bit will always be set to '1' for this processor (A2 does implement the LPID field in LRAT entries).
51	///	0b0	<u>Reserved</u>
52:63	NENTRY	0x8	<u>Number of Entries</u> Indicates the number of entries that are implemented in this processor's LRAT. This field will always be set to "0000_0000_1000" for this processor (8 entries).

6.18.24 LRAT Page Size Register (LRATPS)

The LRATPS register is a read-only register that can be read into a GPR using **mf spr**. LRATPS is used to provide additional implementation-specific parameters about the supported LRAT page sizes to the hypervisor. This processor supports only power of 4×1 K page sizes for the LRAT, but the architecture defines this register based on power of 2×1 K page sizes.

Register Short Name:	LRATPS	Read Access:	Hypv
Decimal SPR Number:	343	Write Access:	None
Initial Value:	0x0000000051544400	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	///	0b0	<u>Reserved</u>
33	PS30	0b1	<u>Page Size 30</u> Indicates whether a 2^{30} KB (1TB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 1TB page sizes for the LRAT).
34	///	0b0	<u>Reserved</u>
35	PS28	0b1	<u>Page Size 28</u> Indicates whether a 2^{28} KB (256 GB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 256 GB page sizes for the LRAT).
36:38	///	0b000	<u>Reserved</u>
39	PS24	0b1	<u>Page Size 24</u> Indicates whether a 2^{24} KB (16 GB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 16 GB page sizes for the LRAT).
40	///	0b0	<u>Reserved</u>
41	PS22	0b1	<u>Page Size 22</u> Indicates whether a 2^{22} KB (4 GB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 4 GB page sizes for the LRAT).
42	///	0b0	<u>Reserved</u>
43	PS20	0b1	<u>Page Size 20</u> Indicates whether a 2^{20} KB (1 GB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 1 GB page sizes for the LRAT).
44	///	0b0	<u>Reserved</u>
45	PS18	0b1	<u>Page Size 18</u> Indicates whether a 2^{18} KB (256 MB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 256 MB page sizes for the LRAT).
46:48	///	0b000	<u>Reserved</u>
49	PS14	0b1	<u>Page Size 14</u> Indicates whether a 2^{14} KB (16 MB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 16 MB page sizes for the LRAT).
50:52	///	0b000	<u>Reserved</u>

Bit(s):	Field Name:	Init	Description
53	PS10	0b1	<u>Page Size 10</u> Indicates whether a 2 ¹⁰ KB (1 MB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 1 MB page sizes for the LRAT).
54:63	///	0x0	<u>Reserved</u>

6.18.25 Embedded Page Table Configuration Register (EPTCFG)

The EPTCFG register is a read-only register that can be read into a GPR using **mfspr**. EPTCFG is used to provide additional implementation-specific parameters regarding the supported TLB indirect entry page sizes and sub-page sizes to the operating system. This processor supports only two combinations of page and sub-page sizes for the TLB IND = 1 entries; hence, the architected fields PS2 and SPS2 are treated as reserved in this implementation. This processor supports only power of 4×1 K page and sub-page sizes for the TLB indirect entries, but the architecture defines this register based on power of 2×1 K sizes.

Register Short Name:	EPTCFG	Read Access:	Hypv
Decimal SPR Number:	350	Write Access:	None
Initial Value:	0x00000000000091942	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:43	///	0x0	<u>Reserved</u>
44:48	PS1	0x12	<u>Page Size 1</u> Indicates whether an indirect entry with page size 2^{PS1} KB combined with sub-page size indicated by SPS1 is supported by the TLB. (A2 supports an indirect page size of 256 MB with sub-page size 64 KB)
49:53	SPS1	0x6	<u>Sub-Page Size 1</u> Indicates whether an indirect entry with sub-page size 2^{SPS1} KB combined with page size indicated by PS1 is supported by the TLB. (A2 supports an indirect page size of 256 MB with sub-page size 64 KB)
54:58	PS0	0xa	<u>Page Size 0</u> Indicates whether an indirect entry with page size 2^{PS0} KB combined with sub-page size indicated by SPS0 is supported by the TLB. (A2 supports an indirect page size of 1 MB with sub-page size 4 KB)
59:63	SPS0	0x2	<u>Sub-Page Size 0</u> Indicates whether an indirect entry with sub-page size 2^{SPS0} KB combined with page size indicated by PS0 is supported by the TLB. (A2 supports an indirect page size of 1 MB with sub-page size 4 KB)

6.18.26 Logical Page Exception Register (LPER)

The LPER register captures the logical page number and page size of a page table entry (PTE) logical-to-real translation that results in an LRAT miss exception.

Register Short Name:	LPER	Read Access:	Hypv
Decimal SPR Number:	56	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:21	///	0x0	<u>Reserved</u>
22:51	ALPN	0x0	<u>Abbreviated Logical Page Number</u> This field is used to capture the abbreviated logical page number from the PTE translation which caused an LRAT Miss exception.
52:59	///	0x0	<u>Reserved</u>
60:63	LPS	0b0000	<u>Logical Page Size</u> This field is used to capture the logical page size from the PTE translation which caused an LRAT Miss exception.

6.18.27 Logical Page Exception Register Upper (LPERU)

The LPERU register captures the most-significant bits of the logical page number of a PTE logical-to-real translation that results in an LRAT miss exception.

Note: The ALPNU field of this register is an alias for bits 22:31 of the ALPN field in the LPER register to support 32-bit accesses (that is, the same physical register bits are used as the source and destination for both LPER and LPERU registers).

Register Short Name:	LPERU	Read Access:	Hypv
Decimal SPR Number:	57	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:53	///	0x0	<u>Reserved</u>
54:63	ALPNU	0x0	<u>Abbreviated Logical Page Number (Upper Bits 22:31)</u> This field is used to capture the msb's of the abbreviated logical page number from a PTE translation which caused an LRAT Miss exception (supports 32-bit accesses).

6.18.28 MAS Register Update Summary

Table 6-15 summarizes how this implementation's MAS registers are modified by instruction TLB error interrupts, data TLB error interrupts, and the TLB management instructions.

Table 15. MAS Register Update Summary (Sheet 1 of 2)

MAS Field Updated	Value Loaded on Event			
	Data or Instruction TLB Error Interrupt ²	tlbsx hit	tlbsx miss	tlbre
MAS0 _{ATSEL}	0	0	0	—
MAS0 _{TLBSEL} ³	reserved field	reserved field	reserved field	—
MAS0 _{ESEL}	hardware hint	Way of entry that hit	hardware hint	—
MAS0 _{HES}	TLB0CFG _{HES}	TLB0CFG _{HES}	TLB0CFG _{HES}	—
MAS0 _{WQ}	0b01	0b01	0b01	—
MAS0 _{NV} ³	reserved field	reserved field	reserved field	reserved field
MAS1 _V	1	1	0	TLB _V ⁴
MAS1 _{IPROT}	0	TLB _{IPROT}	0	TLB _{IPROT}
MAS1 _{TID}	PID or EPLC _{EPID} ⁸ or EPSC _{EPID} ⁹	TLB _{TID}	MAS6 _{SPID}	TLB _{TID}
MAS1 _{IND}	MAS4 _{INDD}	TLB _{IND}	MAS4 _{INDD}	TLB _{IND}
MAS1 _{TS}	MSR _{DS} ⁸ or MSR _{IS} or EPLC _{EAS} ⁸ or EPSC _{EAS} ⁹	TLB _{TS}	MAS6 _{SAS}	TLB _{TS}
MAS1 _{TSIZE}	MAS4 _{TSIZED}	TLB _{SIZE}	MAS4 _{TSIZED}	TLB _{SIZE} ⁴
MAS2 _{EPN}	EA _{0:51} ¹	TLB _{EPN}	MAS2 _{EPN}	TLB _{EPN} ⁴
MAS2 _{ACM} ³	reserved field	reserved field	reserved field	reserved field
MAS2 _{VLE} ³	reserved field	reserved field	reserved field	reserved field
MAS2 _{WIMGE}	MAS4 _{WDIDMDGED}	TLB _{WIMGE}	MAS4 _{WDIDMDGED}	TLB _{WIMGE}
MAS3 _{RPNL[32:51]}	0	TLB _{RPNL[32:51]}	0	TLB _{RPNL[32:51]} ⁴
MAS3 _{RPNL[52]}	0	TLB _{SR} 0 ⁵	0	TLB _{SR} 0 ⁵
MAS3 _{U0:U3 UX SX UW SW UR SR}	0	TLB _{U0:U3 UX SX UW SW UR} 0 ⁶	0	TLB _{U0:U3 UX SX UW SW UR} 0 ⁶
MAS5 and MAS4	— ⁷	—	—	—
MAS6 _{SPID}	PID or EPLC _{EPID} ⁸ or EPSC _{EPID} ⁹	—	—	—
MAS6 _{ISIZE}	MAS4 _{TSIZED}	—	—	—

1. If MSR_{CM} = 0 (32-bit mode) at the time of the exception, EPN_{0:31} are set to 0.
2. If EHCSR_{DMIUH} = 0; otherwise, the interrupt is directed to the guest state.
3. These fields are not implemented and are shown for reference only.
4. The respective LRAT fields are returned when MAS0_{ATSEL} = 1.
5. MAS3_{RPNL[52]} is loaded with TLB_{SR} when TLB_{IND} = 1; otherwise, it receives '0'.
6. MAS3_{SR} is loaded with '0' when TLB_{IND} = 1; otherwise, it receives TLB_{SR}.
7. MAS5 and MAS8 are not updated on a data or instruction TLB error interrupt. The hypervisor must ensure that they already contain values appropriate to the partition.
8. EPLC values are used for data TLB error interrupts for external process ID loads (category E.PD).
9. EPSC values are used for data TLB error interrupts for external process ID stores (category E.PD).

Table 15. MAS Register Update Summary (Sheet 2 of 2)

MAS Field Updated	Value Loaded on Event			
	Data or Instruction TLB Error Interrupt ²	tlbsx hit	tlbsx miss	tlbre
MAS6 _{SAS}	MSR _{DS} or MSR _{IS} or EPLC _{EAS} ⁸ or EPSC _{EAS} ⁹	—	—	—
MAS7 _{RPNU}	0	TLB _{RPN} [22:31]	0	TLB _{RPN} [22:31] ⁴
MAS8 _{TGS VF TLPID}	— ⁷	TLB _{TGS VF TLPID}	—	TLB _{TGS VF TLPID}

1. If MSR_{CM} = 0 (32-bit mode) at the time of the exception, EPN_{0:31} are set to 0.
2. If EHCSR_{DMIUH} = 0; otherwise, the interrupt is directed to the guest state.
3. These fields are not implemented and are shown for reference only.
4. The respective LRAT fields are returned when MAS0_{ATSEL} = 1.
5. MAS3_{RPNL}[52] is loaded with TLB_{SR} when TLB_{IND} = 1; otherwise, it receives '0'.
6. MAS3_{SR} is loaded with '0' when TLB_{IND} = 1; otherwise, it receives TLB_{SR}.
7. MAS5 and MAS8 are not updated on a data or instruction TLB error interrupt. The hypervisor must ensure that they already contain values appropriate to the partition.
8. EPLC values are used for data TLB error interrupts for external process ID loads (category E.PD).
9. EPSC values are used for data TLB error interrupts for external process ID stores (category E.PD).

6.19 Storage Control Registers (Non-Architected)

This section describes the implementation-specific (nonarchitected) storage control registers.

6.19.1 Memory Management Unit Control Register 0 (MMUCR0)

The MMUCR0 register is written from a GPR using **mtspr** and can be read into a GPR using **mfspr**. In addition, the MMUCR0[TGS], [TS] and [TID] fields are updated with the TGS, TS, and TID fields of the selected ERAT entry when an **eratre** instruction is executed. Conversely, the TGS, TS, and TID fields of the selected ERAT entry are updated with the value of the MMUCR0[TGS], [TS], and [TID] fields when an **eratwe** instruction is executed. Other functions associated with fields of the MMUCR0 are described in more detail in the sections that follow.

Register Short Name:	MMUCR0	Read Access:	Hypv
Decimal SPR Number:	1020	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	ECL	0b0	<u>Extended Class</u> Used to transfer the ExtClass field of the selected ERAT entry.
33	TID_NZ	0b0	<u>Translation ID Non-zero</u> Used to transfer the TID_NZ field of the selected ERAT entry.
34	TGS	0b0	<u>Translation Guest State</u> Used to transfer the TGS bit of the selected ERAT entry.
35	TS	0b0	<u>Translation Space</u> Used to transfer the TS bit of the selected ERAT entry.
36:37	TLBSEL	0b00	<u>TLB Select</u> ERAT structure selection: 00 - Reserved 01 - Reserved 10 - I-ERAT 11 - D-ERAT
38:49	///	0x0	<u>Reserved</u>
50:63	TID	0x0	<u>Translation ID</u> Used to transfer the TID field of the selected ERAT entry

Extended Class (ECL) Field

The ECL field is used to designate the value to transfer for the ExtClass field of the ERAT entries for **eratre** and **eratwe** instructions. The ECL field serves as an auxiliary extension to the class identifier field in the ERAT entries, but is decoupled from the RS source register class field used in the ERAT management instructions. The **eratilx** and **tlbilx** and the **erativax** and **tlbivax** instructions assume an ExtClass value of zero by default. The MMUCR0[ECL] field can be used by supervisory software to create ERAT entries that are “immune” to the local or global invalidations and context synchronizing event invalidations that would normally affect all entries.

Translation ID Non-Zero (TID_NZ) Field

The MMUCR0[TID_NZ] field is used to transfer the ERAT entry's TID_NZ field on **eratre** and **eratwe** instructions that target ERAT word 0. Depending on the settings of MMUCR1[ICPID], [ITPID], [DCPID], and [DTPID] bits, the ERAT entries might contain less than the full 14 bits of the translation ID. This presents an ambiguity about whether the entry's full TID is actually zero or not (zero being the “don't care” value translation ID for matching). The ERAT compare logic uses the TID_NZ field to resolve this ambiguity.

For an **mtspr** to this register, the source register data bit 33 is not used directly as the update data for this bit. The MMUCR0[TID_NZ] bit is written to a value dependent on the data being written to the MMUCR0[TID] field (that is, new value for MMUCR0[TID_NZ] = OR_REDUCE(new value for MMUCR0[TID])). The most recently updated value of this bit is used as the new value for the ERAT entry's TID_NZ field upon completion of an **eratwe** instruction. This register bit is also updated to the chosen ERAT entry's TID_NZ field after completion of an **eratre** instruction. In this regard, this bit is treated like a read-only bit.

Translation Guest State (TGS) Field

The MMUCR0[TGS] field is used to transfer the ERAT entry's TGS field on **eratre** and **eratwe** instructions that target ERAT word 0. For instruction fetch and data storage accesses, the TGS field of the ERAT entries is compared with the MSR[GS] bit. For **eratsx[.]** however, the MMUCR0[TGS] bit is used, allowing the ERAT to be searched for entries with a TGS field that references a guest state entry while in hypervisor privilege level.

Translation Space (TS) Field

The TS field is used by the **eratsx[.]** instruction to designate the value against which the TS field of the ERAT entries is to be matched. For instruction fetch and data storage accesses, the TS field of the ERAT entries is compared with the MSR[IS] bit or the MSR[DS] bit, respectively. For **eratsx[.]** however, the MMUCR0[TS] field is used, allowing the ERAT to be searched for entries with a TS field that references an address space other than the one being used by the currently executing process.

The MMUCR0[TS] field is also used to transfer the ERAT entry's TS field on **eratre** and **eratwe** instructions that target ERAT word 0. There are two reasons for this: there are not enough bits in the GPR used for transferring the other fields so that it can hold this field as well, and this allows software to set up entries with a TS field that references an address space other than the one being used by the currently executing process.

See *Section 6.2.2 Address Space Identifier Convention* on page 171 for more information about the TS field.

Translation ID (TID) Field

The TID field is used by the **eratsx[.]** instruction to designate the process identifier value to be compared with the TID field of the ERAT entries. For instruction fetch and data storage accesses and cache management operations, the TID field of the ERAT entries is compared with the value in the PID register (see *Section 6.17.1 Process ID Register (PID)* on page 228). For **eratsx[.]**, however, the MMUCR0[TID] field is used, allowing the ERAT to be searched for entries with a TID field that does not match the process ID of the currently executing process.

The MMUCR0[TID] field is also used to transfer the ERAT entry's TID field on **eratre** and **eratwe** instructions that target ERAT word 0. There are two reasons for this: there are not enough bits in the GPR used for transferring the other fields so that it can hold this field as well, and this allows software to setup entries with a TID field that references a process identifier other than the one being used by the currently executing process.

This field is 14 bits to provide consistency with other 14-bit PID values used elsewhere on this processor. Depending on the settings of the MMUCR1[ICTID], [ITTID], [DCTID], and [DTTID] configuration bits, the 6 MSBs of this field might or might not be used directly by the ERAT structures (which can contain the 6 MSBs of the TID value in their respective Class and ThdID fields when configured for this). The 6 MSBs of this field are used to transfer the Class and/or ThdID fields of the chosen ERAT entry for **eratre** and **eratwe** instructions when these fields are configured to serve as additional TID bits via MMUCR1 settings. Likewise, the 6 MSBs of this field can be used to compare against the Class and/or the ThdID fields of the chosen ERAT entry for **eratsx[.]** instructions. See *Section 6.18.2 Memory Management Unit Control Register 1 (MMUCR1)* for a description of how these bits can be configured for 14-bit ERAT TID operation.

The entire 14 bits of this field are always used in the determination of the value of the TID_NZ bit when this register is written.

See *Section 6.2.4 TLB Match Process* on page 173 for more information about the TID field and the address matching process. Also see *Section 6.10.1 ERAT Read and Write Instructions (eratre and eratwe)* on page 203 for more information about how the MMUCR0[TID] field is used by these instructions.

6.19.2 Memory Management Unit Control Register 1 (MMUCR1)

The MMUCR1 register is written from a GPR using **mtspr** and can be read into a GPR using **mfspr**. This register is shared between all processing threads. Therefore, software locking is recommended to access this register.

The MMUCR1[IRRE] and [DRRE] bits separately enable the instruction or data ERAT LRU round-robin (that is, atomically increment mod watermark) mode of operation. The MMUCR1[PEI] bits are used to enable parity error injection when the TLB or ERATs are written using the **tlbwe** or **eratwe** instructions. This is used for parity error exception handler testing as described in *Section 6.13.2 Simulating TLB and ERAT Parity Errors for Software Testing*. In addition, the MMUCR1[IEEN], [DEEN], or [TEEN] error entry number field is updated with the entry number of the TLB or ERAT entry when a parity or multi-hit error occurs. Other functions associated with the fields of the MMUCR1 are described in more detail in the sections that follow.

Register Short Name:	MMUCR1	Read Access:	Hypv
Decimal SPR Number:	1021	Write Access:	Hypv
Initial Value:	0x00000000c000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32	IRRE	0b0	<u>I-ERAT LRU Round-Robin Enable</u> 0 - LRU normal operation 1 - LRU atomically increments upon eratwe
33	DRRE	0b0	<u>D-ERAT LRU Round-Robin Enable</u> 0 - LRU normal operation 1 - LRU atomically increments upon eratwe
34	REE	0b0	<u>Reference Exception Enable</u> 0 - Not Enabled 1 - Translation hit with R bit cleared generates Instruction Storage Interrupt or Data Storage Interrupt
35	CEE	0b0	<u>Change Exception Enable</u> 0 - Not Enabled 1 - Translation hit with C bit cleared generates Instruction Storage Interrupt or Data Storage Interrupt
36:37	CSINV	0b11	<u>Context Sync Invalidate</u> This field controls how certain ERAT context affecting instructions affect the invalidation of non-protected (EXTCLASS=0) I-ERAT and D-ERAT entries. See the CSINV field description below for a definition of the set of ERAT context affecting instructions. Bit 36: 0 - ERAT context affecting instructions other than isync will invalidate non-protected ERAT entries (enabled) 1 - ERAT context affecting instructions other than isync do not invalidate ERAT entries (disabled) Bit 37: 0 - The isync instruction will invalidate non-protected ERAT entries (enabled) 1 - The isync instruction does not invalidate ERAT entries (disabled)

Bit(s):	Field Name:	Init	Description
38:43	PEI	0x0	<u>Parity Error Inject</u> Parity Error Inject Bits: 0=Normal Parity Calculation; 1=Invert Parity (when writing) 38 - I-ERAT WS=0 Parity Error Inject 39 - I-ERAT WS=1 Parity Error Inject 40 - D-ERAT WS=0 Parity Error Inject 41 - D-ERAT WS=1 Parity Error Inject 42 - TLB Parity Error Inject 43 - TLB LRU Parity Error Inject
44	ICTID	0b0	<u>I-ERAT Class Translation ID Enable</u> 0 - I-ERAT Class field operates as a class ID 1 - I-ERAT Class field operates as TID(0:1) bits (of TID(0:13) total value).
45	ITTID	0b0	<u>I-ERAT ThdID Translation ID Enable</u> 0 - I-ERAT ThdID field operates as a thread ID 1 - I-ERAT ThdID field operates as TID(2:5) bits (of TID(0:13) total value).
46	DCTID	0b0	<u>D-ERAT Class Translation ID Enable</u> 0 - D-ERAT Class field operates as a class ID 1 - D-ERAT Class field operates as TID(0:1) bits (of TID(0:13) total value).
47	DTTID	0b0	<u>D-ERAT ThdID Translation ID Enable</u> 0 - D-ERAT ThdID field operates as a thread ID 1 - D-ERAT ThdID field operates as TID(2:5) bits (of TID(0:13) total value).
48	DCCD	0b0	<u>D-ERAT Class Compare Disable</u> 0 - D-ERAT Class field is used for normal and external PID translation compares. 1 - D-ERAT Class field is ignored for translation compares (mutually exclusive to using external PID ops).
49	TLBWE_BINV	0b0	<u>TLBWE Back Invalidate</u> 0 - No back invalidates are generated to the ERAT's for tlbwe instructions. 1 - When tlbwe with MAS0[HES]=0 instruction overwrites a valid, direct TLB entry without an exception being generated, send a back invalidate to the ERAT's targetting the old virtual address.
50	TLBI_MSB	0b0	<u>TLB Invalidate Most Significant Bit</u> 0 - TLB invalidate snoops from bus assume EPN[31:51] is significant (EPN[27:30] is ignored). 1 - TLB invalidate snoops from bus assume EPN[27:51] is significant.
51	TLBI_REJ	0b0	<u>TLB Invalidate Reject</u> 0 - TLB invalidate snoops from bus are accepted and compared against LPID values in the TLB. 1 - TLB invalidate snoops from bus compare against LPIDR.LPID value for acceptance or rejection.
52	IERRDET	0b0	<u>I-ERAT Error Detect</u> 0 - No Error Detected 1 - I-ERAT error detected and EEN field contains a snapshot of first entry number with error detected
53	DERRDET	0b0	<u>D-ERAT Error Detect</u> 0 - No Error Detected 1 - D-ERAT error detected and EEN field contains a snapshot of first entry number with error detected
54	TERRDET	0b0	<u>TLB Error Detect</u> 0 - No Error Detected 1 - TLB error detected and EEN field contains a snapshot of first entry number with error detected

Bit(s):	Field Name:	Init	Description
55:63	EEN	0x0	<u>Error Entry Number</u> I-ERAT, D-ERAT, or TLB entry number for which first error was found after a read of this register

I-ERAT LRU Round-Robin Enable (IRRE) Bit

The MMUCR1[IRRE] bit is used to enable the I-ERAT LRU round-robin mode of operation. See *Section 6.7.4 ERAT LRU Round-Robin Replacement Mode* for a description of this behavior.

D-ERAT LRU Round-Robin Enable (DRRE) Bit

The MMUCR1[DRRE] bit is used to enable the D-ERAT LRU round-robin mode of operation. See *Section 6.7.4 ERAT LRU Round-Robin Replacement Mode* for a description of this behavior.

Reference Exception Enable (REE) Bit

The MMUCR1[REE] bit is used to enable the generation of a read access control DSI exception or an execute access control ISI exception, when a matching TLB or ERAT entry with R = 0 is accessed via a load or instruction fetch. See *Section 6.12 Page Reference and Change Status Management* for a description of this behavior.

Change Exception Enable (CEE) Bit

The MMUCR1[CEE] bit is used to enable the generation of a write access control DSI exception when a matching TLB or ERAT entry is accessed with C = 0 via a store. See *Section 6.12 Page Reference and Change Status Management* for a description of this behavior.

Context Sync Invalidate (CSINV) Field

The MMUCR1[CSINV] field is used to control the invalidation of nonprotected (ExtClass = 0) entries in the I-ERAT and D-ERAT as the result of executing certain ERAT context-altering instructions in MMU mode (CCR2[NOTLB] = 0). This set of instructions includes: **sc**, **ehpriv**, **mtmsr**, **mtpid**, **mtlpidr**, **rfi**, **rfdi**, **rfmci**, **rfgi**, and **isync**. In ERAT-only mode (CCR2[NOTLB] = 1), the ERAT entries are not invalidated as the result of these instructions, and this field is effectively disabled. This field has no impact on the TLB and LRAT structures in this implementation, nor does it affect the operation of any invalidate instructions. Bit 36 of this field impacts the non-**isync** instructions of this set, and bit 37 controls the behavior of **isync**. Setting this field to “00” allows the entire defined set of ERAT context-altering instructions to flush nonprotected entries from the ERATs. Setting this field to “01” allows the defined set of ERAT context-altering instructions other than **isync** to flush nonprotected entries from the ERATs. Setting this field to “10” allows the non-**isync** instructions to complete without invalidating ERAT entries, but the **isync** instruction will flush the ERAT nonprotected entries. Setting this field to “11” effectively immunizes the ERAT nonprotected entries from invalidation as the result of any of these context-altering instructions.

Parity Error Inject (PEI) Field

The MMUCR1[PEI] field is used to inject parity errors into the I-ERAT, D-ERAT, and/or the TLB entry targeted by a subsequent **eratwe** or **tlbwe** instruction. One bit is provided for each array and word select combination to individually test the parity error logic of the tag and data portion of each ERAT structure, or the separate

physical TLB and LRU arrays, and the resulting software handling of each error. The specific ERAT word or TLB congruence class and way chosen for error injection is dependent on the parameters associated with the **eratwe** and **tlbwe** instructions.

I-ERAT Class Translation ID (ICTID) Field

The MMUCR1[ICTID] field is used to control the behavior of the I-ERAT entries Class field. When this bit is zero, the Class field of the I-ERAT entries behaves normally as a Class value as described in other sections of this document (that is, it responds to class-based invalidations, and so forth). The I-ERAT Class field is loaded from RS[52:53] upon execution of the **eratwe** with WS = 0 when this bit is zero.

When this bit is set to a '1', the I-ERAT logic treats the Class field of each entry as 2 additional bits of the translation ID (TID). In this mode, the 2-bit Class field is used as TID[0:1] of the full TID[0:13] value (that is, the 2 MSBs of the 14-bit TID). The I-ERAT Class field is compared against the PID[50:51] value for translations. The I-ERAT Class field is loaded from MMUCR0[50:51] upon execution of the **eratwe** with WS = 0. MMUCR0[50:51] are used to compare against the I-ERAT Class field for **eratsx[.]** instructions when this bit is set. Class-based invalidations of the I-ERAT in this mode are ignored. It is recommended that supervisory software only set this bit when the MMUCR1[ITTID] bit is also set (providing for the full 14-bit PID compare) to avoid a noncontiguous TID value being used in the I-ERAT compare logic.

I-ERAT ThdID Translation ID (ITTID) Field

The MMUCR1[ITTID] field is used to control the behavior of the I-ERAT entries ThdID field. When this bit is zero, the ThdID field of the I-ERAT entries behave normally as a thread identifier value as described in other sections of this document (that is, compares ThdID against thread valid for the fetch, and so forth). The I-ERAT ThdID field is loaded from RS[60:61] upon execution of the **eratwe** with WS = 0 when this bit is zero.

When this bit is set to a '1', the I-ERAT logic treats the ThdID field of each entry as 4 additional bits of the TID. In this mode, the 2-bit ThdID field is used as TID(2:3) of the full TID(0:13) value (that is, the 4 MSBs of a truncated 12-bit TID value). The I-ERAT ThdID field is loaded from MMUCR0[52:53] upon execution of the **eratwe** with WS = 0. MMUCR0[52:55] are used to compare against the I-ERAT ThdID field for **eratsx[.]** instructions when this bit is set. The full 14-bit TID value can be realized by setting both this bit and the ICTID bit of this register.

Setting the ITTID bit disables any thread-based assignment of translation entries to a specific hardware thread or groups of threads (that is, thread ID is not a factor in determining hit or miss for any entry). This function is provided for operating systems that might realize a performance benefit from housing a larger subset (greater than 8 bits) of the 14-bit TID value in I-ERAT entries, or which would otherwise always set ThdID = '11' for all entries.

D-ERAT Class Translation ID (DCTID) Field

The MMUCR1[DCTID] field is used to control the behavior of the D-ERAT entries Class field. When this bit is zero, the Class field of the D-ERAT entries behaves normally as a Class value as described in other sections of this document (that is, responds to class-based invalidations, and so forth). The D-ERAT Class field is loaded from RS[52:53] upon execution of the **eratwe** with WS = 0 when this bit is zero.

When this bit is set to a '1', the D-ERAT logic treats the Class field of each entry as 2 additional bits of the TID. In this mode, the 2-bit Class field is used as TID[0:1] of the full TID[0:13] value (that is, the 2 MSBs of the 14-bit TID). The D-ERAT Class field is compared against the PID[50:51] value for translations. The D-ERAT Class field is loaded from MMUCR0[50:51] upon execution of the **eratwe** with WS = 0. MMUCR0[50:51] are

used to compare against the D-ERAT Class field for **eratsx[.]** instructions when this bit is set. Class-based invalidations of the D-ERAT in this mode are ignored. It is recommended that supervisory software only set this bit when the MMUCR1[DTTID] bit is also set (providing for the full 14-bit PID compare) to avoid a noncontiguous TID value being used in the D-ERAT compare logic.

Note: Care must be exercised when overriding the function of the Class field in the D-ERAT. In addition to normal load, store, and cache management operations, the D-ERAT structure is used for translation of external PID load and store instructions that inherently override, among other things, the PID and LPID values. Because the D-ERAT entries are not tagged with the LPID value, ambiguities might occur with multiple D-ERAT entries with the same (or similar) TID values, but that pertain to different logical partitions (that is, the TLPID value in the TLB is different for these entries, and the LPID values in the LPIDR, EPLC, and EPSC registers are unique). The D-ERAT Class values of 2 and 3 are normally used by hardware to denote external load and external store associated entries respectively to overcome this LPID ambiguity. It is the responsibility of the operating system and/or hypervisor to resolve this external PID operation ambiguity when DCTID = '1'. Some ways of avoiding such ambiguities include simply not using external loads and stores that target different LPID values or maintaining a unique set of process IDs assigned to each logical partition ID currently referenced by this processor, and so forth.

D-ERAT ThdID Translation ID (DTTID) Field

The MMUCR1[DTTID] field is used to control the behavior of the D-ERAT entries ThdID field. When this bit is zero, the ThdID field of the D-ERAT entries behaves normally as a thread identifier value as described in other sections of this document (that is, it compares ThdID against thread valid for loads, stores, and cache management operations, and so forth). The D-ERAT ThdID field is loaded from RS[60:61] upon execution of the **eratwe** with WS = 0 when this bit is zero.

When this bit is set to a '1', the D-ERAT logic treats the ThdID field of each entry as 2 additional bits of the TID. In this mode, the 2-bit ThdID field is used as TID(2:3) of the full TID(0:13) value (that is, the 2 MSBs of a truncated 12-bit TID value). The D-ERAT ThdID field is loaded from MMUCR0[52:53] upon execution of the **eratwe** with WS = 0. MMUCR0[52:53] are used to compare against the D-ERAT ThdID field for **eratsx[.]** instructions when this bit is set. The full 14-bit TID value can be realized by setting both this bit and the DCTID bit of this register.

Setting the DTTID bit disables any thread-based assignment of translation entries to a specific hardware thread or groups of threads (that is, the thread ID is not a factor in determining hit or miss for any entry). This function is provided for operating systems that might realize a performance benefit from housing a larger subset (greater than 8 bits) of the 14-bit TID value in I-ERAT entries, or which would otherwise always set ThdID = '11' for all entries.

D-ERAT Class Compare Disable (DCCD) Bit

The MMUCR1[DCCD] field is used to control the behavior of the D-ERAT entries Class field compare enable. When this bit is zero, the Class field of the D-ERAT entries behaves normally as described in other sections of this document (that is, it responds to class-based invalidations, or can be part of the TID value, and so forth). When this bit is set, the D-ERAT Class field is ignored for normal load/store translations and for external PID load/store translations. This puts the D-ERAT into a mode of operation similar to that of the I-ERAT in which the Class field is used only for invalidation compares (caused by invalidation instructions or context-altering operations, and so forth) and is ignored for translation activity.

This bit also impacts the value of the Class field data that is written into a D-ERAT entry as the result of a unified TLB entry reload. See *Table 6-5 ERAT Class Field Reload Value For UTLB Hits* for details on how this bit affects the Class field for UTLB reloads. This bit has no direct impact on D-ERAT invalidation operations of any kind.

TLB Write Entry Back Invalidate Enable (TLBWE_BINV) Bit

The MMUCR1[TLBWE_BINV] is used to enable ERAT shadow-copy back invalidation when a valid TLB entry is being overwritten by a **tlbwe** instruction. When this bit is zero, no **tlbwe**-sourced back invalidates to the ERATs are allowed. When this bit is set, virtual-address-based back invalidates are sent to both ERAT structures as the result of certain **tlbwe** operations. See *Section 6.7.6 ERAT (TLB Lookaside Information) Coherency and Back-Invalidation* for the exact **tlbwe** instruction parameters that can cause a back invalidation when this bit is set and the ERAT entry matching parameters for this back invalidation.

TLB Invalidate Most Significant Bit (TLBI_MSB) Bit

The MMUCR1[TLBI_MSB] bit is used to determine the behavior of **tlbivax** back-invalidate snoop handling of the EPN field by the TLB invalidation hardware. This bit is an override for the invalidation snoop handling logic to behave as though it were placed in a system that supports the full EPN[27:51] width of the A2 to L2 request bus interface EPN definition (PBus Category B.E supports only EPN[31:51]). This bit also controls the format of the A2 core's downbound **tlbivax** and **erativax** request EPN field. See *Section 6.9.4 TLB Invalidate Virtual Address (Indexed) Instruction (tlbivax)* and *Section 6.10.3 ERAT Invalidate Virtual Address (Indexed) Instruction (erativax)* for a detailed descriptions of this format.

TLB Invalidate Reject (TLBI_REJ) Bit

The MMUCR1[TLBI_REJ] bit is used to determine the behavior of **tlbivax** back-invalidate snoop handling by the TLB invalidation hardware. This bit is an override for the invalidation snoop handling logic to behave as though it were placed in ERAT-only mode (even though there is a TLB resident and the hardware is otherwise operating in MMU mode).

When this bit is zero, the invalidation hardware assumes all **tlbivax** snoops are intended for this processor, and accepts and processes all snoops. The snoop LPID value is used in the TLB invalidation compare to determine if an entry should be invalidated. There is no snoop "filtering" based on the LPIDR[LPID] value. All invalidation snoops are completed to the bus by sending a downbound TLBI_COMPLETE transaction.

When this bit is set to a '1', the invalidation hardware assumes **tlbivax** snoops are only intended for this processor when the snoop LPID value matches this processor's LPIDR[LPID] value. If there is a mismatch, a snoop reject is delivered immediately to the bus. Otherwise, the matching invalidation snoop will be completed to the bus by sending a downbound TLBI_COMPLETE transaction.

I-ERAT Error Detect (IERRDET) Bit

The MMUCR1[IERRDET] bit is set to a '1' by hardware when the I-ERAT detects a multi-hit error or parity error, and the current values of the IERRDET, DERRDET, and TERRDET bits are all zero. A read of this register returns the current state of this bit, clears this bit, and re-enables the capture property of this bit and that of the EEN field. When this bit (or any of the other ERRDET bits) is set to a '1', the EEN field retains its current state, as do all of the ERRDET bits.

D-ERAT Error Detect (DERRDET) Bit

The MMUCR1[DERRDET] bit is set to a '1' by hardware when the D-ERAT detects a multi-hit error or parity error, and the current values of the IERRDET, DERRDET, and TERRDET bits are all zero. A read of this register returns the current state of this bit, clears this bit, and re-enables the capture property of this bit and that of the EEN field. When this bit (or any of the other ERRDET bits) is set to a '1', the EEN field retains its current state, as do all of the ERRDET bits.

TLB Error Detect (TERRDET) Bit

The MMUCR1[TERRDET] bit is set to a '1' by hardware when the TLB detects a multi-hit error or parity error, and the current values of the IERRDET, DERRDET, and TERRDET bits are all zero. A read of this register returns the current state of this bit, clears this bit, and re-enables the capture property of this bit and that of the EEN field. When this bit (or any of the other ERRDET bits) is set to a '1', the EEN field retains its current state, as do all of the ERRDET bits.

Error Entry Number (EEN) Field

The MMUCR1[EEN] field is used to capture the I-ERAT, D-ERAT, or TLB entry number that corresponds to the first multi-hit error or parity error occurrence after this register is read by software. A read of this register returns the current state of this field, clears this field, and re-enables the capture mode for this field, as described above for the ERRDET bits. This is provided for software parity and/or multi-hit error handling and debugging assistance.

Engineering Note: A multi-hit error or parity error that is detected by the I-ERAT, D-ERAT, or TLB will set the appropriate MMUCR1[IERRDET], MMUCR1[DERRDET], or MMUCR1[TERRDET] bit as described above, and the MMUCR1[EEN] field, regardless of the state of the MSR[ME] or [GS] bits. This means that an error will be speculatively logged in MMUCR1 regardless of whether or not a machine check interrupt is taken. In other words, these debug bits capture the state of multi-hit and parity error exceptions, not interrupts. This allows for a broader range of multi-hit and parity error debugging capability. The update of the error detect fields of this register are also gated by the XUCR4[MMU_MCHK]=0 and CCR2[NOTLB]=0 flash invalidate mode. When both of these bits are low, a multi-hit or parity error in the ERATs and/or TLB is handled as a miss, rather than an exception, and no update to the MMUCR1 error detection fields are done. Refer to *Section 6.13 TLB and ERAT Parity Operations* for further explanation of this mode.

6.19.3 Memory Management Unit Control Register 2 (MMUCR2)

The MMUCR2 register is written from a GPR using **mtspr** and can be read into a GPR using **mf spr**. This register is shared between all processing threads. Therefore, software locking is recommended to access this register.

The MMUCR2[PSn] fields are used to control the MMU TLB page size sequencer hardware. The TLB is probed multiple times for the supported page sizes during ERAT miss servicing. The page size is used in the calculation of the congruence class selection in the TLB array. The MMUCR2 register provides a means to configure how many of the page sizes should be used (that is, software can choose to use less than the maximum number of supported page sizes) and in which order the page sizes should be applied in TLB congruence class calculation (that is, one page size is used more often than others, therefore check it first).

Setting a PSn field to '0000' disables checking for that page size probe. The PSn fields of this register should be used in a strictly monotonic fashion, meaning software needs to PS0 first (that is, set it to a valid nonzero value), then PS1, and so forth, because page size probing ceases for the first zeroed PSn field encountered.

Note: This register has no impact on TLB probing for indirect (IND = 1) entries for hardware page table walking. Indirect entries are always searched for in order; that is, indirect page size = 1 MB, followed by indirect page size = 256 MB.

The MMUCR2[EXT_DBG_SEL] bits are used to provide extended (or alternate) event selection for certain debug trace and trigger groups. The MMUCR2[CLKG_CTL] bits are clock gating override bits that, when set, provide for an override of the normal power clock gating provided in certain parts of the MMU hardware. These bits have no noticeable impact to software or mainline MMU hardware functions.

Register Short Name:	MMUCR2	Read Access:	Hypv
Decimal SPR Number:	1022	Write Access:	Hypv
Initial Value:	0x000000000000a7531	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:39	CLKG_CTL	0x0	MMU Clock Gating Control Power clock gating overrides for various parts of the MMU. Setting these bits has no functional impact.
40:43	EXT_DBG_SEL	0b0000	MMU Extended Debug Select Alternate debug group selects for the MMU. Refer to the MMU unit debug group and trigger selects section of this document. Bit 40: Alternate debug groups 10 and 11 select Bit 41: Alternate debug groups 12 and 13 select Bit 42: Alternate debug groups 14 and 15 select Bit 43: Alternate debug trigger group 3
44:47	PS4	0b1010	TLB Page Size 4 Select 0000 - Disabled (do not apply the hash for this page size) 0001 - Page size = 4KB 0011 - Page size = 64KB 0101 - Page size = 1MB 0111 - Page size = 16MB 1010 - Page size = 1GB Others - Reserved

Bit(s):	Field Name:	Init	Description
48:51	PS3	0b0111	TLB Page Size 3 Select 0000 - Disabled (do not apply the hash for this page size) 0001 - Page size = 4KB 0011 - Page size = 64KB 0101 - Page size = 1MB 0111 - Page size = 16MB 1010 - Page size = 1GB Others - Reserved
52:55	PS2	0b0101	TLB Page Size 2 Select 0000 - Disabled (do not apply the hash for this page size) 0001 - Page size = 4KB 0011 - Page size = 64KB 0101 - Page size = 1MB 0111 - Page size = 16MB 1010 - Page size = 1GB Others - Reserved
56:59	PS1	0b0011	TLB Page Size 1 Select 0000 - Disabled (do not apply the hash for this page size) 0001 - Page size = 4KB 0011 - Page size = 64KB 0101 - Page size = 1MB 0111 - Page size = 16MB 1010 - Page size = 1GB Others - Reserved
60:63	PS0	0b0001	TLB Page Size 0 Select 0000 - Disabled (do not apply the hash for this page size) 0001 - Page size = 4KB 0011 - Page size = 64KB 0101 - Page size = 1MB 0111 - Page size = 16MB 1010 - Page size = 1GB Others - Reserved

Page Size 0 (PS0) Field

The MMUCR2[PS0] field is used to select which page size should be used first in the congruence class calculation for multiple probes of the TLB. Setting this field to '0000' disables probing of the TLB for this page size (because this is first field used by the hardware TLB probe sequencer, setting this field to all zeros guarantees a TLB miss to occur).

Page Size 1 (PS1) Field

The MMUCR2[PS1] field is used to select which page size should be used second in the congruence class calculation for multiple probes of the TLB. Setting this field to '0000' disables probing of the TLB for this page size (that is, only one page size probe for PS0 occurs).

Page Size 2 (PS2) Field

The MMUCR2[PS2] field is used to select which page size should be used third in the congruence class calculation for multiple probes of the TLB. Setting this field to '0000' disables probing of the TLB for this page size (that is, only two page size probes for PS0 and PS1 occur).

Page Size 3 (PS3) Field

The MMUCR2[PS3] field is used to select which page size should be used fourth in the congruence class calculation for multiple probes of the TLB. Setting this field to '0000' disables probing of the TLB for this page size (that is, only three page size probes for PS0 through PS2 occur).

Page Size 4 (PS4) Field

The MMUCR2[PS4] field is used to select which page size should be used fifth in the congruence class calculation for multiple probes of the TLB. Setting this field to '0000' disables probing of the TLB for this page size (that is, only four page size probes for PS0 through PS3 occur).

6.19.4 Memory Management Unit Control Register 3 (MMUCR3)

The MMUCR3 register is written from a GPR using **mtspr** and can be read into a GPR using **mfspr**. This register is replicated for each thread. MMUCR3 is used to transfer implementation-specific fields of the selected TLB entry when a **tlbre** or **tlbwe** instruction is executed or when a **tlbsx[.]** instruction is executed and a matching entry is found.

Register Short Name:	MMUCR3	Read Access:	Priv
Decimal SPR Number:	1023	Write Access:	Priv
Initial Value:	0x000000000000000f	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:48	///	0x0	<u>Reserved</u>
49	X	0b0	<u>Exclusion Range Enable</u> This bit is used to transfer the X bit to/from the selected TLB entry.
50	R	0b0	<u>Reference</u> This bit is used to transfer the R bit to/from the selected TLB entry.
51	C	0b0	<u>Change</u> This bit is used to transfer the C bit to/from the selected TLB entry.
52	ECL	0b0	<u>Extended Class</u> This field is used to transfer the extended class field to/from the selected TLB entry.
53	TID_NZ	0b0	<u>Translation ID Non-zero</u> This field is used to transfer the TID_NZ field from the selected TLB entry.
54:55	Class	0b00	<u>Class</u> This field is used to transfer the Class field to/from the selected TLB entry.
56:57	WLC	0b00	<u>L1 D-Cache Way Locking Class Attribute</u> This field is used to transfer the WLC bits to/from the selected TLB entry.
58	ResvAttr	0b0	<u>Reserved Attributes</u> This field is used to transfer the reserved attributes to/from the selected TLB entry.
59	///	0b0	<u>Reserved</u>
60:63	ThdID	0b1111	<u>Thread Identifier</u> This field is used to transfer the thread ID field to/from the selected TLB entry.

Extended Class (ECL) Field

The ECL field is used to designate the value to transfer for the ExtClass field of the TLB entries for **tlbre** and **tlbwe** instructions. The ECL field serves as an auxiliary extension to the class identifier field in the TLB entries, but decoupled from the class field used in certain management invalidation instructions. The **eratlix** and **tlbilx** and the **erativax** and **tlbivax** instructions assume an ExtClass value of '0' by default. The MMUCR3[ECL] field can be used by supervisory software to create shadow ERAT entries that are “immune” to the local or global invalidations. The TLB[IPROT] bit is used exclusively to determine a TLB entry’s protection status (that is, the TLB[ExtClass] field is not factored into a TLB entry’s invalidation protection).

The setting of the MAS1[IPROT] field controls how this field is used when writing TLB entries. When TLB entries are created via **tlbwe** instructions while MAS1[IPROT] = 0, this field is ignored, and the ExtClass field of the TLB entry is set to "0". This is to prevent TLB entries intended to be volatile (that is, not protected) from creating subsequent shadow copies that are protected. When TLB entries are created via **tlbwe** instructions while MAS1[IPROT] = 1, this field is used to define the ExtClass field of the TLB entry and the subsequent value of any shadow copies of this entry that might be created in the ERATs. This mechanism allows for protecting TLB entries with IPROT = 1, but allowing subsequent shadow copies to be volatile (ECL = 0) or to also be protected (ECL = 1). When a shadow copy ERAT entry is installed from the TLB, the resulting ExtClass field of the shadow ERAT entry (hence its protection status) is a logical AND between the TLB entry IPROT bit and this field (that is, ERAT[ExtClass] = TLB.IPROT AND MMUCR3[ECL] after TLB reload). Setting this bit before setting up TLB entries with the **tlbwe** instructions provides for TLB shadow copies in the ERATs that always have their respective ExtClass field mimic the setting of the backing TLB entry's IPROT setting.

Translation ID Non-Zero (TID_NZ) Field

The TID_NZ field is used to designate the value to transfer of the TID_NZ field of the TLB entries for **tlbre**. The update data value for the TID_NZ field for **tlbwe** instructions is calculated from the MAS1[TID] field (the TID_NZ field is updated to the value of the logical OR of the MAS1 14 TID bits for **tlbwe**).

The X, R, C, Class, WLC, ResvAttr, and ThdID fields of this register are simply used to transfer these implementation-specific fields to and from the TLB upon **tlbwe** or **tlbre** execution; they require no additional description.

7. CPU Interrupts and Exceptions

This chapter begins by defining the terminology and classification of interrupts and exceptions in *Overview* and *Directed Interrupts*.

Interrupt Processing on page 281 explains in general how interrupts are processed, including the requirements for partial execution of instructions.

Several registers support interrupt handling and control. *Interrupt Processing Registers* on page 284 describes these registers.

Table 7-3 Interrupt and Exception Types on page 306 lists the interrupts and exceptions handled by the A2O core, in the order of interrupt offset. Detailed descriptions of each interrupt type follow, in the same order.

Finally, *Interrupt Ordering and Masking* on page 346 and *Exception Priorities* on page 349 define the priority order for the processing of simultaneous interrupts and exceptions.

7.1 Overview

An *interrupt* is the action in which the processor saves its old context (Machine State Register (MSR) and next instruction address) and begins execution at a predetermined interrupt-handler address, with a modified MSR. The term processor in this context is a single hardware thread on the A2 core. An interrupt on one thread does not affect the execution of another thread. *Exceptions* are the events that can cause the processor to take an interrupt, if the corresponding interrupt type is enabled.

Exceptions can be generated by the execution of instructions or by signals from devices external to the A2O Core, the internal timer facilities, debug events, or error conditions.

A *hypervisor* program is a layer of trusted software that manages other software running on different local collections of real storage, which are called *partitions*. The collection of software that runs in a given partition and its associated resources is called a *guest*. The guest normally includes an operating system (or other system software) running in privileged state and its associated processes running in the problem state under the management of the hypervisor. The processor is in the *guest state* when a guest is executing, and it is in the *hypervisor state* when the hypervisor is executing. The processor is executing in the guest state when MSR[GS] = 1. The processor is executing in the hypervisor state when MSR[GS,PR] = 0b00. An instruction or register that is hypervisor privileged must be in the hypervisor state to successfully execute. If executed from guest privileged state (MSR[GS,PR] = 0b10), an embedded hypervisor privilege exception occurs.

7.2 Directed Interrupts

Interrupts can be *directed* to either the guest 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 registers are used to capture the processor state at the time of the interrupt, and which registers are used to post exception status.

Interrupts directed to the embedded hypervisor state use the Interrupt Vector Prefix Register (IVPR) for the upper 48 bits of the address and Interrupt Fixed Offsets to provide the lower 16. Interrupts that are directed to the embedded hypervisor state use SRR0 - Save/Restore Register 0 and Save/Restore Register 1 (SRR1) to save the context at interrupt time; they use the ESR - Exception Syndrome Register to post exception

syndrome information and the Data Exception Address Register (DEAR) to post the effective address of a data reference. Doorbell interrupts are directed to embedded hypervisor state, but use Guest Save/Restore Register 0 (GSRR0) and Guest Save/Restore Register 1 (GSRR1) to save context.

Interrupts directed to the guest state use the Guest Interrupt Vector Prefix Register (GIVPR) for the upper 48 bits of the address and Interrupt Fixed Offsets to provide the lower 16. Interrupts that are directed to the guest state use Guest Save/Restore Register 0 (GSRR0) and Guest Save/Restore Register 1 (GSRR1) to save the context at interrupt time; they use the Guest Exception Syndrome Register (GESR) to post exception syndrome information and the Guest Data Exception Address Register (GDEAR) to post the effective address of a data reference.

Most interrupts are directed to the embedded hypervisor state. Some interrupts can be directed to the guest state if the interrupt is a system call interrupt or if the processor is currently executing in guest state ($MSR[GS] = 1$) and the interrupt is configured by the Embedded Processor Control Register (EPCR) to be directed to the guest state.

7.3 Interrupt Classes

All interrupts, except for machine check, can be categorized according to two independent characteristics of the interrupt:

- Asynchronous or synchronous
- Critical or noncritical

7.3.1 Asynchronous Interrupts

Asynchronous interrupts are caused by events that are independent of instruction execution. For asynchronous interrupts, the address reported to the interrupt handling routine is the address of the instruction that would have executed next, had the asynchronous interrupt not occurred.

7.3.2 Synchronous Interrupts

Synchronous interrupts are those that are caused directly by the execution (or attempted execution) of instructions, and are further divided into two classes, *precise* and *imprecise*.

Synchronous, precise interrupts are those that *precisely* indicate the address of the instruction causing the exception that generated the interrupt; or, for certain synchronous, precise interrupt types, the address of the immediately following instruction.

Synchronous, imprecise interrupts are those that can indicate the address of the instruction that caused the exception that generated the interrupt, or the address of some instruction after the one that caused the exception.

7.3.2.1 Synchronous, Precise Interrupts

When the execution or attempted execution of an instruction causes a synchronous, precise interrupt, the following conditions exist when the associated interrupt handler begins execution:

- SRR0 (see *Section 7.5.5 Save/Restore Register 0 (SRR0)* on page 289, CSRR0 (see *Section 7.5.9 Critical Save/Restore Register 0 (CSRR0)* on page 294), or GSRR0 (see *Section 7.5.7 Guest Save/Restore Register 0 (GSRR0)* on page 292) addresses either the instruction that caused the exception that gener-

ated the interrupt, or the instruction immediately following this instruction. Which instruction is addressed can be determined from a combination of the interrupt type and the setting of certain fields of the ESR (see *Section 7.5.17 Exception Syndrome Register (ESR)* on page 302) or GESR (see *Section 7.5.18 Guest Exception Syndrome Register (GESR)* on page 303).

- The interrupt is generated such that all instructions preceding the instruction that caused the exception appear to have completed with respect to the executing processor. However, some storage accesses associated with these preceding instructions might not have been performed with respect to other processors and mechanisms.
- The instruction that caused the exception might appear not to have begun execution (except for having caused the exception), might have been partially executed, or might have completed, depending on the interrupt type (see *Partially Executed Instructions* on page 283).
- Architecturally, no instruction beyond the one that caused the exception has executed.

7.3.2.2 Synchronous, Imprecise Interrupts

When the execution or attempted execution of an instruction causes a synchronous, imprecise interrupt, the following conditions exist when the associated interrupt handler begins execution:

- SRR0, GSRR0, or CSRR0 addresses either the instruction that caused the exception that generated the interrupt or some instruction following this instruction.
- The interrupt is generated such that all instructions preceding the instruction addressed by SRR0, GSRR0, or CSRR0 appear to have completed with respect to the executing processor.
- If the imprecise interrupt is forced by the context-synchronizing mechanism due to an instruction that causes another exception that generates an interrupt (for example, alignment or data storage), then SRR0 addresses the interrupt-forcing instruction; the interrupt-forcing instruction might have been partially executed (see *Partially Executed Instructions* on page 283).
- If the imprecise interrupt is forced by the execution-synchronizing mechanism due to executing an execution-synchronizing instruction other than **msync** or **isync**, then SRR0, CSRR0, or GSRR0 addresses the interrupt-forcing instruction, and the interrupt-forcing instruction appears not to have begun execution (except for its forcing the imprecise interrupt). If the imprecise interrupt is forced by an **msync** or **isync** instruction, SRR0, CSRR0, or GSRR0 can address either the **msync** or **isync** instruction or the following instruction.
- If the imprecise interrupt is not forced by either the context-synchronizing mechanism or the execution-synchronizing mechanism, the instruction addressed by SRR0, CSRR0, or GSRR0 might have been partially executed (see *Partially Executed Instructions* on page 283).
- No instruction following the instruction addressed by SRR0, CSRR0, or GSRR0 has executed.

Many synchronous, imprecise interrupts in the A2O Core are the special cases of delayed interrupts, which can result when certain kinds of exceptions occur while the corresponding interrupt type is disabled. The first of these is the floating-point enabled exception type of program interrupt. For this type of interrupt to occur, a floating-point unit must be attached to the auxiliary processor interface of the A2O Core, and the Floating-Point Enabled Exception Summary bit of the Floating-Point Status and Control Register (FPSCR[FEX]) must be set while floating-point enabled exception type of program interrupts are disabled because MSR[FE0,FE1] are both 0. If and when such interrupts are subsequently enabled by setting one or the other (or both) of MSR[FE0,FE1] to 1 while FPSCR[FEX] is still 1, a synchronous, imprecise form of the floating-point enabled exception type of program interrupt occurs; SRR0 is set to the address of the instruction that would have

executed next (that is, the instruction after the one that updated MSR[FE0,FE1]). If the MSR was updated by an **rfi**, **rftci**, **rftgi**, or **rftmci** instruction, SRR0 is set to the address to which the **rfi**, **rftci**, or **rftmci** was returning, and not to the instruction address that is sequentially after the **rftci**, **rftgi**, or **rftmci**.

The second type of delayed interrupt that can be handled as a synchronous, imprecise interrupt is the debug interrupt. Similar to the floating-point enabled exception type of program interrupt, the debug interrupt can be temporarily disabled by an MSR bit, MSR[DE]. Accordingly, certain kinds of debug exceptions can occur and be recorded in the Debug Status Register (DBSR) while MSR[DE] is 0, and later lead to a delayed debug interrupt if MSR[DE] is set to 1 while a debug exception is still set in the DBSR. If and when this occurs, the interrupt is either synchronous and imprecise or it is asynchronous, depending on the type of debug exception causing the interrupt. In either case, CSRR0 is set to the address of the instruction that would have executed next (that is, the instruction after the one that set MSR[DE] to 1). If MSR[DE] is set to 1 by **rftci**, **rftgi**, or **rftmci**, CSRR0 is set to the address to which the **rftci**, **rftgi**, or **rftmci** was returning, and not to the address of the instruction that was sequentially after the **rftci**, **rftgi**, or **rftmci**.

Another interrupt that is handled as a synchronous, imprecise interrupt is the debug interrupt, when using the data value compare (DVC) facility on a load instruction.

Besides these special cases of program and debug interrupts, all other synchronous interrupts are handled precisely by the A20 Core, except the **FP** enabled exception type of program interrupts when the processor is operating in one of the architecturally-defined imprecise modes (MSR[FE0,FE1] != 0b00).

See *Program Interrupt* on page 321 and *Debug Interrupt* on page 331 for a more detailed description of these interrupt types, including both the precise and imprecise cases.

7.3.3 Critical and Noncritical Interrupts

Interrupts can also be classified as critical or noncritical interrupts. Certain interrupt types demand immediate attention, even if other interrupt types are currently being processed and have not yet had the opportunity to save the state of the machine (that is, the return address and captured state of the MSR). To enable taking a critical interrupt immediately after a noncritical interrupt has occurred (that is, before the state of the machine has been saved), two sets of Save/Restore Register pairs are provided. Critical interrupts use the Save/Restore Register pair CSRR0/CSRR1. Noncritical interrupts use Save/Restore Register pair SRR0/SRR1 or, for interrupts directed to guest state, GSRR0/GSRR1.

7.3.4 Machine Check Interrupts

Machine check interrupts are a special case. They are typically caused by some kind of hardware or storage subsystem failure or by an attempt to access an invalid address. A machine check can be caused indirectly by the execution of an instruction, but not be recognized or reported until long after the processor has executed past the instruction that caused the machine check. As such, machine check interrupts cannot properly be classified as either synchronous or asynchronous, nor as precise or imprecise. They also do not belong to either the critical or the noncritical interrupt class. Instead, machine check interrupts have associated with them a unique pair of save/restore registers, Machine Check Save/Restore Registers 0/1 (MCSRR0/1).

Architecturally, the following general rules apply for Machine Check interrupts:

1. No instruction after the one whose address is reported to the machine check interrupt handler in MCSRR0 has begun execution.
2. The instruction whose address is reported to the machine check interrupt handler in MCSRR0, and all prior instructions, might or might not have completed successfully. All those instructions that are ever

going to complete appear to have done so already, and have done so within the context existing before the machine check interrupt. No further interrupt (other than possible additional machine check interrupts) will occur as a result of those instructions.

With the A2O core, machine check interrupts can be caused by machine check exceptions on a memory access for an instruction fetch, for a data access, or for a translation lookaside buffer (TLB) access. Some of the interrupts generated behave as synchronous, precise interrupts, while other are handled in an asynchronous fashion.

In the case of an instruction-synchronous machine check exception, the A2O core handles the interrupt as a synchronous, precise interrupt, assuming machine check interrupts are enabled ($MSR[ME] = 1$). That is, if a machine check exception is detected during an instruction fetch, the exception is not *reported* to the interrupt mechanism unless and until execution is attempted for the instruction address at which the machine check exception occurred. If, for example, the direction of the instruction stream is changed (perhaps due to a branch instruction), such that the instruction at the address associated with the machine check exception will not be executed, then the exception will not be reported and no interrupt will occur. If and when an instruction machine check exception is reported and if machine check interrupts are enabled at the time of the reporting of the exception, the interrupt will be synchronous and precise and $MCSRR0$ will be set to the instruction address that led to the exception. If machine check interrupts are *not* enabled at the time of the reporting of an instruction machine check exception, a machine check interrupt will *not* be generated (*ever*, even if and when $MSR[ME]$ is subsequently set to 1).

Instruction asynchronous machine check, data asynchronous machine check, and TLB asynchronous machine check exceptions, on the other hand, are handled in an asynchronous fashion. That is, the address reported in $MCSRR0$ might not be related to the instruction that prompted the access that led, directly or indirectly, to the machine check exception. The address might be that of an instruction before or after the exception-causing instruction, or it might reference the exception causing instruction, depending on the nature of the access, the type of error encountered, and the circumstances of the instruction's execution within the processor pipeline. If $MSR[ME]$ is 0 at the time of a machine check exception that is handled in this asynchronous way, a machine check interrupt *will* subsequently occur if and when $MSR[ME]$ is set to 1.

See *Machine Check Interrupt* on page 310 for more detailed information about machine check interrupts.

7.4 Interrupt Processing

Associated with each type of interrupt is an *interrupt vector*; that is, the address of the initial instruction that is executed when the corresponding interrupt occurs.

Interrupt processing consists of saving a small part of the processor state in certain registers, identifying the cause of the interrupt in another register, and continuing execution at the corresponding interrupt vector location. When an exception exists and the corresponding interrupt type is enabled, the following actions are performed, in order:

1. $SRR0$ (for noncritical class interrupts), $CSRR0$ (for critical class interrupts), $GSRR0$ (for interrupt directed to guest state), or $MCSRR0$ (for machine check interrupts) is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.

Programming Note: The contents of $SRR0$, $CSRR0$, $GSRR0$, $MCSRR0$, $DEAR$, or $GDEAR$ when an interrupt is taken are mode dependent, reflecting the computation mode currently in use (specified by $MSR[CM]$) and the computation mode entered for execution of the interrupt (specified by $EPCR[ICM]$).

The undefined portions are defined in the A2 hardware, and the contents of these registers can be described as follows:

- if (MSR[CM] = 0) & (EPCR[ICM] = 0) then
SRR0 ≤ 32^0 || Addr_{32:63}
 - if (MSR[CM] = 0) & (EPCR[ICM] = 1) then
SRR0 ≤ 32^0 || Addr_{32:63}
 - if (MSR[CM] = 1) & (EPCR[ICM] = 1) then
SRR0 ≤ Addr_{0:63}
 - if (MSR[CM] = 1) & (EPCR[ICM] = 0) then
SRR0 ≤ Addr_{0:63}
2. The ESR or GESR (for interrupts directed to the guest state) is loaded with information specific to the exception type. Note that many interrupt types can only be caused by a single type of exception, and thus do not need nor use an ESR setting to indicate the cause of the interrupt. Machine check interrupts load the MCSR.
 3. SRR1 (for noncritical class interrupts), CSRR1 (for critical class interrupts), GSRR0 (for interrupt directed to guest state), or MCSRR1 (for machine check interrupts) is loaded with a copy of the contents of the MSR.
 4. The MSR is updated as described below. The new values take effect beginning with the first instruction following the interrupt.
 - MSR[EE, PR, FP, FE0, FE1, IS, DS] are set to 0 by all interrupts.
 - MSR[GS] is left unchanged when an interrupt is directed to the guest state; otherwise, it is set to 0 by all interrupts.
 - MSR bits corresponding to MSRP bits set to 1 are left unchanged when an interrupt is directed to the guest state; otherwise, it is set to 0 by all interrupts.
 - MSR[ME] is set to 0 by machine check interrupts and left unchanged by all other interrupts.
 - MSR[CE,DE] is set to 0 by critical class interrupts, Debug interrupts, and machine check interrupts, and is left unchanged by all other interrupts.
 - If the interrupt is directed to the guest state, MSR[CM] is set to EPCR[GICM]; otherwise, MSR[CM] is set to EPCR[ICM].
 - Other supported MSR bits are left unchanged by all interrupts.

See *Machine State Register (MSR)* on page 285 for more detail on the definition of the MSR.

5. Instruction fetching and execution resumes, using the new MSR value, at the interrupt vector address, which is specific to the interrupt type and is determined as follows:

$$\text{IVPR}_{0:51} \parallel \text{12-bit Interrupt Offset or GIVPR}_{0:51} \parallel \text{12-bit Interrupt Offset}$$

IVPR is used if MSR[GS] is set to 0; otherwise GIVPR is used, except for the guest processor doorbell interrupt. Also see *Interrupt Fixed Offsets* on page 300, *Interrupt Vector Prefix Register (IVPR)* on page 301, and *Guest Interrupt Vector Prefix Register (GIVPR)* on page 301.

At the end of a noncritical interrupt handling routine, execution of an **rfi** causes the MSR to be restored from the contents of SRR1 and instruction execution to resume at the address contained in SRR0. Likewise, execution of an **rfdi** performs the same function at the end of a critical interrupt handling routine, using CSRR0 instead of SRR0 and CSRR1 instead of SRR1. **rfdi** uses MCSRR0 and MCSRR1 in the same manner. **rfdi** uses GSRR0 and GSRR1.

Programming Note: In general, at process switch, due to possible process interlocks and possible data availability requirements, the operating system needs to consider executing the following instructions:

- **stwcx.** or **stdcx.** to clear the reservation if one is outstanding, to ensure that an **lwarx** in the “old” process is not paired with an **stwcx.** or **stdcx.** in a “new” process. See the instruction descriptions for **lwarx,** **ldarx,** **stwcx.** and **stdcx.** in the Power ISA specification for more information about storage reservations.
- **msync,** to ensure that all storage operations of an interrupted process are complete with respect to other processors before that process begins executing on another processor.
- **isync, rfi, rfc1, rfg1,** or **rfmci,** to ensure that the instructions in the “new” process execute in the “new” context.

7.4.1 Partially Executed Instructions

In general, the architecture permits load and store instructions to be partially executed, interrupted, and then to be restarted from the beginning upon return from the interrupt. To guarantee that a particular load or store instruction will complete without being interrupted and restarted, software must mark the storage being referred to as guarded, and must use an elementary (not a string or multiple) load or store that is 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 instruction is partially executed and then interrupted:

- For an elementary load, no part of the target register (**GPR(RT), FPR(FRT),** or auxiliary processor register) will have been altered.
- For the “update” forms of load and store instructions, the update register, **GPR(RA),** will not have been altered.

On the other hand, the following effects are permissible when certain instructions are partially executed and then restarted:

- For any store instruction, some of the bytes at the addressed storage location might have been accessed or updated (if write access to that page in which bytes were altered is permitted by the access control mechanism). In addition, if the address for an **stwcx.** instruction is not aligned on a word boundary or the address for an **stdcx.** instruction is not aligned on a doubleword boundary, the value in **CR[CR0]** is undefined. It is also undefined whether or not the reservation (if one existed) has been cleared.
- For any load, some of the bytes at the addressed storage location might have been accessed (if read access to that page in which bytes were accessed is permitted by the access control mechanism). In addition, if the address for an **lwarx** instruction is not aligned on a word boundary or the address for an **ldarx** instruction is not aligned on a doubleword boundary, it is undefined whether or not a reservation has been set.
- For load multiple and load string instructions, some of the registers in the range to be loaded might have been altered. Including the addressing registers (**GPR[RA]** and possibly **GPR[RB]**) in the range to be loaded is an invalid form of these instructions (and a programming error). Thus, the rules for partial execution do not protect against overwriting of these registers. Such possible overwriting of the addressing registers makes these invalid forms of load multiple and load strings inherently nonrestartable.

In no case will access control be violated.

As previously stated, the only load or store instructions that are guaranteed to not be interrupted after being partially executed are elementary-aligned and guarded loads and stores. All others can be interrupted after being partially executed. The following list identifies the specific instruction types for which interruption after partial execution can occur, as well as the specific interrupt types that can cause the interruption:

1. Any load or store (except elementary-aligned and guarded):

- Critical input
- Machine check
- Guest processor doorbell machine check
- External input
- Program (imprecise mode floating-point enabled)

Note: This type of interrupt can lead to partial execution of a load or store instruction under the architectural definition only; the A2O Core handles the imprecise modes of the floating-point enabled exceptions precisely; hence, this type of interrupt does not lead to partial execution.

- Embedded hypervisor privilege
- Decrementer
- Guest Decrementer
- Fixed-interval timer
- Guest Fixed-interval timer
- Watchdog timer
- Guest Watchdog timer
- Processor doorbell critical
- Guest processor doorbell critical
- Processor doorbell
- Guest processor doorbell
- User decrementer
- Debug (unconditional debug event)

2. Unaligned-elementary load or store or any load or store multiple or string:
All of the above listed under item 1, plus the following:

- Alignment
- Data storage (if the access crosses a memory page boundary)
- Debug (data address compare, data value compare)

7.5 Interrupt Processing Registers

The interrupt processing registers include:

- *Machine State Register Protect (MSRP)* on page 287
- *Embedded Processor Control Register (EPCR)* on page 288
- *Embedded Processor Control Register (EPCR)* on page 288
- *Save/Restore Register 1 (SRR1)* on page 290
- *Guest Save/Restore Register 0 (GSRR0)* on page 292
- *Guest Save/Restore Register 0 (GSRR0)* on page 292
- *Critical Save/Restore Register 0 (CSRR0)* on page 294
- *Critical Save/Restore Register 1 (CSRR1)* on page 295
- *Machine Check Save/Restore Register 0 (MCSRR0)* on page 297
- *Machine Check Save/Restore Register 1 (MCSRR1)* on page 297

- *Data Exception Address Register (DEAR)* on page 299
- *Guest Data Exception Address Register (GDEAR)* on page 299
- *Interrupt Vector Prefix Register (IVPR)* on page 301
- *Guest Interrupt Vector Prefix Register (GIVPR)* on page 301
- *Exception Syndrome Register (ESR)* on page 302
- *Guest Exception Syndrome Register (GESR)* on page 303
- *Machine Check Status Register (MCSR)* on page 305

Also described in this section is the *Machine State Register (MSR)* on page 285, which belongs to the category of processor control registers.

7.5.1 Register Mapping

Some special purpose register (SPR) accesses in guest state are mapped to analogous registers for the guest state. This removes the requirement for the hypervisor software to handle embedded hypervisor privilege interrupts for these accesses and makes the required emulated changes by the hypervisor for these high-use registers.

Accesses to the registers listed in *Table 7-1* are changed by the processor to the registers given in the table when the processor is in guest state (MSR[GS] = 1). Access to these registers is not mapped when not in guest state.

Table 1. Register Mapping in Guest State

SPR Accessed	SPR Mapped to	Type of Access
SRR0	GSRR0	mtspr, mfspr
SRR1	GSRR1	mtspr, mfspr
ESR	GESR	mtspr, mfspr
DEAR	GDEAR	mtspr, mfspr
PIR	GPIR	mtspr, mfspr
SPRG0	GSPRG0	mtspr, mfspr
SPRG1	GSPRG1	mtspr, mfspr
SPRG2	GSPRG2	mtspr, mfspr
SPRG3	GSPRG3	mtspr, mfspr
USPRG3	GSPRG3	mtspr, mfspr

7.5.2 Machine State Register (MSR)

The MSR is a register of its own unique type that controls important chip functions, such as the enabling or disabling of various interrupt types.

The MSR can be written from a GPR using the **mtmsr** instruction. The contents of the MSR can be read into a GPR using the **mfmsr** instruction. The MSR[EE] bit can be set or cleared atomically using the **wrtee** or **wrteei** instructions. The MSR contents are also automatically saved, altered, and restored by the interrupt-handling mechanism.

The following table shows the MSR bit definitions and describes the function of each bit.

Register Short Name:	MSR	Read Access:	Priv
Decimal SPR Number:	N/A	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32	CM	0b0	<u>Computation Mode</u> 0 The processor runs in 32-bit mode 1 The processor runs in 64-bit mode
33:34	///	0b00	<u>Reserved</u>
35	GS	0b0	<u>Guest State</u> 0 The processor is in hypervisor state if MSR[PR] = 0. 1 The processor is in guest state. MSR[GS] cannot be changed unless MSR[GS]=0.
36	///	0b0	<u>Reserved</u>
37	UCLE	0b0	<u>User Cache Locking Enable</u> 0 Cache Locking instructions are privileged 1 Cache Locking instructions can be executed in user mode (MSR[PR]=1)
38	SPV	0b0	<u>Vector Available</u> 0 The processor cannot execute any Vector instruction 1 The processor can execute Vector instructions
39:45	///	0x0	<u>Reserved</u>
46	CE	0b0	<u>Critical Enable</u> 0 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are disabled 1 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are enabled
47	///	0b0	<u>Reserved</u>
48	EE	0b0	<u>External Enable</u> 0 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are disabled. 1 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are enabled. When an interrupt masked by MSR[EE] is directed to the hypervisor state, the interrupt is enabled if MSR[EE]=1 or MSR[GS]=1 except for Guest Processor Doorbell which is enabled if MSR[EE]=1 and MSR[GS]=1. When an interrupt masked by MSR[EE] is directed to the guest state, the interrupt is enabled if MSR[EE]=1 and MSR[GS]=1.
49	PR	0b0	<u>Problem State</u> 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.) 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource
50	FP	0b0	<u>Floating-Point Available</u> 0 The processor cannot execute any floating-point instructions, including floating-point loads, stores and moves. 1 The processor can execute floating-point instructions.

Bit(s):	Field Name:	Init	Description
51	ME	0b0	<u>Machine Check Enable</u> 0 Machine Check interrupts are disabled 1 Machine Check interrupts are enabled
52	FE0	0b0	<u>Floating-Point Exception Mode 0</u> Sets Floating-Point Exception Mode
53	///	0b0	<u>Reserved</u>
54	DE	0b0	<u>Debug Interrupt Enable</u> 0 Debug interrupts are disabled 1 Debug interrupts are enabled if DBCR0[IDM]=1
55	FE1	0b0	<u>Floating-Point Exception Mode 1</u> Sets Floating-Point Exception Mode
56:57	///	0b00	<u>Reserved</u>
58	IS	0b0	<u>Instruction Address Space</u> 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	0b0	<u>Data Address Space</u> 0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry) 1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry)
60:63	///	0b0000	<u>Reserved</u>

7.5.3 Machine State Register Protect (MSRP)

Register Short Name:	MSRP	Read Access:	Hypv
Decimal SPR Number:	311	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:36	///	0x0	<u>Reserved</u>
37	UCLEP	0b0	<u>User Cache Lock Enable Protect</u> 0 Guest privileged state can modify MSR[UCLE]. 1 Guest privileged state cannot modify MSR[UCLE]. When MSRP[UCLEP] = 1, cache locking instructions are not permitted to execute in the guest privileged state and cause an Embedded Hypervisor Privilege exception when executed.
38:53	///	0x0	<u>Reserved</u>
54	DEP	0b0	<u>Debug Enable Protect</u> 0 Guest privileged state can modify MSR[DE]. 1 Guest privileged state cannot modify MSR[DE].
55:63	///	0x0	<u>Reserved</u>

7.5.4 Embedded Processor Control Register (EPCR)

Register Short Name:	EPCR	Read Access:	Hypv
Decimal SPR Number:	307	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	EXTGS	0b0	<p><u>External Input interrupt directed to Guest State</u> Controls whether an External Input Interrupt is taken in the guest state or the hypervisor state.</p> <p>0 External Inputs interrupts are directed to the hypervisor state. External Input interrupts pend until MSR[GS]=1 or MSR[EE]=1. 1 External Inputs interrupts are directed to the guest state. External Input interrupts pend until MSR[GS]=1 and MSR[EE]=1.</p>
33	DTLBGS	0b0	<p><u>Data TLB Error interrupt directed to Guest State</u> Controls whether a Data TLB Error Interrupt that occurs in the guest state is taken in the guest state or the hypervisor state.</p> <p>0 Data TLB Error Interrupts that occur in the guest state are directed to the hypervisor state. 1 Data TLB Error Interrupts that occur in the guest state are directed to the guest state.</p>
34	ITLBGS	0b0	<p><u>Instruction TLB Error interrupt directed to Guest State</u> Controls whether an Instruction TLB Error Interrupt that occurs in the guest state is taken in the guest state or the hypervisor state.</p> <p>0 Instruction TLB Error Interrupts that occur in the guest state are directed to the hypervisor state. 1 Instruction TLB Error Interrupts that occur in the guest state are directed to the guest state.</p>
35	DSIGS	0b0	<p><u>Data Storage interrupt directed to Guest State</u> Controls whether a Data Storage Interrupt that occurs in the guest state is taken in the guest state or the hypervisor state.</p> <p>0 Data Storage Interrupts that occur in the guest state are directed to the hypervisor state. 1 Data Storage Interrupts that occur in the guest state are directed to the guest state, except for a Data Storage Interrupt due to a TLB Ineligible exception is directed to the hypervisor state, regardless of the existence of other exceptions that cause a Data Storage interrupt.</p>
36	ISIGS	0b0	<p><u>Instruction Storage interrupt directed to Guest State</u> Controls whether an Instruction Storage Interrupt that occurs in the guest state is taken in the guest state or the hypervisor state.</p> <p>0 Instruction Storage Interrupts that occur in the guest state are directed to the hypervisor state. 1 Instruction Storage Interrupts that occur in the guest state are directed to the guest state, except for an Instruction Storage Interrupt due to a TLB Ineligible exception is directed to the hypervisor state, regardless of the existence of other exceptions that cause an Instruction Storage interrupt.</p>
37	DUVD	0b0	<p><u>Disable Hypervisor Debug</u> Controls whether Debug Events occur in the hypervisor state.</p> <p>0 Debug events can occur in the hypervisor state. 1 Debug events are suppressed in the hypervisor state.</p>

Bit(s):	Field Name:	Init	Description
38	ICM	0b0	<u>Interrupt Computation Mode</u> Controls the computational mode of the processor when an interrupt occurs that is directed to the hypervisor state. At interrupt time, EHCSR[ICM] is copied into MSR[CM] if the interrupt is directed to the hypervisor state 0 Interrupts that are directed to the hypervisor state will execute in 32-bit mode. 1 Interrupts that are directed to the hypervisor state will execute in 64-bit mode.
39	GICM	0b0	<u>Guest Interrupt Computation Mode</u> Controls the computational mode of the processor when an interrupt occurs that is directed to the guest state. At interrupt time, EHCSR[GICM] is copied into MSR[CM] if the interrupt is directed to the guest state 0 Interrupts that are directed to the guest state will execute in 32-bit mode. 1 Interrupts that are directed to the guest state will execute in 64-bit mode.
40	DGTMI	0b0	<u>Disable TLB Guest Management Instructions</u> Controls whether guest state can execute any TLB management instructions. 0 tlbilx, tlbwe, and tlbsrx (for a Logical to Real Address translation hit) are allowed to execute normally when MSR[GS,PR] = 0b10. 1 tlbilx, tlbwe, and tlbsrx always cause an Embedded Hypervisor Privilege Interrupt when MSR[GS,PR] = 0b10.
41	DMIUH	0b0	<u>Disable MAS Interrupt Updates for Hypervisor</u> 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 when a Data or Instruction TLB Error Interrupt or a Data or Instruction Storage Interrupt is taken in the hypervisor. 1 MAS registers updates are disabled and left unchanged when a Data or Instruction TLB Error Interrupt or a Data or Instruction Storage Interrupt is taken in the hypervisor.
42:63	///	0x0	<u>Reserved</u>

7.5.5 Save/Restore Register 0 (SRR0)

SRR0 is an SPR that is used to save the machine state on noncritical interrupts and to restore the machine state when an **rfi** is executed. When a noncritical interrupt occurs, SRR0 is set to an address associated with the process that was executing at the time. When **rfi** is executed, instruction execution returns to the address in SRR0.

In general, SRR0 contains the address of the instruction that caused the noncritical interrupt or the address of the instruction to return to after a noncritical interrupt is serviced. See the individual descriptions under *Interrupt Definitions* on page 306 for an explanation of the precise address recorded in SRR0 for each noncritical interrupt type.

SRR0 can be written from a GPR using **mtspr** and can be read into a GPR using **mfspr**.

SRR0 is mapped to GSRR0 when in the guest state (MSR[GS] = 1).

Register Short Name:	SRR0	Read Access:	Priv
Decimal SPR Number:	26	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GSRR0	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	SRR0	0x0	<u>Save/Restore Register 0</u> This register is used to save machine state on non-critical interrupts, and to restore machine state when an rfi is executed. On a non-critical interrupt, SRR0 is set to the current or next instruction address. When rfi is executed, instruction execution continues at the address in SRR0. In general, SRR0 contains the address of the instruction that caused the non-critical interrupt, or the address of the instruction to return to after a non-critical interrupt is serviced.
62:63	///	0b00	<u>Reserved</u>

7.5.6 Save/Restore Register 1 (SRR1)

SRR1 is an SPR that is used to save the machine state on noncritical interrupts and to restore the machine state when an **rfi** is executed. When a noncritical interrupt is taken, the contents of the MSR (before the MSR is cleared by the interrupt) are placed into SRR1. When **rfi** is executed, the MSR is restored with the contents of SRR1.

Bits of SRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note: An MSR bit that is reserved can be altered by **rfi**, consistent with the value being restored from SRR1.

SRR1 can be written from a GPR using **mtspr** and can be read into a GPR using **mfspr**.

SRR1 is mapped to GSRR1 when in the guest state (MSR[GS] = 1).

Register Short Name:	SRR1	Read Access:	Priv
Decimal SPR Number:	27	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GSRR1	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32	CM	0b0	<u>Computation Mode</u> 0 The processor runs in 32-bit mode 1 The processor runs in 64-bit mode
33:34	///	0b00	<u>Reserved</u>
35	GS	0b0	<u>Guest State</u> 0 The processor is in hypervisor state if MSR[PR] = 0. 1 The processor is in guest state.
36	///	0b0	<u>Reserved</u>
37	UCLE	0b0	<u>User Cache Locking Enable</u> 0 Cache Locking instructions are privileged 1 Cache Locking instructions can be executed in user mode (MSR[PR]=1)
38	SPV	0b0	<u>Vector Available</u> 0 The processor cannot execute any Vector instruction 1 The processor can execute Vector instructions

Bit(s):	Field Name:	Init	Description
39:45	///	0x0	<u>Reserved</u>
46	CE	0b0	<u>Critical Enable</u> 0 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are disabled 1 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are enabled
47	///	0b0	<u>Reserved</u>
48	EE	0b0	<u>External Enable</u> 0 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are disabled. 1 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are enabled. When an interrupt masked by MSR[EE] is directed to the hypervisor state, the interrupt is enabled if MSR[EE]=1 or MSR[GS]=1 except for Guest Processor Doorbell which is enabled if MSR[EE]=1 and MSR[GS]=1. When an interrupt masked by MSR[EE] is directed to the guest state, the interrupt is enabled if MSR[EE]=1 and MSR[GS]=1.
49	PR	0b0	<u>Problem State</u> 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.) 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource
50	FP	0b0	<u>Floating-Point Available</u> 0 The processor cannot execute any floating-point instructions, including floating-point loads, stores and moves. 1 The processor can execute floating-point instructions.
51	ME	0b0	<u>Machine Check Enable</u> 0 Machine Check interrupts are disabled 1 Machine Check interrupts are enabled
52	FE0	0b0	<u>Floating-Point Exception Mode 0</u> Sets Floating-Point Exception Mode
53	///	0b0	<u>Reserved</u>
54	DE	0b0	<u>Debug Interrupt Enable</u> 0 Debug interrupts are disabled 1 Debug interrupts are enabled if DBCR0[IDM]=1
55	FE1	0b0	<u>Floating-Point Exception Mode 1</u> Sets Floating-Point Exception Mode
56:57	///	0b00	<u>Reserved</u>
58	IS	0b0	<u>Instruction Address Space</u> 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	0b0	<u>Data Address Space</u> 0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry) 1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry)
60:63	///	0b0000	<u>Reserved</u>

7.5.7 Guest Save/Restore Register 0 (GSRR0)

GSRR0 is an SPR that is used to save the machine state on interrupts directed to the guest state and to restore the machine state when an **rfgi** is executed. When an interrupt occurs, GSRR0 is set to an address associated with the process that was executing at the time. When **rfgi** is executed, instruction execution returns to the address in GSRR0.

In general, GSRR0 contains the address of the instruction that caused the noncritical interrupt or the address of the instruction to return to after a noncritical interrupt is serviced. See the individual descriptions under *Interrupt Definitions* on page 306 for an explanation of the precise address recorded in GSRR0 for each noncritical interrupt type.

GSRR0 can be written from a GPR using **mtspr** and can be read into a GPR using **mfspir**.

GSRR0 is also accessed by reading SRR0 when in the guest state (MSR[GS] = 1).

Register Short Name:	GSRR0	Read Access:	Priv
Decimal SPR Number:	378	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	GSRR0	0x0	<u>Guest Save/Restore Register 0</u> This register is used to save machine state on interrupts directed to the guest state, and to restore machine state when an rfgi is executed. When an interrupt is taken, the GSRR0 is set to the current or next instruction address. When rfgi is executed, instruction execution continues at the address in GSRR0. In general, GSRR0 contains the address of the instruction that caused the interrupt, or the address of the instruction to return to after a critical interrupt is serviced.
62:63	///	0b00	<u>Reserved</u>

7.5.8 Guest Save/Restore Register 1 (GSRR1)

GSRR1 is an SPR that is used to save the machine state on interrupts directed to the guest state and to restore the machine state when an **rfgi** is executed. When an interrupt is taken, the contents of the MSR (before the MSR being cleared by the interrupt) are placed into SRR1. When **rfgi** is executed, the MSR is restored with the contents of GSRR1.

Bits of GSRR1 that correspond to reserved bits in the MSR are also reserved.

GSRR1 can be written from a GPR using **mtspir** and can be read into a GPR using **mfspir**.

GSRR1 is also accessed by reading SRR1 when in the guest state (MSR[GS] = 1).

Register Short Name:	GSRR1	Read Access:	Priv
Decimal SPR Number:	379	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	CM	0b0	<u>Computation Mode</u> 0 The processor runs in 32-bit mode 1 The processor runs in 64-bit mode
33:34	///	0b00	<u>Reserved</u>
35	GS	0b0	<u>Guest State</u> When set, indicates the processor is running in the Guest State under the control of an hypervisor program. 0 The processor is not running in the guest state. 1 The processor is running in the guest state.
36	///	0b0	<u>Reserved</u>
37	UCLE	0b0	<u>User Cache Locking Enable</u> 0 Cache Locking instructions are privileged 1 Cache Locking instructions can be executed in user mode (MSR[PR]=1)
38	SPV	0b0	<u>Vector Available</u> 0 The processor cannot execute any Vector instruction 1 The processor can execute Vector instructions
39:45	///	0x0	<u>Reserved</u>
46	CE	0b0	<u>Critical Enable</u> 0 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are disabled 1 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are enabled
47	///	0b0	<u>Reserved</u>
48	EE	0b0	<u>External Enable</u> 0 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are disabled. 1 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are enabled. When an interrupt masked by MSR[EE] is directed to the hypervisor state, the interrupt is enabled if MSR[EE]=1 or MSR[GS]=1 except for Guest Processor Doorbell which is enabled if MSR[EE]=1 and MSR[GS]=1. When an interrupt masked by MSR[EE] is directed to the guest state, the interrupt is enabled if MSR[EE]=1 and MSR[GS]=1.
49	PR	0b0	<u>Problem State</u> 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.) 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource
50	FP	0b0	<u>Floating-Point Available</u> 0 The processor cannot execute any floating-point instructions, including floating-point loads, stores and moves. 1 The processor can execute floating-point instructions.
51	ME	0b0	<u>Machine Check Enable</u> 0 Machine Check interrupts are disabled 1 Machine Check interrupts are enabled
52	FE0	0b0	<u>Floating-Point Exception Mode 0</u> Sets Floating-Point Exception Mode
53	///	0b0	<u>Reserved</u>

Bit(s):	Field Name:	Init	Description
54	DE	0b0	<u>Debug Interrupt Enable</u> 0 Debug interrupts are disabled 1 Debug interrupts are enabled if DBCR0[IDM]=1
55	FE1	0b0	<u>Floating-Point Exception Mode 1</u> Sets Floating-Point Exception Mode
56:57	///	0b00	<u>Reserved</u>
58	IS	0b0	<u>Instruction Address Space</u> 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	0b0	<u>Data Address Space</u> 0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry) 1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry)
60:63	///	0b0000	<u>Reserved</u>

7.5.9 Critical Save/Restore Register 0 (CSRR0)

CSRR0 is an SPR that is used to save the machine state on critical interrupts and to restore the machine state when an **rfci** is executed. When a critical interrupt occurs, CSRR0 is set to an address associated with the process that was executing at the time. When **rfci** is executed, instruction execution returns to the address in CSRR0.

In general, CSRR0 contains the address of the instruction that caused the critical interrupt or the address of the instruction to return to after a critical interrupt is serviced. See the individual descriptions under *Interrupt Definitions* on page 306 for an explanation of the precise address recorded in CSRR0 for each critical interrupt type.

CSRR0 can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Register Short Name:	CSRR0	Read Access:	Hypv
Decimal SPR Number:	58	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	SRR0	0x0	<u>Critical Save/Restore Register 0</u> This register is used to save machine state on critical interrupts, and to restore machine state when an rfci is executed. When a critical interrupt is taken, the CSRR0 is set to the current or next instruction address. When rfci is executed, instruction execution continues at the address in CSRR0. In general, CSRR0 contains the address of the instruction that caused the critical interrupt, or the address of the instruction to return to after a critical interrupt is serviced.
62:63	///	0b00	<u>Reserved</u>

7.5.10 Critical Save/Restore Register 1 (CSRR1)

CSRR1 is an SPR that is used to save the machine state on critical interrupts and to restore the machine state when an **rfci** is executed. When a critical interrupt is taken, the contents of the MSR (before the MSR is cleared by the interrupt) are placed into CSRR1. When **rfci** is executed, the MSR is restored with the contents of CSRR1.

Bits of CSRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note: An MSR bit that is reserved can be altered by **rfci**, consistent with the value being restored from CSRR1.

CSRR1 can be written from a GPR using **mtspr** and can be read into a GPR using **mfspr**.

Register Short Name:	CSRR1	Read Access:	Hypv
Decimal SPR Number:	59	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	CM	0b0	<u>Computation Mode</u> 0 The processor runs in 32-bit mode 1 The processor runs in 64-bit mode
33:34	///	0b00	<u>Reserved</u>
35	GS	0b0	<u>Guest State</u> 0 The processor is in hypervisor state if MSR[PR] = 0. 1 The processor is in guest state.
36	///	0b0	<u>Reserved</u>
37	UCLE	0b0	<u>User Cache Locking Enable</u> 0 Cache Locking instructions are privileged 1 Cache Locking instructions can be executed in user mode (MSR[PR]=1)
38	SPV	0b0	<u>Vector Available</u> 0 The processor cannot execute any Vector instruction 1 The processor can execute Vector instructions
39:45	///	0x0	<u>Reserved</u>
46	CE	0b0	<u>Critical Enable</u> 0 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are disabled 1 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are enabled
47	///	0b0	<u>Reserved</u>

Bit(s):	Field Name:	Init	Description
48	EE	0b0	<p><u>External Enable</u></p> <p>0 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are disabled.</p> <p>1 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are enabled.</p> <p>When an interrupt masked by MSR[EE] is directed to the hypervisor state, the interrupt is enabled if MSR[EE]=1 or MSR[GS]=1 except for Guest Processor Doorbell which is enabled if MSR[EE]=1 and MSR[GS]=1.</p> <p>When an interrupt masked by MSR[EE] is directed to the guest state, the interrupt is enabled if MSR[EE]=1 and MSR[GS]=1.</p>
49	PR	0b0	<p><u>Problem State</u></p> <p>0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.)</p> <p>1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource</p>
50	FP	0b0	<p><u>Floating-Point Available</u></p> <p>0 The processor cannot execute any floating-point instructions, including floating-point loads, stores and moves.</p> <p>1 The processor can execute floating-point instructions.</p>
51	ME	0b0	<p><u>Machine Check Enable</u></p> <p>0 Machine Check interrupts are disabled</p> <p>1 Machine Check interrupts are enabled</p>
52	FE0	0b0	<p><u>Floating-Point Exception Mode 0</u></p> <p>Sets Floating-Point Exception Mode</p>
53	///	0b0	<u>Reserved</u>
54	DE	0b0	<p><u>Debug Interrupt Enable</u></p> <p>0 Debug interrupts are disabled</p> <p>1 Debug interrupts are enabled if DBCR0[IDM]=1</p>
55	FE1	0b0	<p><u>Floating-Point Exception Mode 1</u></p> <p>Sets Floating-Point Exception Mode</p>
56:57	///	0b00	<u>Reserved</u>
58	IS	0b0	<p><u>Instruction Address Space</u></p> <p>0 The processor directs all instruction fetches to address space 0 (TS=0 in the relevant TLB entry)</p> <p>1 The processor directs all instruction fetches to address space 1 (TS=1 in the relevant TLB entry)</p>
59	DS	0b0	<p><u>Data Address Space</u></p> <p>0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry)</p> <p>1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry)</p>
60:63	///	0b0000	<u>Reserved</u>

7.5.11 Machine Check Save/Restore Register 0 (MCSRR0)

MCSRR0 is an SPR that is used to save the machine state on machine check interrupts and to restore the machine state when an **rfmci** is executed. When a machine check interrupt occurs, MCSRR0 is set to an address associated with the process that was executing at the time. When **rfmci** is executed, instruction execution returns to the address in MCSRR0.

In general, MCSRR0 contains the address of the instruction that caused the machine check interrupt or the address of the instruction to return to after a machine check interrupt is serviced. See the individual descriptions under *Interrupt Definitions* on page 306 for an explanation of the precise address recorded in MCSRR0 for each machine check interrupt type.

MCSRR0 can be written from a GPR using **mtspr** and can be read into a GPR using **mfspir**.

Register Short Name:	MCSRR0	Read Access:	Hypv
Decimal SPR Number:	570	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	SRR0	0x0	<u>Critical Save/Restore Register 0</u> Machine Check Save/Restore Register 0 (MCSRR0) is used to save machine state on Machine Check interrupts, and to restore machine state when an rfmci is executed. When a Machine Check interrupt is taken, the MCSRR0 is set to the current or next instruction address. When rfmci is executed, instruction execution continues at the address in MCSRR0. In general, MCSRR0 contains the address of an instruction that was executing or about to be executed when the Machine Check exception occurred.
62:63	///	0b00	<u>Reserved</u>

7.5.12 Machine Check Save/Restore Register 1 (MCSRR1)

MCSRR1 is an SPR that is used to save the machine state on machine check interrupts and to restore the machine state when an **rfmci** is executed. When a machine check interrupt is taken, the contents of the MSR (before the MSR is cleared by the interrupt) are placed into MCSRR1. When **rfmci** is executed, the MSR is restored with the contents of MCSRR1.

Bits of MCSRR1 that correspond to reserved bits in the MSR are also reserved.

Programming Note: An MSR bit that is reserved can be altered by **rfmci**, consistent with the value being restored from MCSRR1.

MCSRR1 can be written from a GPR using **mtspr** and can be read into a GPR using **mfspir**.

Register Short Name:	MCSRR1	Read Access:	Hypv
Decimal SPR Number:	571	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	CM	0b0	<u>Computation Mode</u> 0 The processor runs in 32-bit mode 1 The processor runs in 64-bit mode
33:34	///	0b00	<u>Reserved</u>
35	GS	0b0	<u>Guest State</u> 0 The processor is in hypervisor state if MSR[PR] = 0. 1 The processor is in guest state.
36	///	0b0	<u>Reserved</u>
37	UCLE	0b0	<u>User Cache Locking Enable</u> 0 Cache Locking instructions are privileged 1 Cache Locking instructions can be executed in user mode (MSR[PR]=1)
38	SPV	0b0	<u>Vector Available</u> 0 The processor cannot execute any Vector instruction 1 The processor can execute Vector instructions
39:45	///	0x0	<u>Reserved</u>
46	CE	0b0	<u>Critical Enable</u> 0 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are disabled 1 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are enabled
47	///	0b0	<u>Reserved</u>
48	EE	0b0	<u>External Enable</u> 0 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are disabled. 1 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are enabled. When an interrupt masked by MSR[EE] is directed to the hypervisor state, the interrupt is enabled if MSR[EE]=1 or MSR[GS]=1 except for Guest Processor Doorbell which is enabled if MSR[EE]=1 and MSR[GS]=1. When an interrupt masked by MSR[EE] is directed to the guest state, the interrupt is enabled if MSR[EE]=1 and MSR[GS]=1.
49	PR	0b0	<u>Problem State</u> 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.) 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource
50	FP	0b0	<u>Floating-Point Available</u> 0 The processor cannot execute any floating-point instructions, including floating-point loads, stores and moves. 1 The processor can execute floating-point instructions.
51	ME	0b0	<u>Machine Check Enable</u> 0 Machine Check interrupts are disabled 1 Machine Check interrupts are enabled
52	FE0	0b0	<u>Floating-Point Exception Mode 0</u> Sets Floating-Point Exception Mode
53	///	0b0	<u>Reserved</u>
54	DE	0b0	<u>Debug Interrupt Enable</u> 0 Debug interrupts are disabled 1 Debug interrupts are enabled if DBCR0[IDM]=1

Bit(s):	Field Name:	Init	Description
55	FE1	0b0	<u>Floating-Point Exception Mode 1</u> Sets Floating-Point Exception Mode
56:57	///	0b00	<u>Reserved</u>
58	IS	0b0	<u>Instruction Address Space</u> 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	0b0	<u>Data Address Space</u> 0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry) 1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry)
60:63	///	0b0000	<u>Reserved</u>

7.5.13 Data Exception Address Register (DEAR)

The DEAR contains the address that was referenced by a load, store, or cache management instruction that caused an alignment, data TLB miss, or data storage exception.

The DEAR can be written from a GPR using **mtspr** and can be read into a GPR using **mfspr**.

DEAR is mapped to GDEAR when in the guest state (MSR[GS] = 1).

Register Short Name:	DEAR	Read Access:	Priv
Decimal SPR Number:	61	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GDEAR	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DEAR	0x0	<u>Data Exception Address Register</u> The DEAR contains the address that was referenced by a Load, Store or Cache Management instruction that caused an Alignment, Data TLB Miss, or Data Storage interrupt.

7.5.14 Guest Data Exception Address Register (GDEAR)

The GDEAR contains the address that was referenced by a load, store, or cache management instruction that caused an alignment, data TLB miss, or data storage exception when the interrupt is directed to the guest state.

The GDEAR can be written from a GPR using **mtspr** and can be read into a GPR using **mfspr**. GDEAR is also accessed by reading DEAR when in the guest state (MSR[GS] = 1).

Register Short Name:	GDEAR	Read Access:	Priv
Decimal SPR Number:	381	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y

Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	GDEAR	0x0	<u>Guest Data Exception Address Register</u> The GDEAR contains the address that was referenced by a Load, Store or Cache Management instruction that caused an Alignment, Data TLB Miss, or Data Storage interrupt when directed to the guest state.

Interrupt Fixed Offsets

An Interrupt offset specifies the 12-bit low-order effective address offset for each interrupt type. The value is the offset from the base address provided by either the IVPR (see *Interrupt Vector Prefix Register (IVPR)* on page 301) or the GIVPR (see *Guest Interrupt Vector Prefix Register (GIVPR)* on page 301). The interrupt effective address is either:

$$IVPR_{0:51} \parallel \text{12-bit Interrupt Offset}$$

or

$$GIVPR_{0:51} \parallel \text{12-bit Interrupt Offset}$$

IVPR is used if MSR[GS] is set to 0, otherwise GIVPR is used.

Table 7-2 identifies the specific interrupt offsets associated with each interrupt type.

Table 2. *Interrupt Types and Associated Offsets* (Sheet 1 of 2)

Offset	Interrupt Type
0x040	Debug
0x020	Critical input
0x000	Machine check
0x060	Data storage
0x080	Instruction storage
0x0A0	External input
0x0C0	Alignment
0x0E0	Program
0x100	Floating-point unavailable
0x120	System call
0x140	Auxiliary processor unavailable
0x160	Decrementer, Guest Decrementer
0x180	Fixed interval timer, Guest Fixed Interval timer
0x1A0	Watchdog timer, Guest Watchdog timer
0x1C0	Data TLB error
0x1E0	Instruction TLB error
0x200	Vector unavailable interrupt

Table 2. Interrupt Types and Associated Offsets (Sheet 2 of 2)

Offset	Interrupt Type
0x280	Processor doorbell interrupt
0x2A0	Processor doorbell critical interrupt
0x2C0	Guest processor doorbell
0x2E0	Guest processor doorbell critical; guest processor doorbell machine check
0x300	Embedded hypervisor system call
0x320	Embedded hypervisor privilege
0x340	LRAT Error interrupt
0x360...0x7FF	Reserved
0x800	User decrementer
0x820	Performance monitor
0x840...0xFFFF	Reserved

7.5.15 Interrupt Vector Prefix Register (IVPR)

The IVPR provides the high-order 52 bits of the effective address of the interrupt vectors for interrupts that are not directed to the guest state.

The IVPR can be written from a GPR using **mtspr** and can be read into a GPR using **mf spr**.

Register Short Name:	IVPR	Read Access:	Hypv
Decimal SPR Number:	63	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:51	IVPR	0x0	<u>Interrupt Vector Prefix Register</u> Provides the high-order bits of the address of the exception processing routines
52:63	///	0x0	<u>Reserved</u>

7.5.16 Guest Interrupt Vector Prefix Register (GIVPR)

The GIVPR provides the high-order 52 bits of the effective address of the interrupt vectors for interrupts that are directed to the guest state.

The GIVPR can be written from a GPR using **mtspr**, and can be read into a GPR using **mf spr**.

Register Short Name:	GIVPR	Read Access:	Priv
Decimal SPR Number:	447	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N

Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:51	GIVPR	0x0	<u>Interrupt Vector Prefix Register</u> Provides the high-order bits of the address of the exception processing routines when in guest state.
52:63	///	0x0	<u>Reserved</u>

7.5.17 Exception Syndrome Register (ESR)

The ESR provides a *syndrome* to differentiate between the different kinds of exceptions that can generate the same interrupt type. Upon the generation of one of these types of interrupt, the bit or bits corresponding to the specific exception that generated the interrupt is set, and all other ESR bits are cleared. Other interrupt types do not affect the contents of the ESR. See the individual interrupt descriptions under *Interrupt Definitions* on page 306 for an explanation of the ESR settings for each interrupt type, as well as a more detailed explanation of the function of certain ESR fields.

The ESR can be written from a GPR using **mtspr** and can be read into a GPR using **mfspr**.

The ESR is mapped to GESR when in the guest state (MSR[GS] = 1).

Register Short Name:	ESR	Read Access:	Priv
Decimal SPR Number:	62	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GESR	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:35	///	0b0000	<u>Reserved</u>
36	PIL	0b0	<u>Illegal Instruction exception</u> 1 Indicates Illegal Instruction exception
37	PPR	0b0	<u>Privileged Instruction exception</u> 1 Indicates Privileged Instruction exception
38	PTR	0b0	<u>Trap exception</u> 1 Indicates Trap exception
39	FP	0b0	<u>Floating-point operation</u> 1 Indicates Floating-point operation
40	ST	0b0	<u>Store operation</u> 1 Indicates Store operation
41	///	0b0	<u>Reserved</u>
42	DLK0	0b0	<u>Data Locking Exception 0</u> 1 Indicates a dcbtIs, dcbtstIs, or dcbIc instruction was executed with MSR[PR]=1 and MSR[UCLE]=0
43	DLK1	0b0	<u>Data Locking Exception 1</u> 1 Indicates an icbtIs or icbIc was executed MSR[PR]=1 and MSR[UCLE]=0

Bit(s):	Field Name:	Init	Description
44	AP	0b0	<u>Auxiliary Processor operation</u> 1 Indicates Auxiliary Processor operation
45	PUO	0b0	<u>Unimplemented Operation exception</u> 1 Indicates Unimplemented Operation exception
46	BO	0b0	<u>Byte Ordering exception</u> 1 Indicates Byte Ordering exception
47	PIE	0b0	<u>Imprecise exception</u> 1 Indicates Imprecise exception
48	///	0b0	<u>Reserved</u>
49	UCT	0b0	<u>Unavailable Coprocessor Type</u> 1 indicates that execution of an icswx instruction was attempted which specified a coprocessor type which was marked as unavailable.
50:52	///	0b000	<u>Reserved</u>
53	DATA	0b0	<u>Data Access</u> 1 indicates if the interrupt is due to is a LRAT miss resulting from a Page Table translation of a Load, Store or Cache Management operand address
54	TLBI	0b0	<u>TLB Ineligible</u> 1 indicates a TLB Ineligible exception occurred during a Page Table translation for the instruction causing the interrupt
55	PT	0b0	<u>Page Table</u> 1 indicates a Page Table Fault or Read or Write Access Control exception occurred during a Page Table translation for the instruction causing the interrupt
56	SPV	0b0	<u>Vector operation</u> 1 Indicates Vector operation
57	EPID	0b0	<u>External Process ID operation</u> 1 indicates the instruction causing the interrupt is an External Process ID instruction
58:63	///	0x0	<u>Reserved</u>

7.5.18 Guest Exception Syndrome Register (GESR)

The GESR provides a *syndrome* to differentiate between the different kinds of exceptions that can generate the same interrupt type for interrupts that are directed to the guest state. Upon the generation of one of these types of interrupt, the bit or bits corresponding to the specific exception that generated the interrupt is set, and all other GESR bits are cleared. Other interrupt types do not affect the contents of the GESR. See the individual interrupt descriptions under *Interrupt Definitions* on page 306 for an explanation of the GESR settings for each interrupt type, as well as a more detailed explanation of the function of certain GESR fields.

The GESR can be written from a GPR using **mtspr** and can be read into a GPR using **mfspir**.

GESR is also accessed by reading ESR when in the guest state (MSR[GS] = 1).

Register Short Name:	GESR	Read Access:	Priv
Decimal SPR Number:	383	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:35	///	0b0000	<u>Reserved</u>
36	PIL	0b0	<u>Illegal Instruction exception</u> 1 Indicates Illegal Instruction exception
37	PPR	0b0	<u>Privileged Instruction exception</u> 1 Indicates Privileged Instruction exception
38	PTR	0b0	<u>Trap exception</u> 1 Indicates Trap exception
39	FP	0b0	<u>Floating-point operation</u> 1 Indicates Floating-point operation
40	ST	0b0	<u>Store operation</u> 1 Indicates Store operation
41	///	0b0	<u>Reserved</u>
42	DLK0	0b0	<u>Data Locking Exception 0</u> 1 Indicates a dcbtIs, dcbtstIs, or dcbIc instruction was executed in user mode
43	DLK1	0b0	<u>Data Locking Exception 1</u> 1 Indicates an icbtIs or icbIc was executed in user mode
44	AP	0b0	<u>Auxiliary Processor operation</u> 1 Indicates Auxiliary Processor operation
45	PUO	0b0	<u>Unimplemented Operation exception</u> 1 Indicates Unimplemented Operation exception
46	BO	0b0	<u>Byte Ordering exception</u> 1 Indicates Byte Ordering exception
47	PIE	0b0	<u>Imprecise exception</u> 1 Indicates Imprecise exception
48	///	0b0	<u>Reserved</u>
49	UCT	0b0	<u>Unavailable Coprocessor Type</u> 1 indicates that execution of an icswx instruction was attempted which specified a coprocessor type which was marked as unavailable in the HACOP or ACOP (if MSR[PR]=1) registers.
50:52	///	0b000	<u>Reserved</u>
53	DATA	0b0	<u>Data Access</u> 1 indicates if the interrupt is due to is a LRAT miss resulting from a Page Table translation of a Load, Store or Cache Management operand address
54	TLBI	0b0	<u>TLB Ineligible</u> 1 indicates a TLB Ineligible exception occurred during a Page Table translation for the instruction causing the interrupt
55	PT	0b0	<u>Page Table</u> 1 indicates a Page Table Fault or Read or Write Access Control exception occurred during a Page Table translation for the instruction causing the interrupt
56	SPV	0b0	<u>Vector operation</u> 1 Indicates Vector operation
57	EPID	0b0	<u>External Process ID operation</u> 1 indicates the instruction causing the interrupt is an External Process ID instruction
58:63	///	0x0	<u>Reserved</u>

7.5.19 Machine Check Status Register (MCSR)

The MCSR contains status to allow the machine check interrupt handler software to determine the cause of a machine check exception. See *Machine Check Interrupt* on page 310 for more information.

The MCSR can be written from a GPR using **mtspr** and can be read into a GPR using **mfspr**.

Register Short Name:	MCSR	Read Access:	Hypv
Decimal SPR Number:	572	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:47	///	0x0	<u>Reserved</u>
48	DPOVR	0b0	<u>DITC Data Port Overrun Condition</u> 1 Indicates an overrun condition detected on a mtdp instruction.
49	DDMH	0b0	<u>Data Cache Directory Multihit Error</u> 1 Indicates multihit condition detected in data cache directory when enabled by XUCR4[MDDMH]=1
50	TLBIVAXSR	0b0	<u>tlbivax Snoop Reject</u> 1 Indicates that a tlbivax snoop (which is tagged with a local core indication) may be rejected back to the L2 when the snoop's LPID mismatches the current core's LPIDR value. This can only occur when CCR2[NOTLB]=1 or MMUCR1[TLBI_REJ]=1.
51	TLBLRUPE	0b0	<u>TLB LRU Parity Error</u> 1 Indicates Parity Error detected for TLB LRU tlbre, tlbsx, or reload
52	IL2ECC	0b0	<u>Instruction Cache L2 ECC Error</u> 1 Indicates instruction cache detected an L2 uncorrectable ECC error
53	DL2ECC	0b0	<u>Data Cache L2 ECC Error</u> 1 Indicates data cache detected an L2 uncorrectable ECC error
54	DDPE	0b0	<u>Data Cache Directory Parity Error</u> 1 Indicates Parity Error detected in data cache directory when enabled by XUCR0[MDDP]=1
55	EXT	0b0	<u>External Machine Check</u> 1 Indicates external machine check was asserted
56	DCPE	0b0	<u>Data Cache Parity Error</u> 1 Indicates Parity Error detected in data cache when enabled by XUCR0[MDCP]=1
57	IEMH	0b0	<u>I-ERAT Multi-Hit Error</u> 1 Indicates Multiple Entry Hit Error detected for I-ERAT compare
58	DEMH	0b0	<u>D-ERAT Multi-Hit Error</u> 1 Indicates Multiple Entry Hit Error detected for D-ERAT compare
59	TLBMH	0b0	<u>TLB Multi-Hit Error</u> 1 Indicates Multiple Entry Hit Error detected for TLB compare
60	IEPE	0b0	<u>I-ERAT Parity Error</u> 1 Indicates Parity Error detected for I-ERAT eratre, eratsx, or compare

Bit(s):	Field Name:	Init	Description
61	DEPE	0b0	<u>D-ERAT Parity Error</u> 1 Indicates Parity Error detected for D-ERAT eratre, eratsx, or compare
62	TLBPE	0b0	<u>TLB Parity Error</u> 1 Indicates Parity Error detected for TLB tlbre, tlbsx, or reload
63	///	0b0	<u>Reserved</u>

7.6 Interrupt Definitions

Table 7-3 provides a summary of each interrupt type in the order corresponding to their associated offset. The table also summarizes the various exception types that can cause that interrupt type; the classification of the interrupt; which ESR bits can be set, if any; and which mask bits can mask the interrupt type, if any.

Detailed descriptions of each of the interrupt types follow the table.

Table 3. Interrupt and Exception Types (Sheet 1 of 4)

Offset	Interrupt Type	Exception Type	Asynchronous	Synchronous, Precise	Synchronous, Imprecise	Critical	ESR (GESR) (See Note 4)	MSR Mask Bits	DBCRO/TCR Mask Bit	Notes
0x020	Critical Input	Critical Input	x			x		CE GS		1
0x000	Machine Check	Machine Check - All Sources						ME GS		2
0x060	Data Storage	Read Access Control		x			[FP,AP,SPV] [EPID]			
		Write Access Control		x			ST [FP,AP,SPV] [EPID]			
		Cache Locking		x			[ST] {DLK, ILK}			
		Byte Ordering		x			BO [ST] [FP,AP,SPV] [EPID]			5
		Storage Synchronization		x			[ST]			
		Virtualization Fault		x			[ST] [FP,AP,SPV] [EPID]			
		Page Table Fault		x			PT [ST] [FP,AP,SPV] [EPID]			
	TLB Ineligible		x			TLBI [ST] [FP,AP,SPV] [EPID]				

Table 3. Interrupt and Exception Types (Sheet 2 of 4)

Offset	Interrupt Type	Exception Type	Asynchronous	Synchronous, Precise	Synchronous, Imprecise	Critical	ESR (GESR) (See Note 4)	MSR Mask Bits	DBCRO/TCR Mask Bit	Notes
0x080	Instruction Storage	Execute Access Control		x						
		Byte Ordering		x			BO			6
		Page Table Fault		x			PT			
		TLB Ineligible		x			TLBI			
0x0A0	External Input	External Input	x				EE GS EE&GS		1	
0x0C0	Alignment	Alignment		x			[ST] [FP,AP,SPV] [EPID]			
0x0E0	Program	Illegal Instruction		x			PIL,[FP,AP,SPV]			7
		Privileged Instruction		x			PPR,[FP,AP,SPV]			
		Trap		x			PTR			
		FP Enabled		x	x		FP,[PIE]	FE0 FE1		8
		AP Enabled		x			AP			8
		Unimplemented Operation		x			PUO [FP,AP,SPV]			
0x100	FP Unavailable	FP Unavailable		x					8	
0x120	System Call	System Call		x						
0x140	AP Unavailable	AP Unavailable		x					8	
0x160	Decrementer	Decrementer	x					EE GS	DIE	
	Guest Decrementer									
0x180	Fixed Interval Timer	Fixed Interval Timer	x					EE GS	FIE	
	Guest Fixed Internal Timer									
0x1A0	Watchdog Timer	Watchdog Timer	x			x		CE GS	WIE	
	Guest Watchdog Timer									
0x1C0	Data TLB Error	Data TLB Miss		x			[ST],[FP,AP,SPV] [EPID]			
0x1E0	Instruction TLB Error	Instruction TLB Miss		x						
0x200	Vector Unavailable	Vector Unavailable		x			SPV			

Table 3. Interrupt and Exception Types (Sheet 3 of 4)

Offset	Interrupt Type	Exception Type	Asynchronous	Synchronous, Precise	Synchronous, Imprecise	Critical	ESR (GESR) (See Note 4)	MSR Mask Bits	DBCRO/TCR Mask Bit	Notes
0x040	Debug	Trap		x	x	x		DE	IDM	3
		Instruction Address Compare		x	x	x		DE	IDM	3
		Data Address Compare	x	x	x	x		DE	IDM	3
		Data Value Compare	x	x	x	x		DE	IDM	3
		Instruction Complete		x		x		DE	IDM	3
		Branch Taken		x		x		DE	IDM	3
		Return		x	x	x		DE	IDM	3
		Interrupt	x		x	x		DE	IDM	3
		Unconditional	x			x		DE	IDM	3
		Instruction Value Compare		x	x	x		DE	IBM	3
0x280	Processor Doorbell	Processor Doorbell	x				EE GS			
0x2A0	Processor Doorbell Critical	Processor Doorbell Critical	x			x	CE GS			
0x2C0	Guest Processor Doorbell	Guest Processor Doorbell	x				EE&GS			
0x2E0	Guest Processor Doorbell Critical	Guest Processor Doorbell Critical	x			x	CE&GS			
	Guest Processor Doorbell Machine Check	Guest Processor Doorbell Machine Check	x			x	ME&GS			
0x300	Embedded Hypervisor System Call	Embedded Hypervisor System Call		x						
0x320	Embedded Hypervisor Privilege	Embedded Hypervisor Privilege		x						
0x340	LRAT Error	LRAT Error		x			[ST] [FP,AP,SPV] [DATA] [PT] [EPID]			
0x800	User Decrementer	User Decrementer	x					EE GS	UDIE	
0x820	Performance Monitor	Performance Monitor	x					EE GS		

Table 3. Interrupt and Exception Types (Sheet 4 of 4)

Offset	Interrupt Type	Exception Type	Asynchronous	Synchronous, Precise	Synchronous, Imprecise	Critical	ESR (GESR) (See Note 4)	MSR Mask Bits	DBCRO/TCR Mask Bit	Notes
<p>Notes:</p> <ol style="list-style-type: none"> Although it is not specified as part of the Power ISA, 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. Machine check interrupts are not classified as asynchronous nor synchronous. They are also not classified as critical or noncritical, because they use their own unique set of Save/Restore Registers, MCSRR0/1. See <i>Machine Check Interrupts</i> on page 280, and <i>Machine Check Interrupt</i> on page 310. Debug exceptions have special rules regarding their interrupt classification (synchronous or asynchronous and precise or imprecise), depending on the particular debug mode being used and other conditions (see <i>Debug Interrupt</i> on page 331). In general, when an interrupt causes a particular ESR(GESR) bit or bits to be set as indicated in the table, it also causes all other ESR(GESR) bits to be cleared. If no ESR(GESR) setting is indicated for any of the exception types within a given interrupt type, the ESR(GESR) is unchanged for that interrupt type. The syntax for the ESR(GESR) setting indication is as follows: [xxx] means ESR(GESR)[xxx] can be set. [xxx,yyy,zzz] means any one (or none) of ESR(GESR)[xxx] or ESR(GESR)[yyy] or ESR(GESR)[zzz] can be set, but never more than one. {xxx,yyy,zzz} means that any combination of ESR(GESR)[xxx], ESR(GESR)[yyy], and ESR(GESR)[zzz] can be set, including all or none. xxx means ESR[xxx] will be set. The byte ordering exception type of data storage interrupts can only occur when the A20 Core is connected to a floating-point unit or auxiliary processor, and then only when executing <i>EP</i> or <i>AXU</i> load or store instructions. See <i>Data Storage Interrupt</i> on page 313 for more detailed information about these kinds of exceptions. The byte ordering exception type of instruction storage interrupts are defined by the Power ISA, but cannot occur within the A20 Core. The core is capable of executing instructions from both big-endian and little-endian code pages. An attempt to execute an instruction that is not provided by the implementation results in an illegal instruction program type of interrupt. Floating-point unavailable and auxiliary processor unavailable interrupts, as well as floating-point enabled and auxiliary processor enabled exception type of program interrupts, can only occur when the A20 Core is connected to a floating-point unit or an auxiliary processor, and then only when executing instruction opcodes that are recognized by the floating-point unit or auxiliary processor, respectively. 										

7.6.1 Critical Input Interrupt

A critical input interrupt occurs when no higher priority exception exists, a critical input exception is presented to the interrupt mechanism, and (MSR[CE] or MSR[GS]) = 1. A critical input exception is caused by the activation of an asynchronous input to the A20 Core. Although the only mask for this interrupt type within the core is the MSR[CE] bit, system implementations typically provide an alternative means for independently masking the interrupt requests from the various devices that collectively can activate the A20 Core’s critical input interrupt request input.

Note: MSR[CE] also enables other interrupts. See *Table 7-3 Interrupt and Exception Types* on page 306.

Note: When a critical input interrupt occurs, the interrupt processing registers are updated as indicated below (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x020.

Critical Save/Restore Register 0 (CSRR0)	Set to the effective address of the next instruction to be executed.
Critical Save/Restore Register 1 (CSRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM]. ME unchanged. All other MSR bits set to 0.

Programming Note: Software is responsible for taking any actions that are required by the implementation to clear any critical input exception status (such that the input signal of the critical input interrupt request is deasserted) before reenabling MSR[CE], to avoid another, redundant critical input interrupt.

7.6.2 Machine Check Interrupt

A machine check interrupt occurs when no higher priority exception exists, a machine check exception is presented to the interrupt mechanism, and $(MSR[ME] \text{ or } MSR[GS]) = 1$. The Power ISA architecture specifies machine check interrupts as neither synchronous nor asynchronous; indeed, the exact causes and details of handling such interrupts are implementation dependent. Regardless, for this particular processor core, it is useful to describe the handling of interrupts caused by various types of machine check exceptions in those terms. The A2O Core includes the following four types of machine check exceptions:

Instruction Synchronous Machine Check Exception

An instruction synchronous machine check exception is caused when a timeout or read error is signaled on the A2 core interface during an instruction fetch operation.

Such an exception is not presented to the interrupt handling mechanism, however, unless and until such time as the execution is attempted of an instruction at an address associated with the instruction fetch for which the instruction machine check exception was asserted.

If MSR[ME] is 1 when the instruction machine check exception is presented to the interrupt mechanism, execution of the instruction associated with the exception is suppressed, a machine check interrupt occurs, and the interrupt processing registers are updated as described on page 311. If MSR[ME] is 0, however, the instruction associated with the exception is processed as though the exception did not exist, and a machine check interrupt does *not* occur (*ever*, even if and when MSR[ME] is subsequently set to 1), although the ESR is still updated as described on page 311.

Instruction Asynchronous Machine Check Exception

An instruction asynchronous machine check exception is caused when either:

- The read interrupt request is asserted on the A2 core interface.
- External signal `an_ac_external_mchk` is asserted.

Data Asynchronous Machine Check Exception

A data asynchronous machine check exception is caused when one of the following occurs:

- A timeout, read error, or read interrupt request is signaled on the A2 core interface during a data read operation.
- A timeout, write error, or write interrupt request is signaled on the A2 core interface during a data write operation.
- A parity error is detected on an access to the data cache. XUCR[MDCP] is used to disable parity recovery.

TLB Asynchronous Machine Check Exception

A TLB asynchronous machine check exception is caused when a parity error is detected on an access to the TLB.

When any machine check exception that is handled as an asynchronous interrupt occurs, it is immediately presented to the interrupt handling mechanism. Bits of the MCSR are set as appropriate. A machine check interrupt occurs immediately if MSR[ME] is 1, and the interrupt processing registers are updated as described in the following list. If MSR[ME] is 0, however, the exception is recorded by the setting of the appropriate bits, and deferred until such time as MSR[ME] is subsequently set to 1.

When a machine check interrupt occurs, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x000.

Machine Check Save/Restore Register 0 (MCSRR0)	For an instruction synchronous machine check exception, set to the effective address of the instruction presenting the exception. For an instruction asynchronous machine check, data asynchronous machine check, or TLB asynchronous machine check exception, set to the effective address of the next instruction to be executed.
Machine Check Save/Restore Register 1 (MCSRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM]. All other MSR bits set to 0.
Exception Syndrome Register (ESR)	All defined ESR bits are left unchanged.
Machine Check Status Register (MCSR)	

7.6.2.1 Machine Check Status Register (MCSR)

The MCSR collects status for the machine check exceptions that are handled as asynchronous interrupts: Data asynchronous machine check exception or TLB asynchronous machine check exception. Other bits in the MCSR are set to indicate the exact type of machine check exception.

Register Short Name:	MCSR	Read Access:	Hypv
Decimal SPR Number:	572	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:47	///	0x0	<u>Reserved</u>
48	DPOVR	0b0	<u>DITC Data Port Overrun Condition</u> 1 Indicates an overrun condition detected on a mtdp instruction.
49	DDMH	0b0	<u>Data Cache Directory Multihit Error</u> 1 Indicates multihit condition detected in data cache directory when enabled by XUCR4[MDDMH]=1
50	TLBIVAXSR	0b0	<u>tlbivax Snoop Reject</u> 1 Indicates that a tlbivax snoop (which is tagged with a local core indication) may be rejected back to the L2 when the snoop's LPID mismatches the current core's LPIDR value. This can only occur when CCR2[NOTLB]=1 or MMUCR1[TLBI_REJ]=1.
51	TLBLRUPE	0b0	<u>TLB LRU Parity Error</u> 1 Indicates Parity Error detected for TLB LRU tlbre, tlbsx, or reload
52	IL2ECC	0b0	<u>Instruction Cache L2 ECC Error</u> 1 Indicates instruction cache detected an L2 uncorrectable ECC error
53	DL2ECC	0b0	<u>Data Cache L2 ECC Error</u> 1 Indicates data cache detected an L2 uncorrectable ECC error
54	DDPE	0b0	<u>Data Cache Directory Parity Error</u> 1 Indicates Parity Error detected in data cache directory when enabled by XUCR0[MDDP]=1
55	EXT	0b0	<u>External Machine Check</u> 1 Indicates external machine check was asserted
56	DCPE	0b0	<u>Data Cache Parity Error</u> 1 Indicates Parity Error detected in data cache when enabled by XUCR0[MDCP]=1
57	IEMH	0b0	<u>I-ERAT Multi-Hit Error</u> 1 Indicates Multiple Entry Hit Error detected for I-ERAT compare
58	DEMH	0b0	<u>D-ERAT Multi-Hit Error</u> 1 Indicates Multiple Entry Hit Error detected for D-ERAT compare
59	TLBMH	0b0	<u>TLB Multi-Hit Error</u> 1 Indicates Multiple Entry Hit Error detected for TLB compare
60	IEPE	0b0	<u>I-ERAT Parity Error</u> 1 Indicates Parity Error detected for I-ERAT eratre, eratsx, or compare
61	DEPE	0b0	<u>D-ERAT Parity Error</u> 1 Indicates Parity Error detected for D-ERAT eratre, eratsx, or compare

Bit(s):	Field Name:	Init	Description
62	TLBPE	0b0	<u>TLB Parity Error</u> 1 Indicates Parity Error detected for TLB tlbre, tlbxs, or reload
63	///	0b0	<u>Reserved</u>

See *Machine Check Interrupts* on page 280 for more information about the handling of machine check interrupts within the A2O Core.

7.6.3 Data Storage Interrupt

A data storage interrupt *might* occur when no higher priority exception exists and a data storage exception is presented to the interrupt mechanism. The A2O Core includes the following types of data storage exceptions:

Cache Locking Exception

If a cache locking instruction is executed in user mode (MSR[PR] = 1), a data storage interrupt occurs if any of the following conditions are met:

- (MSRP[UCLEP] = 1 & MSR[GS] = 1).
- (MSRP[UCLEP] = 0 | MSR[GS] = 0) and MSR[UCLE] = 0.

When a cache locking type data storage interrupt occurs, one of the following ESR or GESR bits is set to 1:

Bit	Description
42	DLK ₀ 0 Default setting. 1 A dcbtls , dcbtstls , or dcblc instruction was executed in user mode.
43	DLK ₁ 0 Default setting. 1 An icbtls or icblc instruction was executed in user mode.

Read Access Control Exception

A read access control exception is caused by one of the following cases:

- While in user mode (MSR[PR] = 1), a load instruction attempts to access a location in storage that is not enabled for read access in user mode (that is, the TLB entry associated with the memory page being accessed has UR = 0).
- While in supervisor mode (MSR[PR] = 0), a load instruction attempts to access a location in storage that is not enabled for read access in supervisor mode (that is, the TLB entry associated with the memory page being accessed has SR = 0).

See *Access Control Applied to Cache Management Instructions* on page 178.

Programming Note: The instruction cache management instructions **icbi** and **icbt** are treated as loads from the addressed byte with respect to address translation and protection. These instruction cache management instructions use MSR[DS] rather than MSR[IS] to determine machine translation for their target effective address. Similarly, they use the read access control field (**UR** or **SR**) rather than the execute access control field (**UX** or **SX**) of the TLB entry to determine whether a data storage exception should occur. Instruction storage exceptions and instruction TLB miss exceptions are associated with the *fetching* of instructions not with the *execution* of instructions. Data storage exceptions and data TLB miss exceptions are associated with the *execution*

of instruction cache management instructions, as well as with the execution of load, store, and data cache management instructions.

Write Access Control Exception

A Write Access Control exception is caused by one of the following:

- While in user mode ($MSR[PR] = 1$), a store instruction attempts to access a location in storage that is not enabled for write access in user mode (that is, the TLB entry associated with the memory page being accessed has $UW = 0$).
- While in supervisor mode ($MSR[PR] = 0$), a store instruction attempts to access a location in storage that is not enabled for write access in supervisor mode (that is, the TLB entry associated with the memory page being accessed has $SW = 0$).

See *Access Control Applied to Cache Management Instructions* on page 178.

Byte Ordering Exception

A byte ordering exception occurs when a floating-point unit or auxiliary processor is attached to the A2O Core, and a floating-point or auxiliary processor load or store instruction attempts to access a memory page with a byte order that is not supported by the attached processor. Whether or not a given load or store instruction type is supported for a given byte order is dependent on the implementation of the floating-point or auxiliary processor. All integer load and store instructions supported by the A2O Core are supported for both big-endian and little-endian memory pages.

Unavailable Coprocessor Type Exception

An unavailable coprocessor type exception will occur when following expression is true:

$$MSR[GS,PR] \neq 0b00 \ \& \ (HACOP[CT] == 0 \ | \ (ACOP[CT] == 0 \ \& \ MSR[PR] == 1))$$

Note that for `icswepx`, the following substitutions are made.

EPSC_{EPR} is used in place of MSR_{PR}.
 EPSC_{EGS} is used in place of MSR_{GS}.
 EPSC_{EAS} is used in place of MSR_{DS}.

See *Section 12.5.2 Initiate Coprocessor Store Word External Process ID Indexed (icswepx[.])* on page 512.

Storage Synchronization Exception

A storage synchronization exception occurs when an attempt is made to execute a load and reserve or store conditional instruction from or to a location that is write through required or caching inhibited.

Virtualization Fault Exception

A virtualization fault exception occurs when a load, store, or cache management instruction attempts to access a location in storage that has the virtualization fault (VF) bit set. A data storage interrupt resulting from a virtualization fault exception is always directed to hypervisor state regardless of the setting of EPCR[DSIGS]. See *Access Control Applied to Cache Management Instructions* on page 178.

Page Table Fault Exception

A page table fault exception is caused when a page table translation occurs for a data access due to a load, store or cache management instruction and the page table entry that is accessed is invalid (PTE Valid bit = 0).

TLB Ineligible Exception

A TLB ineligible exception is caused when a page table translation occurs for a data access due to a load, store or cache management instruction and any of the following conditions are true:

- The only TLB entries that can be used to hold the translation for the virtual address have IPROT = 1.
- No TLB array can be loaded from page table for the page size specified by the PTE.
- The PTE[ARPN] is treated as an LPN and there is no TLB array that meets all the following conditions:
 - The TLB array supports the page size specified by the PTE.
 - The TLB array can be loaded from the page table (TLB0CFG[PT] = 1).

A data storage interrupt resulting from a TLB ineligible exception is always directed to hypervisor state regardless of the setting of EPCR[DSIGS].

A data storage interrupt occurs regardless of whether a **stwcx.** or **stdcx.** would have performed its store. The CR[CR0] is not updated.

The following instructions are treated as no-ops and cannot cause a data storage interrupt regardless of the effective address (EA):

- **lswx** or **stswx** with a length of zero (although the target register of **lswx** is still undefined, as it is whether or not a data storage exception occurs)
- **icbt**
- **dcbt, dcbtstep**
- **dcbtst, dcbtstep**
- **dcba**

For all other instructions, if a data storage exception occurs, execution of the instruction causing the exception is suppressed, a data storage interrupt is generated, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x060.

If the interrupt is directed to guest state (EPCR[DSIGS] = 1, MSR[GS] = 1, and TLB[VF] = 0), GSRR0, GSRR1, GDEAR, and GESR, are set in place of SRR0, SRR1, DEAR, and ESR respectively, and instruction execution resumes at address GIVPR[IVP] || 0x060.

Save/Restore Register 0 (SRR0)	Set to the effective address of the instruction causing the data storage interrupt.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[GICM] if the interrupt is directed to guest state; otherwise, it is set to EPCR[ICM]. GS is left unchanged if the interrupt is directed to guest state; otherwise, it is set to zero.

	<p>UCLE is left unchanged if the interrupt is directed to the guest state and MSR[UCLEP] = 1; otherwise, it is set to 0.</p>
	<p>CE, ME, DE Unchanged.</p> <p>If the interrupt is directed to guest state, bits in the MSR corresponding to set bits in the MSRP register are left unchanged. All other MSR bits set to 0.</p>
<p>Data Exception Address Register (DEAR)</p>	<p>If the instruction causing the data storage exception does so with respect to the memory page targeted by the initial effective address calculated by the instruction, the DEAR is set to this calculated effective address. On the other hand, if the data storage exception only occurs due to the instruction causing the exception crossing a memory page boundary, in that the exception is with respect to the attributes of the page accessed after crossing the boundary, then the DEAR is set to the address of the first byte within that page.</p> <p>For example, consider a misaligned load word instruction that targets effective address 0x00000FFF, and that the page containing that address is a 4 KB page. The load word will thus cross the page boundary, and access the next page starting at address 0x00001000. If a read access control exception exists within the first page (because the Read Access Control field for that page is 0), the DEAR is set to 0x00000FFF. On the other hand, if the Read Access Control field of the first page is 1, but the same field is 0 for the next page, then the read access control exception exists only for the second page and the DEAR is set to 0x00001000. Furthermore, the load word instruction in this latter scenario has been partially executed (see <i>Partially Executed Instructions</i> on page 283).</p>
<p>Exception Syndrome Register (ESR)</p>	<p>FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise, set to 0.</p> <p>ST Set to 1 if the instruction causing the interrupt is a store, dcbz, or dcbi instruction; otherwise, set to 0.</p> <p>DLK_{0:1}DLK₀ Set to 1 when a dcbtls, dcbtstls, or dcbic instruction was executed in user mode when MSR[UCLE] = 0; otherwise, set to 0.</p> <p>DLK₁ Set to 1 when a icbtlis or icblc instruction was executed in user mode when MSR[UCLE] = 0; otherwise, set to 0.</p> <p>AXU Set to 1 if the instruction causing the interrupt is an auxiliary processor load or store; otherwise, set to 0.</p> <p>BO Set to 1 if the instruction caused a byte ordering exception; otherwise, set to 0.</p> <p>Note that a read or write access control exception can occur in combination with a byte ordering exception, in which case software needs to examine the TLB entry associated with the address reported in the DEAR to determine whether both exceptions occurred, or just a byte ordering exception.</p>

- EPID Set to 1 if the instruction causing the interrupt is an external process ID instruction; otherwise, set to 0.
 - TLBI Set to 1 if a TLB ineligible exception occurred during a page table translation for the instruction causing the interrupt; otherwise, set to 0.
 - PT Set to 1 if a page table fault or read or write access control exception occurred during a page table translation for the instruction causing the interrupt, or if no TLB entry was created from the page table. Set to an implementation-dependent value if a TLB entry was created; otherwise, set to 0.
 - UCT Set to 1 if an unavailable coprocessor type exception occurred; otherwise, set to zero.
- All other defined ESR bits are set to 0.

The following is a prioritized listing of the various exceptions that cause a data storage interrupt and the corresponding ESR bit, if applicable. Even though multiples of these exceptions can occur, at most one of the following exceptions is reported in the ESR:

- Cache locking: DLK0:1
- Page table fault: PT
- Virtualization fault
- TLB ineligible: TLBI
- Byte ordering: BO
- Read access or write access: If the exception occurred during a page table translation: PT
- Unavailable coprocessor type: UCT

7.6.4 Instruction Storage Interrupt

An instruction storage interrupt occurs when no higher priority exception exists and an instruction storage exception is presented to the interrupt mechanism. Note that although an instruction storage exception can occur during an attempt to *fetch* an instruction, such an exception is not actually presented to the interrupt mechanism until an attempt is made to *execute* that instruction. The A2O Core includes the following types of instruction storage exceptions:

Execute Access Control Exception

An execute access control exception is caused by one of the following:

- While in user mode (MSR[PR] = 1), an instruction fetch attempts to access a location in storage that is not enabled for execute access in user mode (that is, the TLB entry associated with the memory page being accessed has UX = 0).
- While in supervisor mode (MSR[PR] = 0), an instruction fetch attempts to access a location in storage that is not enabled for execute access in supervisor mode (that is, the TLB entry associated with the memory page being accessed has SX = 0).

Architecture Note: The Power ISA defines an additional instruction storage exception, the byte ordering exception. This exception is defined to assist implementations that cannot support dynamically switching byte ordering between consecutive instruction fetches or cannot support a given byte order at all. The A2O Core, however, supports instruction fetching from both big-endian and little-endian memory pages, so this exception cannot occur.

Byte Ordering Exception

An instruction storage byte ordering exception cannot occur in A2.

Page Table Fault Exception

A page table fault exception is caused when a page table translation occurs for a data access due to a load, store, or cache management instruction and the page table entry that is accessed is invalid (PTE Valid bit = 0).

TLB Ineligible Exception

A TLB ineligible exception is caused when a page table translation occurs for an instruction fetch and any of the following conditions are true:

- The only TLB entries that can be used to hold the translation for the virtual address have IPROT = 1.
- No TLB array can be loaded from the page table for the page size specified by the PTE.
- The PTE[ARPN] is treated as an LPN, and there is no TLB array that meets all the following conditions:
 - The TLB array supports the page size specified by the PTE.
 - The TLB array can be loaded from the page table (TLB0CFG[PT] = 1).

An instruction storage interrupt resulting from a TLB ineligible exception is always directed to hypervisor state regardless of the setting of EPCR[ISIGS].

When an instruction storage interrupt occurs, the processor suppresses the execution of the instruction causing the instruction storage exception, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x080.

If the interrupt is directed to the guest state (EPCR[ISIGS] = 1 and MSR[GS] = 1), GSRR0, GSRR1, and GESR are set in place of SRR0, SRR1, and ESR respectively, and instruction execution resumes at address GIVPR[IVP] || 0x080.

Save/Restore Register 0 (SRR0)	Set to the effective address of the instruction causing the instruction storage interrupt.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[GICM] if the interrupt is directed to guest state; otherwise, it is set to EPCR[ICM]. GS is left unchanged if the interrupt is directed to guest state; otherwise, it is set to zero. UCLE is left unchanged if the interrupt is directed to the guest state and MSRP[UCLEP] = 1; otherwise, it is set to 0.

CE, ME, DE Unchanged.

If the interrupt is directed to guest state, bits in the MSR corresponding to set bits in the MSRP register are left unchanged. All other MSR bits set to 0.

Exception Syndrome Register (ESR)

BO Set to 0.

TLBI Set to 1 if a TLB ineligible exception occurred during a page table translation for the instruction causing the interrupt; otherwise, set to 0.

PT Set to 1 if a page table fault or read or write access control exception occurred during a page table translation for the instruction causing the interrupt, or if no TLB entry was created from the page table. Set to an implementation-dependent value if a TLB entry was created; otherwise set to 0.

All other defined ESR bits are set to 0.

The following is a prioritized listing of the various exceptions that cause a data storage interrupt and the corresponding ESR bit, if applicable. Even though multiples of these exceptions can occur, at most one of the following exceptions is reported in the ESR:

- Page table fault: PT
- TLB ineligible: TLBI
- Byte ordering: BO
- Execute access: If the exception occurred during a Page Table translation, PT

7.6.5 External Input Interrupt

An external input interrupt occurs when no higher priority exception exists, an external input exception is presented to the interrupt mechanism, and external interrupts are enabled. An external input exception is caused by the activation of an asynchronous input to the A2O Core. Although the only mask for this interrupt type within the core is the MSR[EE] bit, system implementations typically provide an alternative means for independently masking the interrupt requests from the various devices that collectively can activate the core's external input interrupt request input.

External Input interrupts are enabled if:

$$(EPCR[EXTGS] = 0) \& ((MSR[GS] = 1) \mid (MSR[EE] = 1))$$

or

$$(EPCR[EXTGS] = 1) \& (MSR[GS] = 1) \& (MSR[EE] = 1)$$

Note: MSR[EE] also enables other interrupts. See *Table 7-3 Interrupt and Exception Types* on page 306.

When an external input interrupt occurs, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x0A0.

If the interrupt is directed to the guest state (EPCR[EXTGS] = 1 and MSR[GS] = 1), GSRR0 and GSRR1 are set in place of SRR0, and SRR1 respectively, and instruction execution resumes at address GIVPR[IVP] || 0x0A0.

Save/Restore Register 0 (SRR0)	Set to the effective address of the next instruction to be executed.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	<p>CM set to EPCR[GICM] if the interrupt is directed to guest state; otherwise, it is set to EPCR[ICM].</p> <p>GS is left unchanged if the interrupt is directed to guest state; otherwise, it is set to zero.</p> <p>UCLE is left unchanged if the interrupt is directed to the guest state and MSRP[UCLEP] = 1; otherwise, it is set to 0.</p> <p>CE, ME, DE Unchanged.</p> <p>If the interrupt is directed to guest state, bits in the MSR corresponding to set bits in the MSRP register are left unchanged.</p> <p>All other MSR bits set to 0.</p>

Programming Note: Software is responsible for taking any actions that are required by the implementation to clear any External Input exception status (such that the External Input interrupt request input signal is deasserted) before reenabling MSR[EE], to avoid another, redundant External Input interrupt.

7.6.6 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 occurs if execution of any of the following instructions is attempted:

- An integer load or store instruction that references a data storage operand that is not aligned on an operand-sized boundary, when XUCR0[FLSTA] is 1.
- A load or store multiple instruction that is not word aligned (load and store multiple instructions are considered to reference word operands, and hence word-alignment is required). Load and store string instructions are considered to reference byte operands, and hence they cannot cause an alignment exception due to XUCR0[FLSTA] being 1, regardless of the target address alignment. See *Table 2-11 Operand Handling Dependent on Alignment* on page 87 for more information about operand alignments.
- A floating-point or AXU load or store instruction that references a data storage operand that is not aligned on an operand-sized boundary, when XUCR0[AFLSTA].
- An **icswx[.]** or **icswepx[.]** instruction specifying a coprocessor-request block (CRB) that is not located on a 128-byte boundary.
- A **dcbz** instruction that targets a memory page that is either write-through required or caching inhibited.
- A boundary is crossed between memory pages with different storage attributes. See *Section 6.5* on page 179 for a definition of the various storage attributes.
- An alignment interrupt occurs regardless of whether a **stwcx.** or **stdcx.** would have performed its store. The CR[CR0] is not updated.

Programming Note: The architecture does not support the use of an unaligned effective address by the **lwarx**, **ldarx**, **stwcx**, and **stdcx** instructions. If an alignment interrupt occurs due to the attempted execution of one of these instructions, the alignment interrupt handler must not attempt to emulate the instruction; instead, it should treat the instruction as a programming error.

When an alignment interrupt occurs, the processor suppresses the execution of the instruction causing the alignment exception, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address $IVPR[IVP] \parallel 0x0C0$.

Save/Restore Register 0 (SRR0)	Set to the effective address of the instruction causing the alignment interrupt.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM]. CE, ME, DE Unchanged. All other MSR bits set to 0.
Data Exception Address Register (DEAR)	Set to the effective address 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 alignment exception.
Exception Syndrome Register (ESR)	<p>FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise, set to 0.</p> <p>ST Set to 1 if the instruction causing the interrupt is a store, dcbz, or dcbi instruction; otherwise, set to 0.</p> <p>AXU Set to 1 if the instruction causing the interrupt is an auxiliary processor load or store; otherwise, set to 0.</p> <p>EPID Set to 1 if the instruction causing the interrupt is an external process ID instruction; otherwise, set to 0.</p> <p>All other defined ESR bits are set to 0.</p>

7.6.7 Program Interrupt

A program interrupt occurs when no higher priority exception exists, a program exception is presented to the interrupt mechanism, and—for the floating-point enabled form of program exception only—MSR[FE0,FE1] is nonzero. The A2O Core includes following types of program exception:

Illegal Instruction Exception

An illegal instruction exception occurs when execution is attempted of any of the following kinds of instructions:

- A reserved-illegal instruction. 
- An **mtspr** or **mfspr** that specifies an SPRN value with $SPRN_5 = 0$ (user-mode accessible) that represents an unimplemented Special Purpose Register.
- When MSR[PR] = 0 (supervisor-mode), an **mtspr** or **mfspr** that specifies an SPRN value with $SPRN_5 = 1$ (supervisor-mode accessible) that represents an unimplemented Special Purpose Register.

- A defined instruction that is not implemented within the A2O Core and that is not an auxiliary processor instruction.
- A defined auxiliary processor instruction that is not recognized by an attached auxiliary processor unit (or when no such auxiliary processor unit is attached).
- Any one of the following illegal forms of an ERAT or TLB instruction (see *Memory Management* on page 169 or *TLB Management Instructions* on page 476 for more information):
 - **eratwe** or **eratre** with $WS > 3$.
 - **eratwe** or **eratre** with $WS = 2$ in 64-bit mode ($MSR[CM] = 1$).
 - **tlbilx** with $T = 2$.
 - **tlbwe**, **tlbre**, **tlbsx**, **tlbsrx**, **tlbilx**, or **tlbivax** when no TLB is present ($CCR2[NOTLB] = 1$).
 - **erativax** with TLB present ($CCR2[NOTLB] = 0$).
 - **erativax** when $RS[60:63]$ contains an unsupported page size.
 - **eratwe**, **eratre**, or **eratsx** when not targeting an ERAT ($MMUCR0[TLBSEL] \neq (2 \text{ or } 3)$).
 - **tlbivax** or **tlbilx** with $T = 3$, when $MAS6[ISIZE]$ contains an unsupported page size.
 - **eratilx** with $T = 4, 5, 6,$ or 7 and $MMUCR1[ICTID] = 1$ and $MMUCR1[DCTID] = 1$.
 - **tlbre** when $MAS0[ATSEL] = 0$ or $MSR[GS] = 1$, and $MAS1[IND] = 0$ or $TLB0CFG[IND] = 0$, and $MAS1[TSIZE]$ contains an unsupported direct page size for the TLB.
 - **tlbre** when $MAS0[ATSEL] = 0$ or $MSR[GS] = 1$, $MAS1[IND] = 1$, $TLB0CFG[IND] = 1$, and $MAS1[TSIZE]$ contains an unsupported indirect page size for the TLB.
 - **tlbwe** when $MAS0[ATSEL] = 0$ or $MSR[GS] = 1$, $MAS1[IND] = 0$ or $TLB0CFG[IND] = 0$, and $MAS0[WQ] \neq 2$, and $MAS1[TSIZE]$ contains an unsupported direct page size for the TLB.
 - **tlbwe** when $MAS0[ATSEL] = 0$ or $MSR[GS] = 1$, and $MAS1[IND] = 1$, $TLB0CFG[IND] = 1$, and $MAS0[WQ] \neq 2$, and $MAS1[TSIZE]$ contains an unsupported indirect page size for the TLB.
 - **tlbwe** when $MAS0[ATSEL] = 0$ or $MSR[GS] = 1$, and $MAS0[WQ] \neq 2$, $MAS1[IND] = 1$, $TLB0CFG[IND] = 1$, and $MAS2[WIMGE]$ contains memory attributes that are not consistent with those specified in *Section 6.16.6 Hardware Page Table Storage Control Attributes*.
 - **tlbwe** when $MAS0[ATSEL] = 0$ or $MSR[GS] = 1$, and $MAS0[WQ] \neq 2$, $MAS1[IND] = 1$, $TLB0CFG[IND] = 1$, and an unsupported page size and sub-page size combination are contained in $MAS1[TSIZE]$ and $MAS3[SPSIZE]$.
 - **tlbwe** when $MAS0[ATSEL] = 1$, $MSR[GS] = 0$, $MAS0[WQ] = 0$ or 3 , and $MAS1[TSIZE]$ contains an unsupported page size for the LRAT.
 - **tlbwe** when $MAS0[ATSEL] = 1$, $MSR[GS] = 0$, and either $MAS0[HES] = 1$ or $MAS0[WQ] = 1$ or 2 .
- An instruction that is disabled:
 - **attn** and $CCR[EN_ATTN] = 0$.
 - **msgsnd** or **msgclr** and $CCR2[EN_PC] = 0$.
 - **icswx** or **icswepx** and $CCR2[EN_ICSWX] = 0$.
- An illegal form of other defined instructions:
 - store with update instruction with $RA = 0$.
 - load with update instruction and $RA = 0$ or $RA = RT$.

- **lswx** instruction, and RA or RB is in the range of registers to be loaded, including the case in which RA = 0, or RT = RA, or RT = RB.
- **lmw**, **lswi**, **lswx**, and RA is in the range of registers to be loaded.
- **lswx**, and RB is in the range of registers to be loaded.
- **sc** instruction with LEV > 1.

See *Instruction Categories* on page 82 for more information about the A2O Core support for defined and allocated instructions.

Privileged Instruction Exception

A privileged instruction exception occurs when 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 SPRN₅ = 1 (supervisor-mode accessible). A privileged instruction exception occurs regardless of whether or not the SPR referenced by the SPRN value is defined.

Unimplemented Operation Exception

An unimplemented operation exception occurs when an instruction that is microcoded is executed and CCR2[UCODE_DIS] = 1.

Trap Exception

A trap exception occurs when any of the conditions specified in a **tw**, **twi**, or **td**, **tdi** instruction are met. However, if trap debug events are enabled (DBCR0[TRAP] = 1), internal debug mode is enabled (DBCR0[IDM] = 1), and debug interrupts are enabled (MSR[DE] = 1), then a trap exception causes a debug interrupt to occur rather than a program interrupt.

See *Debug Facilities* on page 389 for more information about Trap debug events.

Floating-Point Enabled Exception

A floating-point enabled exception occurs when the execution or attempted execution of a defined floating-point instruction causes FPSCR[FEX] to be set to 1 in an attached floating-point unit. FPSCR[FEX] is the Floating-Point Enabled Exception Summary bit in the Floating-Point Status and Control Register.

If MSR[FE0,FE1] is nonzero when the floating-point enabled exception is presented to the interrupt mechanism, a program interrupt occurs, and the interrupt processing registers are updated as described in the following list. If MSR[FE0,FE1] are both 0, however, then a program interrupt does *not* occur and the instruction associated with the exception executes according to the definition of the floating-point unit. If and when MSR[FE0,FE1] are subsequently set to a nonzero value and the Floating-Point Enabled exception is still being presented to the interrupt mechanism (that is, FPSCR[FEX] is still set), then a “delayed” program interrupt occurs, updating the interrupt processing registers as described in the following list.

See *Synchronous, Imprecise Interrupts* on page 279 for more information about this special form of “delayed” Floating-Point Enabled exception.

Auxiliary Processor Enabled Exception

An auxiliary processor enabled exception can occur due to the execution or attempted execution of an instruction that is recognized and supported by an attached auxiliary processor. The cause of such an exception is implementation-dependent.

When a program interrupt occurs, the processor suppresses the execution of the instruction causing the program exception (for all cases except the “delayed” form of floating-point enabled exception previously described), the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address $IVPR[IVP] \parallel 0x0E0$.

Save/Restore Register 0 (SRR0) Set to the effective address of the instruction causing the Program interrupt, for all cases except the “delayed” form of Floating-Point Enabled exception described above.

For the special case of the delayed Floating-Point Enabled exception, where the exception was already being presented to the interrupt mechanism at the time $MSR[FE0,FE1]$ was changed from 0 to a nonzero value, SRR0 is set to the address of the instruction that would have executed after the MSR-changing instruction. If the instruction that set $MSR[FE0,FE1]$ was **rfi**, **rftci**, or **rfmci**, SRR0 is set to the address to which the **rftci**, **rftci**, or **rfmci** was returning, and not to the address of the instruction that was sequentially after the **rftci**, **rftci**, or **rfmci**.

Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to $EPCR[ICM]$. CE, ME, DE Unchanged. All other MSR bits set to 0.
Exception Syndrome Register (ESR)	<p>PIL Set to 1 for an illegal instruction exception; otherwise, set to 0.</p> <p>PPR Set to 1 for a privileged instruction exception; otherwise, set to 0.</p> <p>PTR Set to 1 for a trap exception; otherwise, set to 0.</p> <p>FP Set to 1 if the instruction causing the interrupt is a floating-point instruction; otherwise, set to 0.</p> <p>AXU Set to 1 if the instruction causing the interrupt is an auxiliary processor instruction; otherwise, set to 0.</p> <p>PIE Set to 1 if a “delayed” form of the floating-point enabled exception type of program interrupt; otherwise, set to 0. The setting of $ESR[PIE]$ to 1 indicates to the program interrupt handler that the interrupt was imprecise because it was caused by changing $MSR[FE0,FE1]$ and not directly by the execution of the floating-point instruction that caused the exception by setting $FPSCR[FEX]$. Thus, the program interrupt handler can recognize that SRR0 contains the address of the instruction after the MSR-changing instruction, and not the address of the instruction that caused the floating-point enabled exception.</p> <p>All other defined ESR bits are set to 0.</p>

7.6.8 Floating-Point Unavailable Interrupt

A floating-point unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction that is recognized by an attached floating-point unit, and $MSR[FP] = 0$.

When a floating-point unavailable interrupt occurs, the processor suppresses the execution of the instruction causing the floating-point unavailable exception, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address $IVPR[IVP] \parallel 0x100$.

Save/Restore Register 0 (SRR0)	Set to the effective address of the instruction causing the floating-point unavailable interrupt.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to $EPCR[ICM]$. CE, ME, DE Unchanged. All other MSR bits set to 0.

7.6.9 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.

When a system call interrupt occurs, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address $IVPR[IVP] \parallel 0x120$.

If the interrupt is directed to guest state ($MSR[GS] = 1$), $GSRR0$ and $GSRR1$ are set in place of $SRR0$ and $SRR1$ respectively, and instruction execution resumes at address $GIVPR[IVP] \parallel 0x120$.

Save/Restore Register 0 (SRR0)	Set to the effective address of the instruction after the system call instruction.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to $EPCR[GICM]$ if the interrupt is directed to guest state; otherwise, it is set to $EPCR[ICM]$. GS is left unchanged if the interrupt is directed to guest state; otherwise, it is set to zero. UCLE is left unchanged if the interrupt is directed to the guest state and $MSRP[UCLEP] = 1$; otherwise, it is set to 0. CE, ME, DE Unchanged. All other MSR bits set to 0. Bits in the MSR corresponding to set bits in the MSRP register are left unchanged.

7.6.10 Auxiliary Processor Unavailable Interrupt

An auxiliary processor unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute an auxiliary processor instruction that is not implemented within the A2O Core but which is recognized by an attached auxiliary processor, and auxiliary processor instruction processing is not enabled (CCR2[AP] = 0).

When an auxiliary processor unavailable interrupt occurs, the processor suppresses the execution of the instruction causing the auxiliary processor unavailable exception, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x140.

Save/Restore Register 0 (SRR0)	Set to the effective address of the instruction causing the auxiliary processor unavailable interrupt.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM]. CE, ME, DE, ICM Unchanged. All other MSR bits are set to 0.

7.6.11 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)). See *Timer Facilities* on page 371 for more information about decrementer exceptions.

Note: MSR[EE] also enables other interrupts. See *Table 7-3 Interrupt and Exception Types* on page 306.

When a decrementer interrupt occurs, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x160.

Save/Restore Register 0 (SRR0)	Set to the effective address of the next instruction to be executed.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM]. CE, ME, DE Unchanged. All other MSR bits set to 0.

Programming Note: Software is responsible for clearing the decrementer exception status by writing to TSR[DIS] before reenabling MSR[EE] to avoid another, redundant decrementer interrupt.

7.6.12 Guest Decrementer Interrupt

A guest decrementer interrupt occurs when no higher priority exception exists, a guest decrementer exception exists (GTSR[DIS] = 1), and the interrupt is enabled (GTCCR[DIE] = 1 and (MSR[EE] = 1 or MSR[GS] = 1)). See *Timer Facilities* on page 371 for more information about decrementer exceptions.

Note: MSR[EE] also enables other interrupts. See *Table 7-3 Interrupt and Exception Types* on page 306.

When a decremter interrupt occurs, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x160.

Guest Save/Restore Register 0 (GSRR0) Set to the effective address of the next instruction to be executed.

Guest Save/Restore Register 1 (GSRR1) Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR) CM set to EPCR[ICM].
 CE, ME, DE Unchanged.
 All other MSR bits set to 0.

Programming Note: Software is responsible for clearing the guest decremter exception status by writing to GTSR[DIS] before reenabling MSR[EE] to avoid another, redundant guest decremter interrupt.

7.6.13 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)). See *Timer Facilities* on page 371 for more information about fixed interval timer exceptions.

Note: MSR[EE] also enables the external input and all decremter interrupts.

When a fixed interval timer interrupt occurs, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x180.

Save/Restore Register 0 (SRR0) Set to the effective address of the next instruction to be executed.
 Save/Restore Register 1 (SRR1) Set to the contents of the MSR at the time of the interrupt.
 Machine State Register (MSR) CM set to EPCR[ICM].
 CE, ME, DE Unchanged.
 All other MSR bits set to 0.

Programming Note: Software is responsible for clearing the fixed interval timer exception status by writing to TSR[FIS] before reenabling MSR[EE] to avoid another, redundant fixed interval timer interrupt.

7.6.14 Guest Fixed-Interval Timer Interrupt

A guest fixed-interval timer interrupt occurs when no higher priority exception exists, a guest fixed-interval timer exception exists (GTSR[FIS] = 1), and the interrupt is enabled (GTCR[FIE] = 1 and (MSR[EE] = 1 or MSR[GS] = 1)). See *Timer Facilities* on page 371 for more information about fixed interval timer exceptions.

Note: MSR[EE] also enables the external input and all decremter interrupts.

When a guest fixed interval timer interrupt occurs, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x180.

Guest Save/Restore Register 0 (GSRR0) Set to the effective address of the next instruction to be executed.

Guest Save/Restore Register 1 (GSRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM]. CE, ME, DE Unchanged. All other MSR bits set to 0.

Programming Note: Software is responsible for clearing the guest fixed interval timer exception status by writing to GTSR[FIS] before reenabling MSR[EE] to avoid another, redundant guest fixed interval timer interrupt.

7.6.15 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)). See *Timer Facilities* on page 371 for more information about watchdog timer exceptions.

Note: MSR[CE] also enables the critical input interrupt.

When a watchdog timer interrupt occurs, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x1A0.

Critical Save/Restore Register 0 (CSRR0)	Set to the effective address of the next instruction to be executed.
Critical Save/Restore Register 1 (CSRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM]. E Unchanged. All other MSR bits set to 0.

Programming Note: Software is responsible for clearing the watchdog timer exception status by writing to TSR[WIS] before reenabling MSR[CE] to avoid another, redundant watchdog timer interrupt.

7.6.16 Guest Watchdog Timer Interrupt

A guest watchdog timer interrupt occurs when no higher priority exception exists, a watchdog timer exception exists (GTSR[WIS] = 1), and the interrupt is enabled (GTCR[WIE] = 1 and (MSR[CE] = 1 or MSR[GS] = 1)). See *Timer Facilities* on page 371 for more information about watchdog timer exceptions.

Note: MSR[CE] also enables the critical input interrupt.

When a guest watchdog timer interrupt occurs, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x1A0.

Critical Save/Restore Register 0 (CSRR0)	Set to the effective address of the next instruction to be executed.
--	--

Critical Save/Restore Register 1 (CSRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM]. E Unchanged. All other MSR bits set to 0.

Programming Note: Software is responsible for clearing the guest watchdog timer exception status by writing to GTSR[WIS] before reenabling MSR[CE] to avoid another, redundant guest watchdog timer interrupt.

7.6.17 Data TLB Error Interrupt

A data TLB error interrupt *might* occur when no higher priority exception exists and a data TLB miss exception is presented to the interrupt mechanism. A data TLB miss exception occurs when a load, store, **icbi**, **icbt**, **dcbst**, **dcbf**, **dcbz**, **dcbi**, **dcbt**, or **dcbtst** instruction attempts to access a virtual address for which a valid TLB entry does not exist. See *Memory Management* on page 169 for more information about the TLB.

Programming Note: The instruction cache management instructions **icbi** and **icbt** are treated as loads from the addressed byte with respect to address translation and protection; therefore, use MSR[DS] rather than MSR[IS] as part of the calculated virtual address when searching the TLB to determine translation for their target storage address. Instruction TLB miss exceptions are associated with the *fetching* of instructions, not with the *execution* of instructions. Data TLB miss exceptions are associated with the *execution* of instruction cache management instructions, as well as with the execution of load, store, and data cache management instructions.

A data TLB miss exception occurs regardless of whether an **stwcx.** or **stdcx.** would have performed its store. The CR[CR0] is not updated.

If a data TLB Miss exception occurs on any of the following instructions, the instruction is treated as a no-op, and a data TLB error interrupt does not occur:

- **lswx** or **stswx** with a length of zero (although the target register of **lswx** is undefined)
- **icbt**
- **dcbt**, **dcbtep**
- **dcbtst**, **dcbtstep**

For all other instructions, if a data TLB miss exception occurs, execution of the instruction causing the exception is suppressed, a data TLB error interrupt is generated, the interrupt processing registers are updated as indicated the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x1C0.

If the interrupt is directed to guest state (EPCR[DTLBGS] = 1 and MSR[GS] = 1), GSRR0, GSRR1, GDEAR, and GESR, are set in place of SRR0, SRR1, DEAR, and ESR respectively, and instruction execution resumes at address IVPR[IVP] || 0x1C0.

Save/Restore Register 0 (SRR0)	Set to the effective address of the instruction causing the data TLB error interrupt.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)	<p>CM set to EPCR[GICM] if the interrupt is directed to guest state; otherwise, it is set to EPCR[ICM].</p> <p>GS is left unchanged if the interrupt is directed to guest state; otherwise, it is set to zero.</p> <p>UCLE is left unchanged if the interrupt is directed to the guest state and MSRP[UCLEP] = 1; otherwise, it is set to 0.</p> <p>CE, ME, DE Unchanged.</p> <p>If the interrupt is directed to guest state, bits in the MSR corresponding to set bits in the MSRP register are left unchanged. All other MSR bits set to 0.</p>
Data Exception Address Register (DEAR)	<p>If the instruction causing the data TLB miss exception does so with respect to the memory page targeted by the initial effective address calculated by the instruction, then the DEAR is set to this calculated effective address. On the other hand, if the data TLB miss exception only occurs due to the instruction causing the exception crossing a memory page boundary, in that the missing TLB entry is for the page accessed after crossing the boundary, then the DEAR is set to the address of the first byte within that page.</p> <p>As an example, consider a misaligned load word instruction that targets effective address 0x00000FFF, and that the page containing that address is a 4 KB page. The load word thus crosses the page boundary and attempts to access the next page starting at address 0x00001000. If a valid TLB entry does not exist for the first page, the DEAR is set to 0x00000FFF. On the other hand, if a valid TLB entry <i>does</i> exist for the first page, but not for the second, then the DEAR is set to 0x00001000. Furthermore, the load word instruction in this latter scenario has been partially executed (see <i>Partially Executed Instructions</i> on page 283).</p>
Exception Syndrome Register (ESR)	<p>FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise, set to 0.</p> <p>ST Set to 1 if the instruction causing the interrupt is a store, dcbz, or dcbi instruction; otherwise, set to 0.</p> <p>AXU Set to 1 if the instruction causing the interrupt is an auxiliary processor load or store; otherwise, set to 0.</p> <p>EPID Set to 1 if the instruction causing the interrupt is an external process ID instruction; otherwise, set to 0.</p> <p>All other defined ESR bits are set to 0.</p>

7.6.18 Instruction TLB Error Interrupt

An instruction TLB error interrupt occurs when no higher priority exception exists and an instruction TLB miss exception is presented to the interrupt mechanism. Note that although an instruction TLB miss exception can occur during an attempt to *fetch* an instruction, such an exception is not actually presented to the interrupt

mechanism until an attempt is made to *execute* that instruction. An instruction TLB miss exception occurs when an instruction fetch attempts to access a virtual address for which a valid TLB entry does not exist. See *Memory Management* on page 169 for more information about the TLB.

When an instruction TLB error interrupt occurs, the processor suppresses the execution of the instruction causing the instruction TLB miss exception, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address $IVPR[IVP] \parallel 0x1E0$.

If the interrupt is directed to guest state ($EPCR[ITLBGS] = 1$ and $MSR[GS] = 1$), $GSRR0$ and $GSRR1$ are set in place of $SRR0$ and $SRR1$ respectively, and instruction execution resumes at address $IVPR[IVP] \parallel 0x1E0$.

Save/Restore Register 0 (SRR0)	Set to the effective address of the instruction causing the instruction TLB error interrupt.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to $EPCR[ICM]$. CE, ME, DE Unchanged. All other MSR bits set to 0.

7.6.19 Vector Unavailable Interrupt

The vector unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a vector instruction that is recognized by an attached vector unit, and $MSR[SPV] = 0$.

When a vector unavailable interrupt occurs, the processor suppresses the execution of the instruction causing the vector unavailable exception, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address $IVPR[IVP] \parallel 0x200$.

Save/Restore Register 0 (SRR0) Set to the effective address of the instruction causing the Auxiliary Processor Unavailable interrupt.

Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to $EPCR[ICM]$. CE, ME, DE, ICM Unchanged. All other MSR bits set to 0.

Exception Syndrome Register (ESR) SPV Set to 1.

7.6.20 Debug Interrupt

A debug interrupt occurs when no higher priority exception exists, a debug exception exists in the Debug Status Register (DBSR), the processor is in internal debug mode ($DBCR0[IDM] = 1$), and debug interrupts are enabled ($MSR[DE] = 1$). A debug exception occurs when a debug event causes a corresponding bit in the DBSR to be set.

There are several types of debug exception, as follows:

Instruction Address Compare (IAC) Exception

An IAC debug exception occurs when execution is attempted of an instruction whose address matches the IAC conditions specified by the various debug facility registers. This exception can occur regardless of debug mode, and regardless of the value of MSR[DE].

Data Address Compare (DAC) Exception

A DAC debug exception occurs when the DVC mechanism is not enabled, and execution is attempted of a load, store, **icbi**, **icbt**, **dcbst**, **dcbf**, **dcbz**, **dcbi**, **dcbt**, or **dcbtst** instruction whose target storage operand address matches the DAC conditions specified by the various debug facility registers. This exception can occur regardless of debug mode and regardless of the value of MSR[DE].

Programming Note: The instruction cache management instructions **icbi** and **icbt** are treated as loads from the addressed byte with respect to debug exceptions. IAC debug exceptions are associated with the *fetching* of instructions not with the *execution* of instructions. DAC debug exceptions are associated with the *execution* of instruction cache management instructions, as well as with the execution of load, store, and data cache management instructions.

Data Value Compare (DVC) Exception

A DVC debug exception occurs when execution is attempted of a load, store, or **dcbz** instruction whose target storage operand address matches the DAC and DVC conditions specified by the various debug facility registers. This exception can occur regardless of debug mode and regardless of the value of MSR[DE].

Branch Taken (BRT) Exception

A BRT debug exception occurs when BRT debug events are enabled (DBCR0[BRT] = 1) and execution is attempted of a branch instruction for which the branch conditions are met. This exception cannot occur in internal debug mode when MSR[DE] = 0 unless external debug mode enabled.

Trap (TRAP) Exception

A TRAP debug exception occurs when TRAP debug events are enabled (DBCR0[TRAP] = 1) and execution is attempted of a **tw** or **twi** instruction that matches any of the specified trap conditions. This exception can occur regardless of debug mode and regardless of the value of MSR[DE].

Return (RET) Exception

An RET debug exception occurs when RET debug events are enabled (DBCR0[RET] = 1) and execution is attempted of an **rfi** instruction. For **rfi**, the RET debug exception can occur regardless of debug mode and regardless of the value of MSR[DE].

Instruction Complete (ICMP) Exception

An ICMP debug exception occurs when ICMP debug events are enabled (DBCR0[ICMP] = 1) and execution of any instruction is completed. This exception cannot occur in internal debug mode when MSR[DE] = 0 unless external debug mode enabled.

Programming Note: If ICMP debug events are enabled and debug interrupts (previously disabled) are subsequently enabled, the ICMP debug interrupt occurs sometime after the instruction that enabled the debug interrupt or on the instruction directly following a context synchronizing event. If the instruction that enabled the debug interrupt was a context synchronizing instruction, the ICMP debug interrupt occurs on the next instruction.

Interrupt (IRPT) Exception

An IRPT debug exception occurs when IRPT debug events are enabled (DBCR0[IRPT] = 1) and a base class interrupt occurs.

Unconditional Debug Event (UDE) Exception



A UDE debug exception occurs when an unconditional debug event is signaled over the [JTAG](#) interface to the A2O Core. This exception can occur regardless of debug mode and regardless of the value of MSR[DE].

Instruction Value Compare (IVC) Exception

An IVC debug exception occurs when IVC events are enabled (DBCR3[IVC] = 1) and execution is attempted of an instruction that matches the following expression:

$$(IMMR[MASK] \& \text{instruction_opcode}) = (IMMR[MASK] \& IMR[MATCH])$$

This exception can occur regardless of debug mode, and regardless of the value of MSR[DE].

There are two debug modes supported by the A2O Core: internal debug mode and external debug mode. Debug exceptions and interrupts are affected by the debug modes that are enabled at the time of the debug exception. Debug interrupts occur only when internal debug mode is enabled, although it is possible for external debug mode to be enabled as well. The remainder of this section assumes that internal debug mode is enabled and that external debug mode is not enabled at the time of a debug exception.

See *Debug Facilities* on page 389 for more information about the different debug modes and the behavior of each of the Debug exception types when operating in each of the modes.

Programming Note: It is a programming error for software to enable internal debug mode (by setting DBCR0[IDM] to 1) while debug exceptions are already present in the DBSR. Software must first clear all DBSR debug exception status fields (that is, all fields except IDE, MRR) before setting DBCR0[IDM] to 1.

A DAC or DVC debug exception occurs regardless of whether a **stwcx.** or **stdcx.** would have performed its store. The CR[CR0] is not updated.

If a DAC exception occurs on an **lswx** or **stswx** with a length of zero, the instruction is treated as a no-op, the debug exception is not recorded in the DBSR, and a debug interrupt does not occur.

If a DAC exception occurs on an **icbt**, **dcbt**, or **dcbtst** instruction that is being no-op'ed for some other reason (either the referenced cache block is in a caching inhibited memory page or a data storage or data TLB miss exception occurs), then the debug exception is not recorded in the DBSR and a debug interrupt does not occur. On the other hand, if the **icbt**, **dcbt**, or **dcbtst** instruction is not being no-op'ed for one of these other reasons, the DAC debug exception does occur and is handled in the same fashion as other DAC debug exceptions.

For all other cases, when a debug exception occurs, it is immediately presented to the interrupt handling mechanism. A debug interrupt occurs immediately if MSR[DE] is 1, and the interrupt processing registers are updated as described in the following list. If MSR[DE] is 0, however, the exception condition remains set in the DBSR. If and when MSR[DE] is subsequently set to 1, and the exception condition is still present in the DBSR, a “delayed” debug interrupt then occurs either as a synchronous, imprecise interrupt or as an asynchronous interrupt, depending on the type of debug exception.

When a debug interrupt occurs, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged) and instruction execution resumes at address IVPR[IVP] || 0x040.

Critical Save/Restore Register 0 (CSRR0)

For debug exceptions that occur while debug interrupts are enabled (MSR[DE] = 1), CSRR0 is set as follows:

- For IAC, IVC, BRT, TRAP, and RET debug exceptions, set to the address of the instruction causing the debug interrupt. Execution of the instruction causing the debug exception is suppressed, and the interrupt is synchronous and precise.
- For DAC and DVC store class debug exceptions, set to the address of the instruction causing the debug interrupt. Execution of the instruction causing the debug exception is suppressed, and the interrupt is synchronous and precise.
- For DVC load class debug exceptions, set to the effective address of the excepting instruction or to the effective address of some subsequent instruction. Execution of the instruction pointed to by CSRR0 is suppressed, and the interrupt is synchronous and imprecise.
- For ICMP debug exceptions, set to the address of the next instruction to be executed (the instruction after the one whose completion caused the ICMP debug exception). The interrupt is synchronous and precise.
- For IRPT debug exceptions, set to the address of the first instruction in the interrupt handler associated with the interrupt type that caused the IRPT debug exception. The interrupt is asynchronous.
- For UDE debug exceptions, set to the address of the instruction that would have executed next if the debug interrupt had not occurred. The interrupt is asynchronous.

For all debug exceptions that occur while debug interrupts are disabled (MSR[DE] = 0), the debug interrupt is delayed and occurs if and when MSR[DE] is again set to 1, assuming the debug exception status is still set in the DBSR. If the debug interrupt occurs in this fashion, CSRR0 is set to the address of the instruction after the one that set MSR[DE]. If the instruction that set MSR[DE] was **rfi**, **rftci**, **rftgi**, or **rftmci**, then CSRR0 is set to the address to which the **rftci**, **rftci**, **rftgi**, or **rftmci** was returning, and not to the address of the instruction that was sequentially after the **rftci**, **rftci**, **rftgi**, or **rftmci**. The interrupt is either synchronous and imprecise or asynchronous, depending on the type of debug exception, as follows:

- For IAC and RET debug exceptions, the interrupt is synchronous and imprecise.
- For BRT debug exceptions, this scenario cannot occur. BRT debug exceptions are not recognized when MSR[DE] = 0 if operating in internal debug mode.
- For TRAP debug exceptions, the debug interrupt is synchronous and imprecise. However, under these conditions (TRAP debug exception occurring while MSR[DE] is 0), the attempted execution of the trap instruction for which one or more of the trap conditions is met itself leads to a trap exception type of program interrupt. The corresponding debug interrupt that occurs later if and when debug interrupts are enabled is *in addition* to the program interrupt.
- For ICMP debug exceptions, this scenario cannot occur in this fashion. ICMP debug exceptions are not recognized when MSR[DE] = 0 if operating in internal debug mode. However, a similar scenario can occur when MSR[DE] is 1 at the time of the ICMP debug exception, but the instruction whose completion is causing the exception is itself setting MSR[DE] to 0. This scenario is described in the subsection on the ICMP debug exception for which MSR[DE] is 1 at the time of the exception. In that scenario, the interrupt is synchronous and imprecise.
- For IRPT and UDE debug exceptions, the interrupt is asynchronous.

Critical Save/Restore Register 1 (CSRR1)

Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)

CM set to EPCR[ICM].

ME unchanged.

All other MSR bits set to 0.

7.6.21 Processor Doorbell Interrupt

A processor doorbell interrupt occurs when no higher priority exception exists, a processor doorbell exception is present, and the interrupt is enabled (MSR[GS] = 1 or MSR[EE] = 1). Processor doorbell exceptions are generated when DBELL messages (see *Processor Messages* on page 341) are received and accepted by the processor.

Note: MSR[EE] also enables other interrupts. See *Table 7-3 Interrupt and Exception Types* on page 306.

The interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged) and instruction execution resumes at address IVPR[IVP] || 0x280.

Save/Restore Register 0 (SRR0)

Set to the effective address of the next instruction to be executed.

Save/Restore Register 1 (SRR1)

Set to the effective address of the next instruction to be executed.

Machine State Register (MSR)	CM set to EPCR[ICM]. CE, ME, DE Unchanged. All other defined MSR bits set to 0.
------------------------------	--

7.6.22 Processor Doorbell Critical Interrupt

A processor doorbell critical interrupt occurs when no higher priority exception exists, a processor doorbell critical exception is present, and the interrupt is enabled ($\text{MSR}[\text{CE}] = 1$ or $\text{MSR}[\text{GS}] = 1$). Processor doorbell critical exceptions are generated when DBELL_CRIT messages (see *Processor Messages* on page 341) are received and accepted by the processor.

The interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged) and instruction execution resumes at address $\text{IVPR}[\text{IVP}] \parallel 0x2A0$.

Critical Save/Restore Register 0 (CSRR0)	Set to the effective address of the next instruction to be executed.
Critical Save/Restore Register 1 (CSRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM]. ME unchanged. All other defined MSR bits set to 0.

7.6.23 Guest Processor Doorbell Interrupt

A guest processor doorbell interrupt occurs when no higher priority exception exists, a guest processor doorbell exception is present, and the interrupt is enabled ($\text{MSR}[\text{GS}] = 1$ and $\text{MSR}[\text{EE}] = 1$). Guest processor doorbell exceptions are generated when G_DBELL messages (see *Processor Messages* on page 341) are received and accepted by the processor.

Programming Note: Guest processor doorbell interrupts are used by the hypervisor to be notified when the guest operating system has set $\text{MSR}[\text{EE}]$ to 1. This allows the hypervisor to reflect base class interrupts to the guest at a time when the guest is ready to accept them ($\text{MSR}[\text{GS}] = 1$ and $\text{MSR}[\text{EE}] = 1$).

Note: $\text{MSR}[\text{EE}]$ also enables other interrupts. See *Table 7-3 Interrupt and Exception Types* on page 306.

The interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged) and instruction execution resumes at address $\text{IVPR}[\text{IVP}] \parallel 0x2C0$.

Guest Save/Restore Register 0 (GSRR0)	Set to the effective address of the next instruction to be executed.
Guest Save/Restore Register 1 (GSRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM]. CE, ME, DE Unchanged. All other defined MSR bits set to 0.

7.6.24 Guest Processor Doorbell Critical Interrupt

A guest processor doorbell critical interrupt occurs when no higher priority exception exists, a guest processor doorbell critical exception is present, and the interrupt is enabled ($MSR[GS] = 1$ and $MSR[CE] = 1$). Guest processor doorbell critical exceptions are generated when `G_DBELL_CRIT` messages (see *Processor Messages* on page 341) are received and accepted by the processor.

Programming Note: Guest processor doorbell critical interrupts are used by the hypervisor to be notified when the guest operating system has set $MSR[CE]$ to 1. This allows the hypervisor to reflect critical class interrupts to the guest at a time when the guest is ready to accept them ($MSR[GS] = 1$ and $MSR[CE] = 1$).

Programming Note: Guest processor doorbell critical interrupts and guest processor doorbell machine check interrupts share the same IVO. Hypervisor software can differentiate between the two interrupts by comparing whether CE or ME is set in CSRR1 and which interrupt class is to be reflected.

The interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged) and instruction execution resumes at address $IVPR[IVP] || 0x2E0$.

Critical Save/Restore Register 0 (CSRR0)	Set to the effective address of the next instruction to be executed.
Critical Save/Restore Register 1 (CSRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to $EPCR[ICM]$. ME unchanged. All other defined MSR bits set to 0.

7.6.25 Guest Processor Doorbell Machine Check Interrupt

A guest processor doorbell machine check interrupt occurs when no higher priority exception exists, a guest processor doorbell machine check exception is present, and the interrupt is enabled ($MSR[GS] = 1$ and $MSR[ME] = 1$). Guest processor doorbell machine check exceptions are generated when `G_DBELL_MC` messages (see *Processor Messages* on page 341) are received and accepted by the processor.

Programming Note: Guest processor doorbell machine check interrupts are used by the hypervisor to be notified when the guest operating system has set $MSR[ME]$ to 1. This allows the hypervisor to reflect machine check class interrupts to the guest at a time when the guest is ready to accept them ($MSR[GS] = 1$ and $MSR[ME] = 1$).

Programming Note: Guest processor doorbell critical interrupts and guest processor doorbell machine check interrupts share the same IVO. Hypervisor software can differentiate between the two interrupts by comparing whether CE or ME is set in CSRR1 and which interrupt class is to be reflected.

The interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged) and instruction execution resumes at address $IVPR[IVP] || 0x2E0$.

Critical Save/Restore Register 0 (CSRR0)	Set to the effective address of the next instruction to be executed.
Critical Save/Restore Register 1 (CSRR1)	Set to the contents of the MSR at the time of the interrupt.

Machine State Register (MSR)	CM set to EPCR[ICM]. ME unchanged. All other defined MSR bits set to 0.
------------------------------	---

7.6.26 Embedded Hypervisor System Call Interrupt

An embedded hypervisor system call interrupt occurs when no higher priority exception exists and a system call (**sc**) instruction with LEV = 1 is executed.

The interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged), and instruction execution resumes at address IVPR[IVP] || 0x300.

Save/Restore Register 0 (SRR0)	Set to the effective address of the instruction <i>after</i> the sc instruction.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM]. CE, ME,DE Unchanged. All other defined MSR bits set to 0.

7.6.27 Embedded Hypervisor Privilege Interrupt

An embedded hypervisor privilege interrupt occurs when no higher priority exception exists and an embedded hypervisor privilege exception is presented to the exception mechanism.

An embedded hypervisor privilege exception occurs when MSR[GS] = 1 and MSR[PR] = 0 and execution is attempted of any of the following instructions:

- A hypervisor privileged instruction (see *Section 2.12.1 Privileged Instructions*)
- An **mtspr** or **mf spr** instruction that specifies an SPR that is hypervisor privileged
- A **tlbwe**, **tlbsrx.**, **tlbwec.**, or **tlbilx** instruction and EPCR[DGTM] = 1
- A cache locking instruction and MSRP[UCLEP] = 1
- A **tlbwe** instruction when TLB0CFG[GTWE] = 0
- A **tlbwe** instruction when MMUCFG[LRAT] = 0
- A **tlbwe** instruction that attempts to write a TLB entry that has IPROT = 1 and MAS0[WQ] = 0 or 3
- A **tlbwe** instruction that attempts to write a TLB entry that has IPROT = 1, MAS0[WQ] = 1, and the executing thread holds a matching TLB-reservation
- A **tlbwe** instruction when MAS1[IPROT] = 1 and MAS0[WQ] = 0 or 3
- A **tlbwe** instruction when MAS1[IPROT] = 1, MAS0[WQ] = 1, and the executing thread holds a matching TLB-reservation
- A **tlbwe** instruction when MAS0[HES] = 0 and MAS0[WQ] != 2

An embedded hypervisor privilege exception also occurs when execution is attempted of an **ehpriv** instruction, regardless of the state of the processor.

The interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged) and instruction execution resumes at address $IVPR[IVP] \parallel 0x320$.

Save/Restore Register 0 (SRR0)	Set to the effective address of the instruction causing the embedded hypervisor privilege interrupt.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CMset to EPCR[ICM]. CE, ME, DE Unchanged All other defined MSR bits set to 0.
EHEIR	Set to a copy of the instruction that caused the interrupt.

7.6 LRAT Error Interrupt

An **LRAT** error interrupt occurs when no higher priority exception exists and an LRAT miss exception is presented to the interrupt mechanism. An LRAT miss exception is caused by any of the following:

- A **tlbwe** instruction is executed with $MSR[GS] = 1$, $MSR[PR] = 0$, $MMUCFG[LRAT] = 1$, $MAS0[WQ] = 0$ or 3 , $MAS1[V] = 1$, and the logical page number (RPN specified by MAS7 and MAS3 and page size specified by MAS1[TSIZE]) does not match any valid entry in the LRAT or matches multiple valid entries in the LRAT.
- A **tlbwe** instruction is executed with $MSR[GS] = 1$, $MSR[PR] = 0$, $MMUCFG[LRAT] = 1$, $MAS0[WQ] = 1$ and the executing thread holds a matching TLB-reservation, $MAS1[V] = 1$, and the logical page number (RPN specified by MAS7 and MAS3 and page size specified by MAS1[TSIZE]) does not match any valid entry in the LRAT or matches multiple valid entries in the LRAT.
- A page table translation is performed when $MSR[GS] = 1$, $MMUCFG[LRAT] = 1$, $PTE[V] = 1$, and the logical page number (RPN based on PTE[ARPN] and page size specified by PTE[PS]) does not match any valid entry in the LRAT or matches multiple valid entries in the LRAT.

When an LRAT error interrupt occurs, the processor suppresses the execution of the instruction causing the LRAT error interrupt, and the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged) and instruction execution resumes at address $IVPR[IVP] \parallel 0x340$.

Save/Restore Register 0 (SRR0)	Set to the effective address at the time of the interrupt.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM]. CE, ME, DE unchanged. All other defined Machine State Register bits set to 0.
Data Exception Address Register (DEAR)	If the LRAT error interrupt occurred for a page table translation, set to the effective address 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 LRAT miss exception. Otherwise, undefined.
Exception Syndrome Register (ESR)	FP Set to 1 if the instruction causing the interrupt is a floating-point load or store; otherwise, set to 0.

ST	Set to 1 if the instruction causing the interrupt is a store or “store-class” cache management instruction; otherwise, set to 0.
AP	Set to 1 if the instruction causing the interrupt is an auxiliary processor load or store; otherwise, set to 0. 
SPV	Set to 1 if the instruction causing the interrupt is an <u>SPE</u> operation or a vector operation; otherwise, set to 0.
PT	Set to 1 if the cause of the interrupt is an LRAT miss exception on a page table translation. Set to 0 if the cause of the interrupt is an LRAT miss exception on a tlbwe .
DATA	Set to 1 if the interrupt is due to is an LRAT miss resulting from a page table translation of a load, store or cache management operand address; otherwise, set to 0.
EPID	Set to 1 if the instruction causing the interrupt is an external process ID instruction; otherwise, set to 0.

All other defined ESR bits are set to 0.

7.6.29 User Decrementer Interrupt

A user decrementer interrupt occurs when no higher priority exception exists, a user decrementer exception exists ($\text{TSR}[\text{UDIS}] = 1$), and the interrupt is enabled ($\text{TCR}[\text{UDIE}] = 1$ and ($\text{MSR}[\text{EE}] = 1$ or $\text{MSR}[\text{GS}] = 1$)). See *Timer Facilities* on page 371 for more information about User Decrementer exceptions.

Note: $\text{MSR}[\text{EE}]$ also enables other interrupts. See *Table 7-3 Interrupt and Exception Types* on page 306.

When a user decrementer interrupt occurs, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged) and instruction execution resumes at address $\text{IVPR}[\text{IVP}] \parallel 0\text{x}800$.

Save/Restore Register 0 (SRR0)	Set to the effective address of the next instruction to be executed.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to $\text{EPCR}[\text{ICM}]$. CE, ME, DE Unchanged. All other MSR bits set to 0.

Programming Note: Software is responsible for clearing the decrementer exception status by writing to $\text{TSR}[\text{UDIS}]$ before re-enabling $\text{MSR}[\text{EE}]$ to avoid another, redundant user decrementer interrupt.

7.6.30 Performance Monitor Interrupt

A performance monitor interrupt occurs when no higher priority exception exists, a performance monitor exception exists, and the interrupt is enabled ($\text{MSR}[\text{EE}] = 1$ or $\text{MSR}[\text{GS}] = 1$).

Note: $\text{MSR}[\text{EE}]$ also enables other interrupts. See *Table 7-3 Interrupt and Exception Types* on page 306.

When a performance monitor interrupt occurs, the interrupt processing registers are updated as indicated in the following list (all registers not listed are unchanged) and instruction execution resumes at address $IVPR[IVP] \parallel 0x820$.

Save/Restore Register 0 (SRR0)	Set to the effective address of the next instruction to be executed.
Save/Restore Register 1 (SRR1)	Set to the contents of the MSR at the time of the interrupt.
Machine State Register (MSR)	CM set to EPCR[ICM].
	CE, ME, DE Unchanged.
	All other MSR bits set to 0.

Programming Note: Software is responsible for taking any actions that are required by the implementation to clear any Performance Monitor exception status (such that the Performance Monitor interrupt request input signal is de-asserted) before re-enabling MSR[EE], to avoid another, redundant Performance Monitor interrupt.

7.7 Processor 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 devices, including the sending processor, in the coherence domain in a reliable manner.

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.

To provide inter processor interrupt capability the following *doorbell* message types are defined:

- Processor Doorbell
- Processor Doorbell Critical
- Guest Processor Doorbell
- Guest Processor Doorbell Critical
- Guest Processor Doorbell Machine Check

A doorbell message causes an interrupt to occur on processors when the message is received and the processor determines through examination of the payload that the message should be accepted. The examination of the payload for this purpose is termed *filtering*. The acceptance of a doorbell message causes an exception to be generated on the accepting processor.

7.7.1 Processor Message Handling and Filtering

Processors filter, accept, and handle message types defined as follows. The message type is specified in the message and is determined by the contents of register $RB_{32:36}$ used as the operand in the **msgsnd** instruction. The message type is interpreted as follows:

Value	Description
-------	-------------

0	<p>Doorbell Interrupt (DBELL)</p> <p>A processor doorbell exception is generated 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 occurs when no higher priority exception exists, a processor doorbell exception</p>
---	--

exists, and the interrupt is enabled ($MSR_{EE} = 1$). If the Embedded.Hypervisor category is supported, the interrupt is enabled if ($MSR_{EE} = 1$ or $MSR_{GS} = 1$).

- 1 **Doorbell Critical Interrupt (DBELL_CRIT)**
A processor doorbell critical exception is generated 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 occurs when no higher priority exception exists, a processor doorbell critical exception exists, and the interrupt is enabled ($MSR_{CE} = 1$). If the Embedded.Hypervisor category is supported, the interrupt is enabled if ($MSR_{CE} = 1$ or $MSR_{GS} = 1$).
- 2 **Guest Doorbell Interrupt (G_DBELL)**
A guest processor doorbell exception is generated 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 occurs when no higher priority exception exists, a guest processor doorbell exception exists, and the interrupt is enabled ($MSR_{EE} = 1$ and $MSR_{GS} = 1$).
- 3 **Guest Doorbell Interrupt Critical (G_DBELL_CRIT)**
A guest processor doorbell critical exception is generated 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 occurs when no higher priority exception exists, a guest processor doorbell critical exception exists, and the interrupt is enabled ($MSR_{CE} = 1$ and $MSR_{GS} = 1$).
- 4 **Guest Doorbell Interrupt Machine Check (G_DBELL_MC)**
A guest processor doorbell machine check exception is generated 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 occurs when no higher priority exception exists, a guest processor doorbell machine check exception exists, and the interrupt is enabled ($MSR_{ME} = 1$ and $MSR_{GS} = 1$).

7.7.2 Doorbell Message Filtering

A processor receiving a DBELL message type filters the message and either ignores the message or accepts the message and generates a processor doorbell exception based on the payload and the state of the processor at the time the message is received.

The payload is specified in the message and is determined by the contents of register $RB_{37:63}$ used as the operand in the **msgsnd** instruction. The payload bits are defined in the following table.

Bits	Field Name	Description
37	BRDCAST	Broadcast. The message is accepted by all processors regardless of the value of the PIR register and the value of PIRTAG. 0 If the values of PIR and PIRTAG are equal, a processor doorbell exception is generated. 1 A processor doorbell exception is generated regardless of the value of PIRTAG and PIR.
38:41	Reserved	Reserved.
42:49	LPIDTAG	LPID Tag The contents of this field are compared with the contents of the LPIDR. If LPIDTAG = 0, it matches all LPIDR values in the LPIDR register.
50:63	PIRTAG	PIR Tag The contents of this field are compared with bits 50:63 of the PIR register.

If a DBELL message is received by a processor, the message is accepted if one of the following conditions exist:

- The message is for this partition ($payload_{LPIDTAG} = LPIDR$).

- The message is for all partitions ($\text{payload}_{\text{LPIDTAG}} = 0$).

If a DBELL message is accepted, a processor doorbell exception is generated if one of the following conditions exist:

- This is a broadcast message ($\text{payload}_{\text{BRDCAST}} = 1$).
- The message is intended for this processor ($\text{PIR}_{50:63} = \text{payload}_{\text{PIRTAG}}$).

The exception condition remains until a processor doorbell interrupt is taken or an **msgclr** instruction is executed on the receiving processor with a message type of DBELL. A change to any of the filtering criteria (such as, changing the PIR register) does not clear a pending processor doorbell exception.

DBELL messages are not cumulative. That is, if a DBELL message is accepted and the interrupt is pending because $\text{MSR}_{\text{EE}} = 0$, further DBELL messages that would be accepted are ignored until the processor doorbell exception is cleared by taking the interrupt or cleared by executing an **msgclr** with a message type of DBELL on the receiving processor.

7.7.3 Doorbell Critical Message Filtering

A processor receiving a DBELL_CRIT message type filters the message and either ignores the message or accepts the message and generates a processor doorbell critical exception based on the payload and the state of the processor at the time the message is received.

The payload is specified in the message and is determined by the contents of register $\text{RB}_{37:63}$ used as the operand in the **msgsnd** instruction. The payload bits are defined as follows.

Bit	Field Name	Description
37	BRDCAST	Broadcast The message is accepted by all processors regardless of the value of the PIR register and the value of PIRTAG. 0 If the values of PIR and PIRTAG are equal, a processor doorbell critical exception is generated. 1 A processor doorbell critical exception is generated regardless of the value of PIRTAG and PIR.
38:41	Reserved	Reserved
42:49	LPIDTAG	LPID Tag The contents of this field are compared with the contents of the LPIDR. If LPIDTAG = 0, it matches all values in the LPIDR register.
50:63	PIRTAG	PIR Tag The contents of this field are compared with bits 50:63 of the PIR register.

If a DBELL_CRIT message is received by a processor, the message is accepted if one of the following conditions exists:

- The message is for this partition ($\text{payload}_{\text{LPIDTAG}} = \text{LPIDR}$).
- The message is for all partitions ($\text{payload}_{\text{LPIDTAG}} = 0$).

If a DBELL_CRIT message is accepted, a processor doorbell critical exception is generated if one of the following conditions exists:

- This is a broadcast message ($\text{payload}_{\text{BRDCAST}} = 1$).
- The message is intended for this processor ($\text{PIR}_{50:63} = \text{payload}_{\text{PIRTAG}}$).

DBELL_CRIT messages are not cumulative. That is, if a DBELL_CRIT message is accepted and the interrupt is pending because $MSR_{CE} = 0$, further DBELL_CRIT messages that would be accepted are ignored until the processor doorbell critical exception is cleared by taking the interrupt or cleared by executing an **msgclr** with a message type of DBELL_CRIT on the receiving processor.

7.7.4 Guest Doorbell Message Filtering

A processor receiving a G_DBELL message type filters the message and either ignores the message or accepts the message and generates a guest processor doorbell critical exception based on the payload and the state of the processor at the time the message is received.

The payload is specified in the message and is determined by the contents of register $RB_{37:63}$ used as the operand in the **msgsnd** instruction. The payload bits are defined as follows.

Bit	Field Name	Description
37	BRDCAST	Broadcast The message is accepted by all processors regardless of the value of the GPIR register and the value of PIRTAG. 0 If the values of GPIR and PIRTAG are equal, a guest processor doorbell exception is generated. 1 A guest processor doorbell exception is generated regardless of the value of PIRTAG and GPIR.
38:41	Reserved	Reserved
42:49	LPIDTAG	LPID Tag The contents of this field are compared with the contents of the LPIDR. If LPIDTAG = 0, it matches all values in the LPIDR register.
50:63	PIRTAG	PIR Tag The contents of this field are compared with bits 50:63 of the GPIR register.

If a G_DBELL message is received by a processor, the message is accepted if one of the following conditions exist:

- The message is for this partition ($payload_{LPIDTAG} = LPIDR$)
- The message is for all partitions ($payload_{LPIDTAG} = 0$)

If a G_DBELL message is accepted, a guest processor doorbell exception is generated if one of the following conditions exists:

- This is a broadcast message ($payload_{BRDCAST} = 1$).
- The message is intended for this processor ($GPIR_{50:63} = payload_{PIRTAG}$).

G_DBELL messages are not cumulative. That is, if a G_DBELL message is accepted and the interrupt is pending because $MSR[CE] = 0$, further G_DBELL messages that would be accepted are ignored until the guest processor doorbell exception is cleared by taking the interrupt or cleared by executing an **msgclr** with a message type of G_DBELL on the receiving processor.

7.7.5 Guest Doorbell Critical Message Filtering

A processor receiving a G_DBELL_CRIT message type filters the message and either ignores the message or accepts the message and generates a guest processor doorbell critical exception based on the payload and the state of the processor at the time the message is received.

The payload is specified in the message and is determined by the contents of register RB_{37:63} used as the operand in the **msgsnd** instruction. The payload bits are defined as follows.

Bit	Field Name	Description
37	BRDCAST	<p>Broadcast</p> <p>The message is accepted by all processors regardless of the value of the GPIR register and the value of PIRTAG.</p> <p>0 If the values of GPIR and PIRTAG are equal, a guest processor doorbell critical exception is generated.</p> <p>1 A guest processor doorbell critical exception is generated regardless of the value of PIRTAG and GPIR.</p>
38:41	Reserved	Reserved
42:49	LPIDTAG	<p>LPID Tag</p> <p>The contents of this field are compared with the contents of the LPIDR. If LPIDTAG = 0, it matches all values in the LPIDR register.</p>
50:63	PIRTAG	<p>PIR Tag</p> <p>The contents of this field are compared with bits 50:63 of the GPIR register.</p>

If a G_DBELL_CRIT message is received by a processor, the message is accepted if one of the following conditions exist:

- The message is for this partition (payload_{LPIDTAG} = LPIDR).
- The message is for all partitions (payload_{LPIDTAG} = 0).

If a G_DBELL_CRIT message is accepted, a guest processor doorbell critical exception is generated if one of the following conditions exists:

- This is a broadcast message (payload_{BRDCAST}=1).
- The message is intended for this processor (GPIR_{50:63} = payload_{PIRTAG}).

G_DBELL_CRIT messages are not cumulative. That is, if a G_DBELL_CRIT message is accepted and the interrupt is pending because MSR_{CE} = 0, further G_DBELL messages that would be accepted are ignored until the guest processor doorbell critical exception is cleared by taking the interrupt or cleared by executing an **msgclr** with a message type of G_DBELL_CRIT on the receiving processor.

7.7.6 Guest Doorbell Machine Check Message Filtering

A processor receiving a G_DBELL_MC message type filters the message and either ignores the message or accepts the message and generates a guest processor doorbell machine check exception based on the payload and the state of the processor at the time the message is received.

The payload is specified in the message and is determined by the contents of register RB_{37:63} used as the operand in the **msgsnd** instruction. The payload bits are defined as follows.

Bit	Field Name	Description
37	BRDCAST	<p>Broadcast</p> <p>The message is accepted by all processors regardless of the value of the GPIR register and the value of PIRTAG.</p> <p>0 If the values of GPIR and PIRTAG are equal, a guest processor doorbell critical exception is generated.</p> <p>1 A guest processor doorbell critical exception is generated regardless of the value of PIRTAG and GPIR.</p>
38:41	Reserved	Reserved

Bit	Field Name	Description
42:49	LPIDTAG	LPID Tag The contents of this field are compared with the contents of the LPIDR. If LPIDTAG = 0, it matches all values in the LPIDR register.
50:63	PIRTAG	PIR Tag The contents of this field are compared with bits 50:63 of the GPIR register.

If a G_DBELL_MC message is received by a processor, the message is accepted if one of the following conditions exist:

- The message is for this partition ($\text{payload}_{\text{LPIDTAG}} = \text{LPIDR}$).
- The message is for all partitions ($\text{payload}_{\text{LPIDTAG}} = 0$).

If a G_DBELL_MC message is accepted, a guest processor doorbell machine check exception is generated if one of the following conditions exist:

- This is a broadcast message ($\text{payload}_{\text{BRDCAST}} = 1$).
- The message is intended for this processor ($\text{GPIR}_{50:63} = \text{payload}_{\text{PIRTAG}}$).

G_DBELL_MC messages are not cumulative. That is, if a G_DBELL_MC message is accepted and the interrupt is pending because $\text{MSR}_{\text{CE}} = 0$, further G_DBELL_MC messages that would be accepted are ignored until the guest processor doorbell machine check exception is cleared by taking the interrupt or cleared by executing an **msgclr** with a message type of G_DBELL_MC on the receiving processor.

The temporal relationship between when a G_DBELL_MC message is sent and when it is received in a given processor is not defined.

7.8 Interrupt Ordering and Masking

It is possible for multiple exceptions to exist simultaneously, each of which could cause the generation of an interrupt. Furthermore, the Power ISA does not provide for the generation of more than one interrupt of the same class (critical or noncritical) at a time. 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 type is masked (disabled) and an event causes an exception that would normally generate an interrupt of that type, the exception *persists* as a *status* bit in a register (which register depends upon the exception type). However, no interrupt is generated. Later, if the interrupt type is enabled (unmasked) and the exception status has not been cleared by software, the interrupt due to the original exception event is then finally generated.

All asynchronous interrupt types can be masked. Machine check interrupts can be masked, as well. In addition, certain synchronous interrupt types can be masked. The two synchronous interrupt types that can be masked are the floating-point enabled exception type of program interrupt (masked by $\text{MSR}[\text{FE0,FE1}]$ and the IAC, DAC, DVC, RET, and ICMP exception type debug interrupts (masked by $\text{MSR}[\text{DE}]$).

Architecture Note: When an otherwise synchronous, *precise* interrupt type is “delayed” in this fashion via masking and the interrupt type is later enabled, the interrupt that is then generated due to the exception event that occurred while the interrupt type was disabled is considered a synchronous, *imprecise* class of interrupt.

To prevent a subsequent interrupt from causing the state information (saved in SRR0/SRR1, CSRR0/CSRR1, or MCSRR0/MCSRR1) from a previous interrupt to be overwritten and lost, the A20 Core performs certain functions. As a first step, upon any noncritical class interrupt, the processor automatically

disables any further asynchronous, noncritical class interrupts (external input, decremter, user decremter, and fixed interval timer) by clearing MSR[EE]. Likewise, upon any critical class interrupt, hardware automatically disables any further asynchronous interrupts of either class (critical and noncritical) by clearing MSR[CE] and MSR[DE], in addition to MSR[EE]. The additional interrupt types that are disabled by the clearing of MSR[CE,DE] are the critical input, watchdog timer, and debug interrupts. For machine check interrupts, the processor automatically disables all maskable interrupts by clearing MSR[ME] as well as MSR[EE,CE,DE].

This first step of clearing MSR[EE] (and MSR[CE,DE] for critical class interrupts and MSR[ME] for machine checks) prevents any subsequent asynchronous interrupts from overwriting the relevant save/restore registers (SRR0/SRR1, CSRR0/CSRR1, or MCSRR0/MCSRR1), before software can save their contents. The processor also automatically clears, on any interrupt, MSR[PR,FP,FE0,FE1,IS,DS]. The clearing of these bits assists in the avoidance of subsequent interrupts of certain other types. However, *guaranteeing* that these interrupt types do not occur and thus do not 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.

7.8.1 Interrupt Ordering Software Requirements

The following list identifies the actions that system software must *avoid*, before saving the save/restore registers' contents:

- Reenabling of MSR[EE] (or MSR[CE,DE] in critical class interrupt handlers)

This prevents any asynchronous interrupts, and (in the case of MSR[DE]) any debug interrupts (which include both synchronous and asynchronous types).

- Branching (or sequential execution) to addresses not mapped by the TLB or mapped without execute access 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 the necessary access permission (read or write)

This prevents data storage and data TLB error interrupts.

- Execution of system call (**sc**) or trap (**tw**, **twi**, **td**, **tdi**) instructions

This prevents system call and trap exception types of program interrupts.

- Execution of any floating-point instructions

This prevents floating-point 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 the possibility of floating-point enabled exceptions.

- Reenabling of MSR[PR]

This prevents privileged Instruction exception type of program interrupts. Alternatively, software can re-enable MSR[PR], but avoid the execution of any privileged instructions.

- Execution of any auxiliary processor instructions that are not implemented in the A2O Core

This prevents auxiliary processor unavailable interrupts. Note that the auxiliary processor instructions that are implemented within the A2O Core do not cause any of these types of exceptions, and can therefore be executed before software has saved the save/restore registers' contents.

- Execution of any illegal instructions or any defined instructions not implemented within the A2O Core

This prevents illegal instruction exception type of program interrupts.

- Execution of any instruction that could cause an alignment interrupt

This prevents alignment interrupts. See *Alignment Interrupt* on page 320 for a complete list of instructions that can cause alignment interrupts.

- In the machine check handler, use of the caches and TLBs until any detected parity errors have been corrected.

This will avoid additional parity errors.

It is not necessary for hardware or software to avoid critical class interrupts from within noncritical class interrupt handlers (hence, the processor does not automatically clear MSR[CE,ME,DE] upon a noncritical interrupt), because the two classes of interrupts 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 both critical *and* noncritical class interrupts from within critical class interrupt handlers, even though the two classes of interrupts use different save/restore register pairs. This is because the critical class interrupt might have occurred from within a noncritical class interrupt handler, before the noncritical class interrupt handler saved SRR0 and SRR1. Therefore, within the critical class interrupt handler, both pairs of save/restore registers can contain data that is necessary to the system software.

Similarly, the machine check handler must avoid further machine checks, as well as both critical and noncritical interrupts, because the machine check handler might have been called from within a critical or noncritical interrupt handler.

7.8.2 Interrupt Order

The following is a prioritized list of the various enabled interrupt types for which exceptions might exist simultaneously:

1. Synchronous (nondebug) interrupts:
 - Data storage
 - Instruction storage
 - Alignment
 - Program
 - Embedded hypervisor privilege
 - Floating-point unavailable
 - System call
 - Embedded hypervisor system call
 - Auxiliary processor unavailable
 - Data TLB error
 - Instruction TLB error
 - LRAT error

Only one of the these types of synchronous interrupts can have an existing exception generating it at any given time. This is guaranteed by the exception priority mechanism (see *Exception Priorities* on page 349) and the requirements of the sequential execution model defined by the Power ISA.

2. Machine check
3. Guest processor doorbell machine check
4. Debug
5. critical input
6. Watchdog timer
7. Guest Watchdog timer
8. Processor doorbell critical
9. Guest processor doorbell critical
10. External input
11. Fixed-interval timer
12. Guest Fixed-interval timer
13. Decrementer
14. Guest Decrementer
15. Processor doorbell
16. Guest processor doorbell
17. User decrementer
18. Performance monitor

Even though, the noncritical, synchronous exception types listed under item 1 are generated with higher priority than the critical interrupt types listed in items 2 - 15, these noncritical interrupts will immediately be followed by the highest priority existing critical interrupt type, without executing any instructions at the noncritical interrupt handler. This is because the noncritical interrupt types do not automatically clear MSR[ME,DE,CE] and hence do not automatically disable the critical interrupt types. In all other cases, a particular interrupt type from the preceding list automatically disables any subsequent interrupts of the same type and all other interrupt types that are listed below it in the priority order.

7.9 Exception Priorities

The Power ISA requires all synchronous (precise and imprecise) interrupts to be reported in program order, as implied by the sequential execution model. The one exception to this rule is the case of multiple synchronous imprecise interrupts. Upon a synchronizing event, all previously executed instructions are required to report any synchronous imprecise interrupt-generating exceptions. The interrupts are then generated according to the general interrupt ordering rules outlined in *Interrupt Order* on page 348. For example, if an **mtmsr** instruction causes MSR[FE0,FE1,DE] to all be set, it is possible that a previous floating-point enabled exception and a previous debug exception both are still being presented (in the FPSCR and DBSR, respectively). In such a scenario, a floating-point enabled exception type of program interrupt occurs first, followed immediately by a debug interrupt.

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 are permitted to 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 listed in item 1 of *Interrupt Order* on page 348. The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions.

This section does not define the permitted setting of multiple exceptions for which the corresponding interrupt types are disabled. The generation of exceptions for which the corresponding interrupt types are disabled has no effect on the generation of other exceptions for which the corresponding 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, the occurrence of that enabled higher priority exception prevents the setting of the other exception, independent of whether that other exception's corresponding interrupt type is enabled or disabled.

Except as specifically noted below, only one of the exception types listed for a given instruction type is permitted to be generated at any given time, assuming that the corresponding interrupt type is enabled. The priority of the exception types is listed in the following sections ranging from highest to lowest, within each instruction type.

Finally, note that machine check exceptions are defined by the Power ISA architecture to be neither synchronous nor asynchronous. As such, machine check exceptions are not considered in the remainder of this section, which is specifically addressing the priority of synchronous interrupts.

7.9.1 Exception Priorities for Integer Load, Store, and Cache Management Instructions

The following list identifies the priority order of the exception types that can occur within the A2O Core as the result of the attempted execution of any integer load, store, or cache management instruction. Included in this category is the former opcode for the **icbt** instruction, which is an allocated opcode still supported by the A2O Core.

1. Debug (IAC exception)
2. Instruction TLB error (instruction TLB miss exception)
3. Instruction storage (execute access control exception)
4. LRAT error (instruction fetch)
5. Program (illegal instruction exception)
Only applies to the defined 64-bit load, store, and cache management instructions, which are not recognized by the A2O Core.
6. Program (privileged instruction)
7. Embedded hypervisor privilege
8. Data TLB error (data TLB miss exception)
9. Data storage (all exception types except byte ordering exception)
10. Alignment (alignment exception)
11. LRAT error (data access)
12. Debug (DAC or DVC exception)

13. Debug (ICMP exception)

7.9.2 Exception Priorities for Floating-Point Load and Store Instructions

The following list identifies the priority order of the exception types that can occur within the A2O Core as the result of the attempted execution of any floating-point load or store instruction.

1. Debug (IAC exception)
2. Instruction TLB error (instruction TLB miss exception)
3. Instruction storage (execute access control exception)
4. LRAT error (instruction fetch)
5. Program (illegal instruction exception)
This exception occurs if no floating-point unit is attached to the A2O Core, or if the particular floating-point load or store instruction is not recognized by the attached floating-point unit.
6. Program (privileged instruction)
7. Floating-point unavailable (floating-point unavailable exception)
This exception occurs if an attached floating-point unit recognizes the instruction, but floating-point instruction processing is disabled (MSR[FP] = 0).
8. Data TLB error (data TLB miss exception)
9. Data storage (all exception types except cache locking exception)
10. Alignment (alignment exception)
11. LRAT error (data access)
12. Debug (DAC or DVC exception)
13. Debug (ICMP exception)

7.9.3 Exception Priorities for Floating-Point Instructions (Other)

The following list identifies the priority order of the exception types that can occur within the A2O Core as the result of the attempted execution of any floating-point instruction other than a load or store.

1. Debug (IAC exception)
2. Instruction TLB error (instruction TLB miss exception)
3. Instruction storage (execute access control exception)
4. LRAT error (instruction fetch)
5. Program (illegal instruction exception)
This exception occurs if no floating-point unit is attached to the A2O Core or if the particular floating-point instruction is not recognized by the attached floating-point unit.
6. Floating-point unavailable (floating-point unavailable exception)
This exception occurs if an attached floating-point unit recognizes the instruction, but floating-point instruction processing is disabled (MSR[FP] = 0).
7. Program (floating-point enabled exception)
This exception occurs if an attached floating-point unit recognizes and supports the instruction, floating-point instruction processing is enabled (MSR[FP] = 1), and the instruction sets FPSCR[FEX] to 1.

8. Debug (ICMP exception)

7.9.4 Exception Priorities for Privileged Instructions

The following list identifies the priority order of the exception types that can occur within the A2O Core as the result of the attempted execution of any privileged instruction other than **dcbi**, **rfi**, **rfdi**, **rfmci**. This list *does* cover, however, the **dci** and **ici** instructions, which are privileged instructions that *are* implemented within the A2O Core.

1. Debug (IAC exception)
2. Instruction TLB error (instruction TLB miss exception)
3. Instruction storage (execute access control exception)
4. LRAT error (for hardware tablewalk for page table translation)
5. Program (illegal instruction exception, except for TLB management instructions with invalid MAS settings, see 8 below)
6. Program (privileged instruction exception)
7. Embedded hypervisor privilege
8. Program (illegal instruction exception, special case for TLB management instructions with invalid MAS settings)
9. LRAT error (for **tlbwe**)
10. Debug (ICMP exception)

7.9.5 Exception Priorities for Trap Instructions

The following list identifies the priority order of the exception types that can occur within the A2O Core as the result of the attempted execution of a trap (**tw**, **twi**, **td**, **tdi**) instruction.

1. Debug (IAC exception)
2. Instruction TLB error (instruction TLB miss exception)
3. Instruction storage (execute access control exception)
4. LRAT error (instruction fetch)
5. Debug (trap exception)
6. Program (trap exception)
7. Debug (ICMP exception)

7.9.6 Exception Priorities for System Call Instruction

The following list identifies the priority order of the exception types that can occur within the A2O Core as the result of the attempted execution of a system call (**sc**) instruction.

1. Debug (IAC exception)
2. Instruction TLB error (instruction TLB miss exception)
3. Instruction storage (execute access control exception)
4. LRAT error (instruction fetch)

5. System call (system call exception)
6. Embedded system call (system call exception)
7. Debug (ICMP exception)

Because the system call exception does not suppress the execution of the **sc** instruction, but rather the exception occurs once the instruction has completed, it is possible for an **sc** instruction to cause both a system call exception and an ICMP debug exception at the same time. In such a case, the associated interrupts occur in the order indicated in *Interrupt Order* on page 348.

7.9.7 Exception Priorities for Branch Instructions

The following list identifies the priority order of the exception types that can occur within the A2O Core as the result of the attempted execution of a branch instruction.

1. Debug (IAC exception)
2. Instruction TLB error (instruction TLB miss exception)
3. Instruction storage (execute access control exception)
4. LRAT error (instruction fetch)
5. Debug (BRT exception)
6. Debug (ICMP exception)

7.9.8 Exception Priorities for Return From Interrupt Instructions

The following list identifies the priority order of the exception types that can occur within the A2O Core as the result of the attempted execution of an **rfi**, **rfgi**, **rfci**, or **rfmci** instruction.

1. Debug (IAC exception)
2. Instruction TLB error (instruction TLB miss exception)
3. Instruction storage (execute access control exception)
4. LRAT error (instruction fetch)
5. Program (privileged instruction)
6. Debug (RET exception)
7. Debug (ICMP exception)

7.9.9 Exception Priorities for Reserved Instructions

The following list identifies the priority order of the exception types that can occur within the A2O Core as the result of the attempted execution of a reserved instruction.

1. Debug (IAC exception)
2. Instruction TLB error (instruction TLB miss exception)
3. Instruction storage (execute access control exception)
4. LRAT error (instruction fetch)
5. Program (illegal instruction exception)

Applies to all reserved instruction opcodes except the reserved nop instruction opcodes.

6. Debug (ICMP exception)
Only applies to the reserved-nop instruction opcodes.

7.9.10 Exception Priorities for All Other Instructions

The following list identifies the priority order of the exception types that can occur within the A2O Core as the result of the attempted execution of all other instructions (that is, those not covered by one of the sections 7.9.1 through 7.9.9). This includes both defined instructions and allocated instructions implemented within the A2O Core.

1. Debug (IAC exception)
2. Instruction TLB error (instruction TLB miss exception)
3. Instruction storage (execute access control exception)
4. LRAT error (instruction fetch)
5. Program (illegal instruction exception)
Applies only to the defined 64-bit instructions, as these are not implemented within the A2O Core.
6. Debug (ICMP exception)
Does not apply to the defined 64-bit instructions, as these are not implemented by the A2O Core.

8. FU Interrupts and Exceptions

An *interrupt* is the action in which the processor saves its old context (Machine State Register [MSR] and next instruction address [NIA]) and begins execution at a predetermined interrupt-handler address, with a modified MSR. *Exceptions* are the events that can cause the processor to take an interrupt, if the corresponding interrupt type is enabled.

Exceptions can be generated by the execution of instructions, or by signals from devices external to the A2O processor core, the internal timer facilities, debug events, or error conditions.

8.1 Floating-Point Exceptions

Book III-E requires all synchronous (precise and imprecise) interrupts to be reported in program order, as required by the sequential execution model. The only exception to this rule is the case of multiple synchronous imprecise interrupts. Upon a synchronizing event, all previously executed instructions are required to report any synchronous imprecise interrupt-generating exceptions, and the interrupt is then generated with all of those exception types reported cumulatively in both the Exception Syndrome Register (ESR) and any status registers associated with the particular exception type, such as the Floating-Point Status and Control Register (FPSCR).

For any single instruction attempting to cause multiple exceptions for which the corresponding synchronous interrupt types are enabled, this section defines the priority order in which the instruction is permitted to cause a *single* enabled exception, thus generating a particular synchronous interrupt. This exception priority mechanism, along with the requirement that synchronous interrupts must be generated in program order, guarantees that only one of the synchronous interrupt types is considered at any given time. The exception priority mechanism also prevents certain debug exceptions from existing in combination with certain other synchronous interrupt-generating exceptions.

This section does not define the permitted setting of multiple exceptions for which the corresponding interrupt types are disabled. The generation of exceptions for which the corresponding interrupt types are disabled has no effect on the generation of other exceptions for which the corresponding 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 other exception, regardless of whether the corresponding interrupt type of the other exception is enabled or disabled.

Except as noted, only one of the exception types listed for a given instruction type can be generated at any given time. The priority of the exception types is listed in subsequent sections ranging from highest to lowest, within each instruction type.

Note: Some exception types can be mutually exclusive of each other and can otherwise be considered the same priority. In such cases, the exceptions are listed in the order suggested by the sequential execution model.

Computational instructions can cause exceptions. Aside from instructions that write the FPSCR, none of the noncomputational instructions can cause a floating-point exception.

All exceptions are handled precisely. Because this can affect performance adversely, it is strongly recommended that exceptions should be disabled when possible. This prevents the product_name instruction stream from waiting for the execution of long latency instructions, such as **fdiv[s]** and **fsqrt[s]**.

8.2 Exceptions List

Book III-E defines the following floating-point exceptions:

- Invalid operation exception (VX)

Table 1. Invalid Operation Exception Categories

 Category	FPSCR Field
SNaN	VXSNAN
Infinity – Infinity	VXISI
Infinity ÷ Infinity	VXIDI
Zero ÷ Zero	VXZDZ
Infinity × Zero	VXIMZ
Invalid Compare	VXVC
Software Request	VXSOFT
Invalid Square Root	VXSQRT
Invalid Integer Convert	VXCVI

- Zero divide exception (ZX)
- Overflow exception (OX)
- Underflow exception (UX)
- Inexact exception (XI)

These exceptions can occur during execution of computational instructions. In addition, an invalid operation exception occurs when an **mtfsf** or **mtfsfi** instruction sets FPSCR[VXSOFT] = 1.

Each floating-point exception, and each category of invalid operation exception, has an exception bit in the FPSCR. Each floating-point exception also has a corresponding enable bit in the FPSCR. The exception bit indicates the occurrence of the corresponding exception. If an exception occurs, the corresponding enable bit controls the result produced by the instruction and, with MSR[FE0, FE1], whether and how the enabled exception type of program interrupt is taken. (See *Floating-Point Exceptions* on page 355 for more information.) In general, the enabling specified by an enable bit is to enable the invoking the interrupt, not to enable the exception to occur. The occurrence of an exception depends only on the instruction and its inputs, not on the setting of any enable bits. The only exceptions to this general rule are the occurrence of an underflow exception, which can depend on the setting of the enable bit, and the occurrence of an inexact exception, which can depend on the Overflow Exception bit not being set.

A single instruction, other than **mtfsf** or **mtfsfi**, can set more than one exception bit only in the following cases:

- An inexact exception can be set with an overflow exception.
- An inexact exception can be set with an underflow exception.
- An invalid operation exception (SNaN) is set with invalid operation exception ($\infty \times 0$) for multiply-add instructions for which the values being multiplied are infinity and 0 and the value being added is an SNaN.
- An invalid operation exception (SNaN) can be set with an invalid operation exception (invalid compare) for compare ordered instructions.

- Invalid operation exception (SNaN) can be set with an invalid operation exception (invalid integer convert) for convert-to-integer instructions.

When an exception occurs, instruction execution might be suppressed or a result might be delivered, depending on the exception.

Instruction execution 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 exceptions, a result is generated and written to the target specified by the instruction causing the exception. The result might be a different value for the enabled and disabled conditions for some of these exceptions. The following exceptions deliver a result:

- 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  are detected.

IEEE 754 specifies the handling of exceptional conditions in terms of “traps” and “trap handlers.” In Book III-E, an FPSCR exception enable bit of 1 causes generation of the result value specified in the IEEE standard for the “trap enabled” case. The exception is expected to be detected by software, which revises 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 is not implemented”) case. Software is not expected to detect the exception; it simply uses the default result. The result to be delivered in each case for each exception is described in subsequent sections.

The IEEE 754 default behavior when an exception occurs is to generate a default value and to not notify software. In Book III-E, if the IEEE 754 default behavior is desired for all exceptions, all FPSCR exception enable bits should be set to 0 and ignore exceptions mode should be used (see *Table 8-2* on page 358). In this case, an enabled exception type of program interrupt is not taken, even if floating-point exceptions occur. Software can inspect the FPSCR exception bits, if necessary, to determine whether exceptions have occurred.

If software is to be notified that a given kind of exception has occurred, the corresponding FPSCR exception enable bit must be set to 1 and a mode other than ignore exceptions mode must be used. In this case, the enabled exception type of program interrupt is taken if an enabled floating-point exception occurs. An enabled exception type of program interrupt is also taken if an **mtsf**s or **mtsf**si instruction sets an exception bit and its corresponding enable bit both to 1; the **mtsf**s or **mtsf**si instruction is considered to cause the enabled exception.

MSR[FE0, FE1] control whether and how enabled exception type of program interrupts are taken when an enabled floating-point exception occurs. An enabled exception type of program interrupt is never taken because of a disabled floating-point exception.

The imprecise modes (MSR[FE0, FE1] = 01 or 10) are not implemented in the A2 core.

Table 2. MSR[FE0, FE1] Modes

MSR[FE0]	MSR[FE1]	Mode
0	0	Ignore Exceptions Mode Floating-point exceptions do not cause an enabled exception type of program interrupt to be taken.
1	1	Precise Mode An enabled exception type of program interrupt is taken precisely at the instruction that caused the enabled exception.

If either MSR[FE0] or MSR[FE1] is 1, enabled exception type of program interrupts are treated as in precise mode.

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 MSR[FE0, FE1].

In all cases in which an enabled exception type of program interrupt is taken, all instructions before the instruction at which the enabled exception type of program interrupt is taken have completed, and no instruction after the instruction at which the enabled exception type of program interrupt is taken has begun execution. (Recall that, for the two imprecise modes, the instruction at which the enabled exception type of program interrupt is taken need not be the instruction that caused the exception.) The instruction at which the enabled exception type of program interrupt is taken has not been executed unless it is the excepting instruction, in which case it has been executed if the exception is not an enabled invalid operation exception or enabled zero divide exception.

Programming Note: In any of the three nonprecise modes, a Floating-Point Status and Control Register instruction can be used to force any exceptions, due to instructions initiated before the Floating-Point Status and Control Register instruction, to be recorded in the FPSCR. (This forcing is superfluous for precise mode.)

A **sync** instruction, or any other execution-synchronizing instruction or event, such as **isync**, also has the effects described above. However, to obtain the best performance across the widest range of implementations, a Floating-Point Status and Control Register instruction should be used to obtain these effects.

To obtain the best performance across the widest range of implementations, the programmer should follow these guidelines.

- If the IEEE 754 default results are acceptable to the application, ignore exceptions mode should be used, with all FPSCR exception enable bits set to 0.
- Ignore exceptions mode should not, in general, be used when any FPSCR exception enable bits are set to 1.
- Precise mode might degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

8.3 Floating-Point Interrupts

The following interrupts are taken under the control of the A2O processor core and are not enabled by or reported in FPSCR bits:

- Floating-point unavailable
- Floating-point assist

8.3.1 Floating-Point Unavailable Interrupt

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 the execution of the instruction causing the floating-point unavailable interrupt.

8.3.2 Floating-Point Assist Interrupt

Book III-E allows for an interrupt to be triggered when an enabled floating-point instruction requires software assistance to complete its execution. Typically, this would be used for handling such cases as denormalized operands or results. In the product_name, this is signalled, using the AXU exception bits, as an unimplemented instruction *after* the instruction has begun. However, in the product_name, there is no need for this interrupt. Either an instruction is fully implemented, or it is not implemented at all.

8.4 Floating-Point Exception Behavior

The following sections describe the behavior that results from the floating-point exceptions. For each exception, the definition of the exception is given, followed by a description of the action caused by the exception.

In general, each exception can result in either of two types of action, depending on whether the exception is enabled by its associated exception enable bit in the FPSCR.

8.4.1 Invalid Operation Exception

An invalid operation exception occurs when an operand is invalid for the specified operation. The invalid operations are:

- 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 conversion involving a number too large in magnitude to be represented in the target format, or involving an infinity or a NaN (invalid integer convert)

In addition, an invalid operation exception occurs if software explicitly requests this by executing an **mtfsf**, **mtfsfi**, or **mtfsb1** instruction that sets $FPSCR[VXSOFT] = 1$.

Programming Note: The purpose of $FPSCR[VXSOFT]$ is to enable software to cause an invalid operation exception for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it can be set by a program that computes a square root, if the source operand is negative.

8.4.1.1 Action

The action taken depends on the setting of $FPSCR[VE]$.

When an invalid operation exception is enabled ($FPSCR[VE] = 1$) and an invalid operation exception occurs or software explicitly requests the exception, the following actions are taken:

- One or two $FPSCR$ invalid operation exception bits, listed in *Table 8-3*, are set.

Table 3. Invalid Operation Exceptions

FPSCR Bit	Category
VXSNAN	SNaN
VXISI	Infinity – Infinity
VXIDI	Infinity ÷ Infinity
VXZDZ	Zero ÷ Zero
VXIMZ	Infinity × Zero
VXVC	Invalid Compare
VXSOFT	Software Request
VXSQRT	Invalid Square Root
VXCVI	Invalid Integer Convert 

- If the operation is an arithmetic, **frsp**, or convert to integer operation, the target FPR is unchanged.
 - $FPSCR[FR, FI] \leftarrow 0$
 - $FPSCR[FPRF] \leftarrow$ unchanged
- If the operation is a compare:
 - $FPSCR[FR, FI, C] \leftarrow$ unchanged
 - $FPSCR[FPCC] \leftarrow$ unordered
- If software explicitly requests the exception:
 - $FPSCR[FR, FI, FPRF]$ are as set by the **mtfsf**, **mtfsfi**, or **mtfsb1** instruction.

When invalid operation exception is disabled ($\text{FPSCR}[\text{VE}] = 0$) and an invalid operation exception occurs, or software explicitly requests the exception, the following actions are taken:

- One or two FPSCR invalid operation exception bits, listed in *Table 8-3*, are set.
- If the operation is an arithmetic or floating round to single-precision operation, the target FPR is set to a Quiet NaN.
 - $\text{FPSCR}[\text{FR}, \text{FI}] \leftarrow 0$
 - $\text{FPSCR}[\text{FPRF}] \leftarrow$ the class of the result (Quiet NaN)
- If the operation is a convert to 32-bit integer operation, the target FPR is set as follows:
 - $\text{FPR}(\text{FRT})_{0:31} \leftarrow$ undefined
 - $\text{FPR}(\text{FRT})_{32:63}$ are set to the most positive 32-bit integer if the operand in $\text{FPR}[\text{FRB}]$ is a positive number or $+\infty$, and to the most negative 32-bit integer if the operand in $\text{FPR}[\text{FRB}]$ is a negative number, $-\infty$, or NaN.
 - $\text{FPSCR}[\text{FR}, \text{FI}] \leftarrow 0$
 - $\text{FPSCR}[\text{FPRF}] \leftarrow$ undefined
- If the operation is a compare:
 - $\text{FPSCR}[\text{FR}, \text{FI}, \text{C}] \leftarrow$ unchanged
 - $\text{FPSCR}[\text{FPCC}] \leftarrow$ unordered
- If software explicitly requests the exception:
 - $\text{FPSCR}[\text{FR}, \text{FI}, \text{FPRF}]$ are as set by the **mtfsf**, **mtfsfi**, or **mtfsb1** instruction.

8.4.2 Zero Divide Exception

A zero divide exception occurs when an **fdiv[s]** instruction is executed with a zero divisor value and a finite nonzero dividend value. This exception also occurs when a reciprocal estimate instruction (**fres** or **frsqste**) is executed with an operand value of zero.

8.4.2.1 Action

The action to be taken depends on the setting of $\text{FPSCR}[\text{ZE}]$.

When a zero divide exception is enabled ($\text{FPSCR}[\text{ZE}] = 1$) and a zero divide occurs, the following actions are taken:

- The Zero Divide exception bit is set.
 - $\text{FPSCR}_{\text{ZX}} \leftarrow 1$
- $\text{FPR}(\text{FRT})_{0:31} \leftarrow$ unchanged
- $\text{FPSCR}[\text{FR}, \text{FI}] \leftarrow 0$
- $\text{FPSCR}[\text{FPRF}] \leftarrow$ unchanged

When a zero divide exception is disabled ($\text{FPSCR}[\text{ZE}] = 0$) and a zero divide occurs, the following actions are taken:

- The Zero Divide exception bit is set.

$$\text{FPSCR}_{\text{ZX}} \leftarrow 1$$



- $\text{FPR}(\text{FRT}) \leftarrow \pm\text{Infinity}$ (the sign is determined by the XOR of the signs of the operands)
- $\text{FPSCR}[\text{FR}, \text{FI}] \leftarrow 0$
- $\text{FPSCR}[\text{FPRF}] \leftarrow$ class and sign of the result ($\pm\text{Infinity}$)

8.4.3 Overflow Exception

Overflow 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.

8.4.3.1 Action

The action to be taken depends on the setting of $\text{FPSCR}[\text{OE}]$.

When overflow exceptions are enabled ($\text{FPSCR}[\text{OE}] = 1$) and exponent overflow occurs, the following actions are taken:

- The Overflow Exception bit is set.

$$\text{FPSCR}[\text{OX}] \leftarrow 1$$

- For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536.
- For single-precision arithmetic instructions and the **frsp** instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192.
- $\text{FPR}(\text{FRT}) \leftarrow$ adjusted rounded result
- $\text{FPSCR}[\text{FPRF}] \leftarrow$ class and sign of the result ($\pm\text{Normal Number}$)

When overflow exception is disabled ($\text{FPSCR}[\text{OE}] = 0$) and overflow occurs, the following actions are taken:

- The Overflow Exception bit is set.

$$\text{FPSCR}[\text{OX}] \leftarrow 1$$

- The Inexact Exception bit is set.

$$\text{FPSCR}[\text{XX}] \leftarrow 1$$

- The result is determined by the rounding mode ($\text{FPSCR}[\text{RN}]$) and the sign of the intermediate result as follows:
 - Round to Nearest
Store $\pm\text{Infinity}$, 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 largest finite number of the format.
- FPR(FRT) ← result
- FPSCR[FR] ← undefined
- FPSCR[FI] ← 1
- FPSCR[FPRF] ← class and sign of the result (\pm Infinity or \pm Normal Number)

8.4.4 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 underflow exception is disabled (FPSCR[UE] = 0), the intermediate result is denormalized (See *Normalization and Denormalization* on page 134) and rounded (See *Rounding Modes* on page 136) 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.

8.4.4.1 Action

The action to be taken depends on the setting of FPSCR[UE].

When underflow exception is enabled (FPSCR[UE] = 1) and exponent underflow occurs, the following actions are taken:

- The Underflow Exception bit is set.
FPSCR[UX] ← 1
- For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by adding 1536.
- For single-precision arithmetic instructions and the **frsp** instruction, the exponent of the normalized intermediate result is adjusted by adding 192.
- The adjusted rounded result is placed into the target FPR.

FPSCR[FPRF] ← class and sign of the result (\pm Normalized Number)

Programming Note: The FR and FI bits are provided to allow the enabled exception type of program interrupt, when taken because of an underflow exception, to simulate a “trap disabled” environment. That is, the FR and FI bits allow the enabled exception type of program interrupt to unround the result, thus allowing the result to be denormalized.

When underflow exception is disabled ($\text{FPSCR}[\text{UE}] = 0$) and underflow occurs, the following actions are taken:

- The Underflow Exception bit is set.
 $\text{FPSCR}[\text{UX}] \leftarrow 1$
- $\text{FPR}(\text{FRT}) \leftarrow$ rounded result
- $\text{FPSCR}[\text{FPRF}] \leftarrow$ class and sign of the result (\pm Normalized Number, \pm Denormalized Number, or \pm Zero)

8.4.5 Inexact Exception

An inexact exception occurs when either of the following 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.

8.4.5.1 Action

The action to be taken does not depend on the setting of $\text{FPSCR}[\text{XX}]$.

When an inexact exception occurs, the following actions are taken:

- The Inexact Exception bit is set.
 $\text{FPSCR}[\text{XX}] \leftarrow 1$
- $\text{FPR}(\text{FRT}) \leftarrow$ rounded or overflowed result
- $\text{FPSCR}[\text{FPRF}] \leftarrow$ class and sign of the result

Programming Note: In some implementations, enabling inexact exceptions might degrade performance more than does enabling other types of floating-point exception.

8.5 Exception Priorities for Floating-Point Load and Store Instructions

The following prioritized list of exceptions can occur as a result of the attempted execution of any floating-point load and store instruction.

1. Debug (ins  on address compare)
2. Instruction TLB error (all types)
3. Instruction storage interrupt (all types)
4. Program (illegal instruction)
5. Floating-point unavailable
6. Program (unimplemented operation)
7. Data TLB error (all types)
8. Data storage (all types)

- 9. Alignment
- 10. Debug (data address compare, data value compare)
- 11. Debug (instruction complete)

If an instruction causes both a debug (instruction address compare) exception, and a debug (data address compare) or debug (data value compare) exception, and does not cause any exception listed in items 2–9, both exceptions can be generated and recorded in the Debug Status Register (DBSR). A single debug interrupt results.

8.6 Exception Priorities for Other Floating-Point Instructions

The following prioritized list of exceptions can occur as a result of the attempted execution of any floating-point instruction other than a load or store.

- 1. Debug (instruction address compare)
- 2. Instruction TLB error (all types)
- 3. Instruction storage interrupt (all types)
- 4. Program (illegal instruction)
- 5. Floating-point unavailable
- 6. Program (unimplemented operation)
- 7. Program (enabled)
- 8. Debug (instruction complete)

8.7 QNaN

If any of the source operands is a NaN, either a signaling (SNaN) or quiet (QNaN), the result will be that NaN with the high-order fraction bit forced to 1 (that is, forced to a QNaN). The precedence, in decreasing order, is FRA, FRB, FRC. The resultant QNaN is only truncated on an **frsp**[.] instruction, in which case the most significant 35 bits are copied to the target, with the least significant 29 forced to zero.

Table 4. QNaN Result

R_a	R_b	R_c	Resultant QNaN ¹
NaN	X	X	R_a
—	NaN	X	R_b^2
—	—	NaN	R_c

1. High-order fraction bit is forced to a 1.
 2. **frsp**: Result is $(FRB)_{0:11} || 1 || (FRB)_{13:34} || 290$.

8.8 Updating FPRs on Exceptions

The target FPR is never updated on enabled invalid exceptions and enabled divide by zero exceptions. This requirement exists because an instruction can potentially use one of the source registers as a target register, yet it is necessary that the trap handler be able to examine and act upon the source operands.

In all other cases, a floating-point exception does not block the writing of the target FPR.

8.9 Floating-Point Status and Control Register (FPSCR)

The computational instructions modify the FPSCR. With the exception of instructions that write directly to the FPSCR, none of the noncomputational instructions modify the FPSCR.

The FPSCR controls the handling of floating-point exceptions and records status resulting from the floating-point operations. FPSCR_{32:55} are status bits. FPSCR_{29:31} and FPSCR_{56:63} are control bits.

The exception bits in the FPSCR (bits 35:44, 53:55) are sticky; that is, once set to 1 they remain set to 1 until they are set to 0 by an **mcrfs**, **mtfsfi**, **mtfsf**, or **mtfsb0** instruction. The exception summary bits FPSCR[F_X, FEX, VX] are not considered as exception bits, and only FPSCR[F_X] is sticky.

FPSCR[FEX, VX] are simply ORs of other FPSCR bits. Therefore, these bits are not listed among the FPSCR bits affected by the various instructions.

FPSCR[FPRF], which contains five result flag bits, is set for arithmetic, rounding, and conversion instructions based on the class of the result value placed into the target FPR. If any portion of a result is undefined, the value placed into FPSCR[FPRF] is undefined. *Table 8-5* describes how the values of the result flags in FPSCR[FPRF] correspond to the result value classes.

Table 5. FPSCR[FPRF] Result Flags

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

Table 8-6 illustrates the FPSCR.

Table 6. Floating-Point Status and Control Register (FPSCR) (Sheet 1 of 3)

Bits	Field Name	Description
0:28		<u>Reserved</u> Note: FPSCR[28] is reserved for extension of the DRN field; therefore DRN can be set by using the mtfsfi instruction to set the rounding mode.
29:31	DRN	<u>DFP Rounding Control</u> 000 Round to nearest, ties to even. 001 Round toward zero. 010 Round toward +infinity. 011 Round toward -infinity. 100 Round to nearest, ties away from 0. 101 Round to nearest, ties toward 0. 110 Round to away from zero. 111 Round to prepare for shorter precision. See Section 5.5.2 in PowerISA Version 2.06B.
32	FX	<u>Floating-Point Exception Summary</u> 0 No FPSCR exception bits changed from 0 to 1. 1 At least one FPSCR exception bit changed from 0 to 1. All floating-point instructions, except mtfsfi and mtfsf , implicitly set this field to 1 if the instruction causes any floating-point exception bits in the FPSCR to change from 0 to 1. mcrfs , mtfsfi , mtfsf , mtfsb0 , and mtfsb1 can alter this field explicitly.
33	FEX	<u>Floating-Point Enabled Exception Summary</u> The OR of all the floating-point exception fields masked by their respective enable fields. mcrfs , mtfsfi , mtfsf , mtfsb0 , and mtfsb1 cannot alter this field explicitly.
34	VX	<u>Floating-Point Invalid Operation Exception Summary</u> The OR of all the invalid operation exception fields. mcrfs , mtfsfi , mtfsf , mtfsb0 , and mtfsb1 cannot alter this field explicitly.
35	OX	<u>Floating-Point Overflow Exception</u> 0 A floating-point overflow exception did not occur. 1 A floating-point overflow exception occurred. See <i>Overflow Exception</i> on page 362.
36	UX	<u>Floating-Point Underflow Exception</u> 0 A floating-point underflow exception did not occur. 1 A floating-point underflow exception occurred. See <i>Underflow Exception</i> on page 363.
37	ZX	<u>Floating-Point Zero Divide Exception</u> 0 A floating-point zero divide exception did not occur. 1 A floating-point zero divide exception occurred. See <i>Zero Divide Exception</i> on page 361.
38	XX	<u>Floating-Point Inexact Exception</u> 0 A floating-point inexact exception did not occur. 1 A floating-point inexact exception occurred. This field is a sticky version of FPSCR[F _I]. The following rules describe how a given instruction sets this field. If the instruction affects FPSCR[F _I], the new value of this field is obtained by ORing the old value of this field with the new value of FPSCR[F _I]. If the instruction does not affect FPSCR[F _I], the value of this field is unchanged.
39	VXSNAN	<u>Floating-Point Invalid Operation Exception (SNaN)</u> 0 A floating-point invalid operation exception (VXSNAN) did not occur. 1 A floating-point invalid operation exception (VXSNAN) occurred. See <i>Invalid Operation Exception</i> on page 359.

Table 6. Floating-Point Status and Control Register (FPSCR) (Sheet 2 of 3)

Bits	Field Name	Description
40	VXISI	<u>Floating-Point Invalid Operation Exception ($\infty - \infty$)</u> 0 A floating-point invalid operation exception (VXISI) did not occur. 1 A floating-point invalid operation exception (VXISI) occurred. See <i>Invalid Operation Exception</i> on page 359.
41	VXIDI	<u>Floating-Point Invalid Operation Exception ($\infty \div \infty$)</u> 0 A floating-point invalid operation exception (VXIDI) did not occur. 1 A floating-point invalid operation exception (VXIDI) occurred. See <i>Invalid Operation Exception</i> on page 359.
42	VXZDZ	<u>Floating-Point Invalid Operation Exception ($0 \div 0$)</u> 0 A floating-point invalid operation exception (VXZDZ) did not occur. 1 A floating-point invalid operation exception (VXZDZ) occurred. See <i>Invalid Operation Exception</i> on page 359.
43	VXIMZ	<u>Floating-Point Invalid Operation Exception ($\infty \times 0$)</u> 0 A floating-point invalid operation exception (VXIMZ) did not occur. 1 A floating-point invalid operation exception (VXIMZ) occurred. See <i>Invalid Operation Exception</i> on page 359.
44	VXVC	<u>Floating-Point Invalid Operation Exception (Invalid Compare)</u> 0 A floating-point invalid operation exception (VXVC) did not occur. 1 A floating-point invalid operation exception (VXVC) occurred. See <i>Invalid Operation Exception</i> on page 359.
45	FR	<u>Floating-Point Fraction Rounded</u> The last arithmetic or rounding and conversion instruction either produced an inexact result during rounding or caused a disabled overflow exception. See <i>Rounding</i> on page 135. This bit is not sticky.
46	FI	<u>Floating-Point Fraction Inexact</u> The last arithmetic or rounding and conversion instruction either produced an inexact result during rounding or caused a disabled overflow exception. See <i>Rounding</i> on page 135. This bit is not sticky. See the definition of FPSCR[XX] regarding the relationship between FPSCR[FI] and FPSCR[XX].
47	FPRF	<u>Floating-Point Result Flag (FPRF)</u>
48	FL	<u>Floating-Point Less Than or Negative</u>
49	FG	<u>Floating-Point Greater Than or Positive</u>
50	FE	<u>Floating-Point Equal to Zero</u>
51	FU	<u>Floating-Point Unordered or NaN</u>
52		Reserved
53	VXSOFT	<u>Floating-Point Invalid Operation Exception (Software Request)</u> 0 A floating-point invalid operation exception (software request) did not occur. 1 A floating-point invalid operation exception (software request) occurred. See <i>Invalid Operation Exception</i> on page 359.
54	VXSQRT	<u>Floating-Point Invalid Operation Exception (Invalid Square Root)</u> 0 A floating-point invalid operation exception (invalid square root) did not occur. 1 A floating-point invalid operation exception (invalid square root) occurred. See <i>Invalid Operation Exception</i> on page 359.
55	VXCVI	<u>Floating-Point Invalid Operation Exception (Invalid Integer Convert)</u> 0 A floating-point invalid operation exception (invalid integer convert) did not occur. 1 A floating-point invalid operation exception (invalid integer convert) occurred. See <i>Invalid Operation Exception</i> on page 359.

Table 6. Floating-Point Status and Control Register (FPSCR) (Sheet 3 of 3)

Bits	Field Name	Description
56	VE	<u>Floating-Point Invalid Operation Exception Enabled</u> 0 Floating-point invalid operation exceptions are disabled. 1 Floating-point invalid operation exceptions are enabled.
57	OE	<u>Floating-Point Overflow Exception Enable</u> 0 Floating-point overflow exceptions are disabled. 1 Floating-point overflow exceptions are enabled.
58	UE	<u>Floating-Point Underflow Exception Enable</u> 0 Floating-point underflow exceptions are disabled. 1 Floating-point underflow exceptions are enabled.
59	ZE	<u>Floating-Point Zero Divide Exception Enable</u> 0 Floating-point zero divide exceptions are disabled. 1 Floating-point zero divide exceptions are enabled.
60	XE	<u>Floating-Point Inexact Exception Enable</u> 0 Floating-point inexact exceptions are disabled. 1 Floating-point inexact exceptions are enabled.
61	NI	<u>Floating-Point Non-IEEE Mode</u> 0 Non-IEEE mode is disabled. 1 Non-IEEE mode is enabled. If FPSCR[NI] = 1, the remaining FPSCR bits might have meanings other than those given in this document, and the results of floating-point operations need not conform to the IEEE standard. If the IEEE-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. The behavior when FPSCR[NI] = 1 can vary from one implementation to another
62:63	RN	<u>Floating-Point Rounding Control</u> 00 Round to nearest. 01 Round toward zero. 10 Round toward +infinity. 11 Round toward -infinity. See <i>Rounding</i> on page 135.

8.10 Updating the Condition Register

Architecturally, excepting floating-point instructions do not block the updating of the Condition Register in the A2O processor core.

8.10.1 Condition Register (CR)

The CR fields are modified by various floating-point instructions.

Register Short Name:	CR	Read Access:	Any
Decimal SPR Number:	N/A	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:35	CR0	0b0000	Condition Register Field 0
36:39	CR1	0b0000	Condition Register Field 1
40:43	CR2	0b0000	Condition Register Field 2
44:47	CR3	0b0000	Condition Register Field 3
48:51	CR4	0b0000	Condition Register Field 4
52:55	CR5	0b0000	Condition Register Field 5
56:59	CR6	0b0000	Condition Register Field 6
60:63	CR7	0b0000	Condition Register Field 7

8.10.2 Updating CR Fields

The floating-point compare instructions **fcmpo** and **fcmpu** specify a CR field that is updated with the compare results.

Table 8-7 illustrates the bit encodings for a CR field containing the results of an **fcmpo** and **fcmpu** instruction.

Table 7. Bit Encodings for a CR Field

CR Field (Bit)	Description
0	<i>Floating-Point Less Than (FL)</i> Floating-point compare: (FRA) < (FRB)
1	<i>Floating-Point Greater Than (FG)</i> Floating-point compare: (FRA) > (FRB)
2	<i>Floating-Point Equal (FE)</i> Floating-point compare: (FRA) = (FRB)
3	<i>Floating-Point Unordered (FU)</i> Floating-point compare: One or both of (FRA) or (FRB) is a NaN.

The **mcrfs** instruction moves a specified FPSCR field into a CR field.

8.10.3 Generation of QNaN Results

If a disabled invalid operation exception is caused by operating on a NaN, the value returned follows the rules indicated in *Table 8-4* on page 365.

If the exception was not caused by operating on a NaN, a QNaN must be generated. The generated QNaN has a sign bit of 0, an exponent of all 1s, a high-order fraction bit of 1 with all other fraction bits of 0: 0x7FF8000000000000.

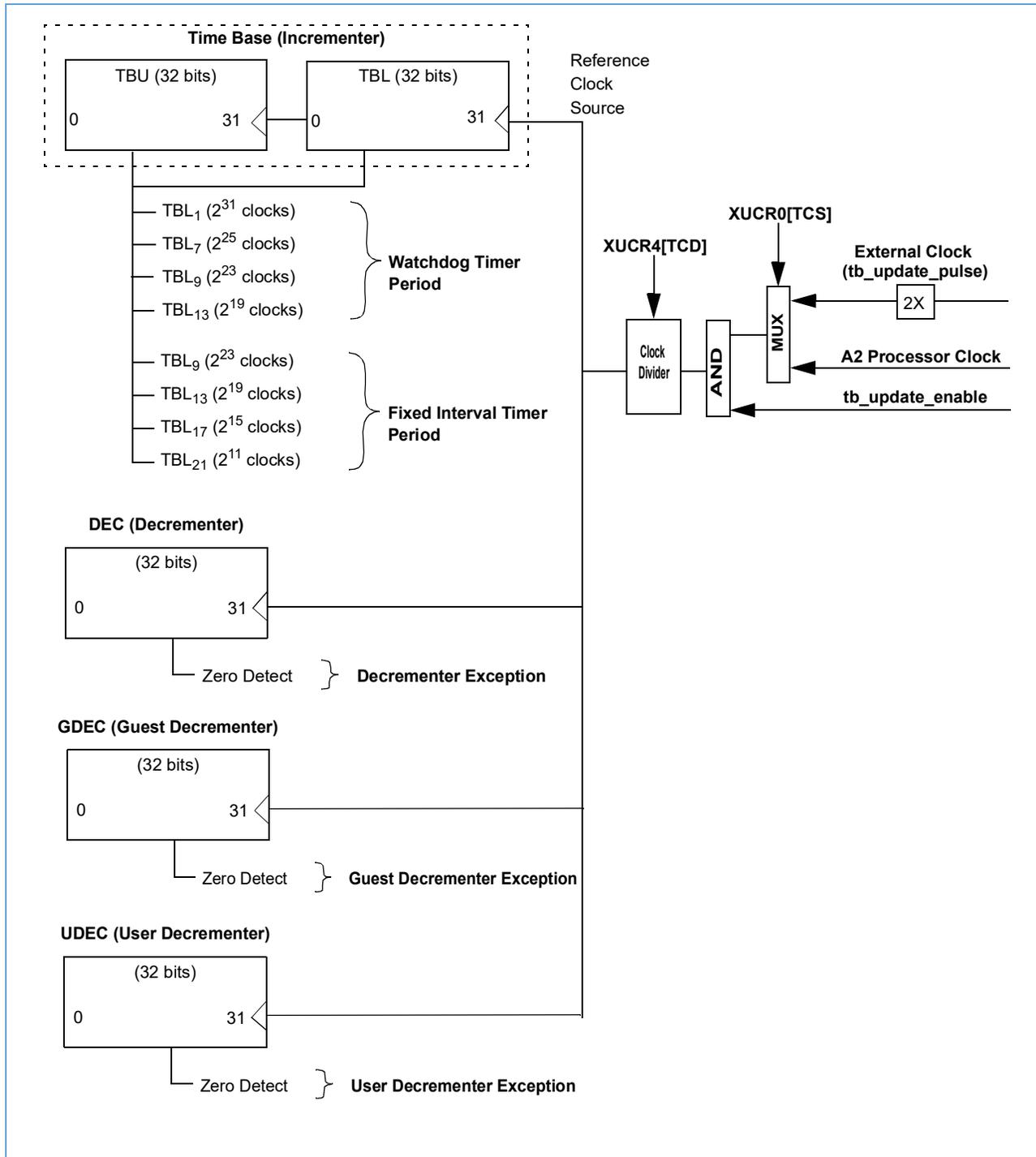
9. Timer Facilities

The A2O core provides five timer facilities: a time base, a decremter (DEC), a guest decremter (GDEC), a user decremter (UDEC), a fixed interval timer (FIT), a guest fixed interval timer (GFIT), a watchdog timer, and a guest watchdog timer. These facilities, which share the same source clock frequency, can support:

- Time-of-day functions
- General software timing functions
- Peripherals requiring periodic service
- General system maintenance
- System error recover ability

Figure 9-1 shows the relationship between these facilities and the clock source.

Figure 3. Relationship of Timer Facilities to the Time Base



9.1 Time Base

The time base is a 64-bit register that increments once during each period of the source clock and provides a time reference. Access to the time base is via two Special Purpose Registers (SPRs). The Time Base Upper (TBU) SPR contains the high-order 32 bits of the time base, while the Time Base Lower (TBL) SPR contains the low-order 32 bits.

Software access to the Timebase Register (TB) and TBU is nonprivileged for read. Software access to TBU and TBL is privileged for write, and hence different SPR numbers are used for reading than for writing. TBU and TBL are written using **mtspr** and read using **mfspr**.



The period of the 64-bit time base is approximately 254 years for a 2.3 GHz clock source. The time base value itself does not generate any exceptions, even when it wraps. For most applications, the time base is set once at system reset and only read thereafter. Note that fixed interval timer and watchdog timer exceptions (described on page 378 and page 379) are caused by 0→1 transitions of selected bits from the time base. Transitions of these bits caused by software alteration of the time base have the same effect as transitions caused by normal incrementing of the time base.

Table 1. Timebase Register (TB)

Register Short Name:	TB	Read Access:	Any
Decimal SPR Number:	268	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	TBU	0x0	<u>Timebase Upper</u> Provides access to the upper portion of the Timebase
32:63	TBL	0x0	<u>Timebase Lower</u> Provides access to the lower portion of the Timebase

Table 2. Timebase Lower Register (TBL)

Register Short Name:	TBL	Read Access:	None
Decimal SPR Number:	284	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	TBL	0x0	<u>Timebase Lower</u> Provides access to the lower portion of the Timebase

Table 3. Timebase Upper Register (TBU)

Register Short Name:	TBU	Read Access:	None/Any
Decimal SPR Number:	285/269	Write Access:	Hypv/None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	TBU	0x0	<u>Timebase Upper</u> Provides access to the upper portion of the Timebase

9.1.1 Reading the Time Base

In 64-bit mode, the time base can be read with one instruction.

```
mfspir      Ry,TB      # Read TB into GPR Ry.
```

In 32-bit mode, the following code provides an example of reading the time base.

```
loop:
mfspir      Rx,TBU     # Read TBU into GPR Rx.
mfspir      Ry,TB      # Read TBL into GPR Ry.
mfspir      Rz,TBU     # Read TBU again, this time into GPR Rz.
cmpw       Rz, Rx      # See if old = new.
bne        loop       # Loop/reread if rollover occurred.
```

The comparison and loop ensure that a consistent pair of values is obtained.

9.1.2 Writing the Time Base

The following code provides an example of writing the time base.

```
lwz        Rx, upper   # Load 64-bit time base value into GPRs Rx and Ry.
lwz        Ry, lower
li         Rz, 0        # Set GPR Rz to 0.
mtspr     TBL,Rz       # Force TBL to 0 (thereby preventing wrap into TBU).
mtspr     TBU,Rx       # Set TBU to initial value.
mtspr     TBL,Ry       # Set TBL to initial value.
```

9.2 Decrementer (DEC)

The DEC is a 32-bit privileged SPR that decrements at the same rate that the time base increments. The DEC is read and written using **mfspir** and **mtspr**, respectively. When a nonzero value is written to the DEC, it begins to decrement with the next time base clock. A decrementer exception is signalled when a decrement occurs on a DEC count of 1, and the Decrementer Interrupt Status field of the Timer Status Register (TSR[DIS]; see page 385) is set. A decrementer interrupt occurs if it is enabled by both the Decrementer Interrupt Enable field of the Timer Control Register (TCR[DIE]; see page 383) and by either the External Inter-

rupt Enable or Guest State fields of the Machine State Register (MSR[EE] or MSR[GS]; see *Section 7.5.2 Machine State Register (MSR)* on page 285). *Section 7 CPU Interrupts and Exceptions* on page 277 provides more information about the handling of decremter interrupts.

The decremter interrupt handler software should clear TSR[DIS] before re-enabling MSR[EE] or MSR[GS] to avoid another decremter interrupt due to the same exception (unless TCR[DIE] is cleared instead).

The behavior of the DEC itself upon a decrement from a DEC value of 1 depends on which of two modes it is operating in: normal, or auto-reload. The mode is controlled by the Auto-Reload Enable (ARE) field of the TCR. When operating in normal mode (TCR[ARE] = 0), the DEC simply decrements to the value 0 and then stops decrementing until it is re-initialized by software.

When operating in auto-reload mode (TCR[ARE] = 1), however, instead of decrementing to the value 0, the DEC is reloaded with the value in the Decrementer Auto-Reload (DECAR) Register (see *Table 9-5*), and continues to decrement with the next time base clock (assuming the DECAR value was nonzero). The DECAR register is a 32-bit privileged SPR, and is read/written using **mfspr/mtspr**.

The auto-reload feature of the DEC is disabled upon reset, and must be enabled by software.

Table 4. Decrementer Register (DEC)

Register Short Name:	DEC	Read Access:	Hypv
Decimal SPR Number:	22	Write Access:	Hypv
Initial Value:	0x000000007ffffff	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	DEC	0x7ffffff	<u>Decrementer</u> The Decrementer (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay. The contents of the Decrementer are treated as a signed integer.

Table 5. Decrementer Auto-Reload Register (DECAR)

Register Short Name:	DECAR	Read Access:	Hypv
Decimal SPR Number:	54	Write Access:	Hypv
Initial Value:	0x000000007ffffff	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	DECAR	0x7ffffff	<u>Decrementer Auto-Reload</u> If TCR[ARE]=1, TSR[DIS] is set to 1, the contents of the Decrementer Auto-Reload Register is then placed into the DEC, and the Decrementer continues decrementing from the reloaded value

Using **mtspr** to force the DEC to 0 does *not* cause a decremter exception, and thus does not cause TSR[DIS] to be set. However, if a time base clock causes a decrement from a DEC value of 1 to occur simultaneously with the writing of the DEC by an **mtspr** instruction, then the decremter exception *does* occur, TSR[DIS] is set, and the DEC is written with the value from the **mtspr**.

For software to quiesce the activity of the DEC and eliminate all DEC exceptions, follow this procedure:

1. Write 0 to TCR[DIE]. This prevents a decremter exception from causing a decremter interrupt.
2. Write 0 to TCR[ARE]. This disables the DEC auto-reload feature.
3. Write 0 to the DEC to halt decrementing. Although this action does not itself cause a decremter exception, it is possible that a decrement from a DEC value of 1 has occurred since the last time that TSR[DIS] was cleared.
4. Write 1 to TSR[DIS] (DEC Interrupt Status bit). This clears the decremter exception by setting TSR[DIS] to 0. Because the DEC is no longer decrementing (due to having been written with 0 in step 3), no further decremter exceptions are possible.

9.3 Guest Decrementer (GDEC)

The GDEC is a 32-bit SPR that decrements at the same rate that the time base increments. The GDEC is read and written using **mfspir** and **mtspir**, respectively. When a nonzero value is written to the GDEC, it begins to decrement with the next time base clock. A guest decremter exception is signalled when:

- A decrement occurs on a GDEC count of 1, and
- **The Guest Decrementer Interrupt Status field of the Timer Status Register (GTSR[UDIS] (see page 385) is set, and**
- The exception is enabled by GTCR[DIE]=1 and MSREE=1 and MSRGS=1

Section 7 CPU Interrupts and Exceptions on page 277 provides more information about the handling of Guest Decrementer interrupts.

The guest decremter interrupt handler software should clear GTSR[UDIS] before re-enabling MSR[EE] or MSR[GS] to avoid another user decremter interrupt due to the same exception (unless GTCR[UDIE] is cleared instead).

When the GDEC decrements from a value of 1, the GDEC simply decrements to the value 0, and then stops decrementing until it is re-initialized by software.

Register Short Name:	UDEC	Read Access:	Any
Decimal SPR Number:	550	Write Access:	Any
Initial Value:	0x000000007ffffff	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	UDEC	0x7ffffff	<p><u>User Decrementer</u></p> <p>The User Decrementer (UDEC) is a 32-bit decrementing counter that provides a mechanism for causing a User Decrementer interrupt after a programmable delay. The contents of the User Decrementer are treated as a signed integer.</p> <p>Note: If TCR[UD]=0, this accesses to this register are treated as an illegal SPR.</p>

Using **mtspr** to force the GDEC to 0 does *not* cause a guest decremter exception, and thus does not cause GTSR[UDIS] to be set. However, if a time base clock causes a decrement from a GDEC value of 1 to occur simultaneously with the writing of the GDEC by an **mtspr** instruction, then the decremter exception *does* occur, GTSR[UDIS] is set, and the GDEC is written with the value from the **mtspr**.

For software to quiesce the activity of the GDEC and eliminate all GDEC exceptions, follow this procedure:

1. Write 0 to GTCR[UDIE]. This prevents a guest decremter exception from causing a guest decremter interrupt.
2. Write 0 to the GDEC to halt decrementing. Although this action does not itself cause a guest decremter exception, it is possible that a decrement from a GDEC value of 1 has occurred since the last time that GTSR[UDIS] was cleared.

Write 1 to GTSR[UDIS] (GDEC Interrupt Status bit). This clears the guest decremter exception by setting GTSR[UDIS] to 0. Because the GDEC is no longer decrementing (due to having been written with 0 in step 2), no further Guest Decrementer exceptions are possible.

9.4 User Decrementer (UDEC)

The UDEC is a 32-bit SPR that decrements at the same rate that the time base increments. The UDEC is read and written using **mfspir** and **mtspir**, respectively. When a nonzero value is written to the UDEC, it begins to decrement with the next time base clock. A user decremter exception is signalled when:

- A decrement occurs on a UDEC count of 1, and
- The User Decrementer Interrupt Status field of the Timer Status Register (TSR[UDIS]) (see page 385) is set, and
- The exception is enabled by either the External Interrupt Enable or Guest State fields of the Machine State Register (MSR[EE] or MSR[GS]) (see *Section 7.5.2 Machine State Register (MSR)* on page 285).

Section 7 CPU Interrupts and Exceptions on page 277 provides more information about the handling of User Decrementer interrupts.

The user decremter interrupt handler software should clear TSR[UDIS] before re-enabling MSR[EE] or MSR[GS] to avoid another user decremter interrupt due to the same exception (unless TCR[UDIE] is cleared instead).

When the UDEC decrements from a value of 1, the UDEC simply decrements to the value 0, and then stops decrementing until it is re-initialized by software.

Register Short Name:	UDEC	Read Access:	Any
Decimal SPR Number:	550	Write Access:	Any
Initial Value:	0x000000007ffffff	Duplicated for MT:	Y

Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	UDEC	0x7fffffff	<p>User Decrementer</p> <p>The User Decrementer (UDEC) is a 32-bit decrementing counter that provides a mechanism for causing a User Decrementer interrupt after a programmable delay. The contents of the User Decrementer are treated as a signed integer.</p> <p>Note: If TCR[UD]=0, this accesses to this register are treated as an illegal SPR.</p>

Using **mtspr** to force the UDEC to 0 does *not* cause a user decrementer exception, and thus does not cause TSR[UDIS] to be set. However, if a time base clock causes a decrement from a UDEC value of 1 to occur simultaneously with the writing of the UDEC by an **mtspr** instruction, then the decrementer exception *does* occur, TSR[UDIS] is set, and the UDEC is written with the value from the **mtspr**.

For software to quiesce the activity of the UDEC and eliminate all UDEC exceptions, follow this procedure:

1. Write 0 to TCR[UDIE]. This prevents a user decrementer exception from causing a user decrementer interrupt.
2. Write 0 to the UDEC to halt decrementing. Although this action does not itself cause a user decrementer exception, it is possible that a decrement from a UDEC value of 1 has occurred since the last time that TSR[UDIS] was cleared.
3. Write 1 to TSR[UDIS] (UDEC Interrupt Status bit). This clears the user decrementer exception by setting TSR[UDIS] to 0. Because the UDEC is no longer decrementing (due to having been written with 0 in step 2), no further User Decrementer exceptions are possible.

9.5 Fixed Interval Timer (FIT)

The FIT provides a mechanism for causing periodic exceptions with a regular period. The FIT is typically used by system software to invoke a periodic system maintenance function, executed by the fixed interval timer interrupt handler.

A fixed interval timer exception occurs on a 0→1 transition of a selected bit from the time base. Note that a fixed interval timer exception also occurs if the selected time base bit transitions from 0→1 due to an **mtspr** instruction that writes 1 to that time base bit when its previous value was 0.

The Fixed Interval Timer Period (FP) field of the TCR selects one of 4 bits from the time base, as shown in *Table 9-6*.

Table 6. Fixed Interval Timer Period Selection

TCR[FP]	Time Base Bit	Period (Time Base Clocks)	Period (32 MHz Clock)	Period (1.6 GHz Clock)	Period (2.3 GHz Clock)
0b00	TBL ₂₁	2 ¹¹ clocks	64.00 μs	1.28 μs	.89 μs
0b01	TBL ₁₇	2 ¹⁵ clocks	1.02 ms	20.48 μs	14.25 μs
0b10	TBL ₁₃	2 ¹⁹ clocks	16.38 ms	327.68 μs	227.95 μs
0b11	TBL ₉	2 ²³ clocks	262.14 ms	5.24 ms	3.65 ms

When a fixed interval timer exception occurs, the exception status is recorded by setting the Fixed interval Timer Interrupt Status (FIS) field of the TSR to 1. A fixed interval timer interrupt occurs if it is enabled by both the Fixed Interval Timer Interrupt Enable (FIE) field of the TCR and by either MSR[EE] or MSR[GS]. *Section 7.6.13 Fixed-Interval Timer Interrupt* on page 327 provides more information about the handling of Fixed Interval Timer interrupts.

The fixed interval timer interrupt handler software should clear TSR[FIS] before re-enabling MSR[EE] or MSR[GS], to avoid another fixed interval timer interrupt due to the same exception (unless TCR[FIE] is cleared instead).

9.6 Guest Fixed Interval Timer (GFIT)

The GFIT provides a mechanism for causing periodic exceptions with a regular period. The GFIT is typically used by system software to invoke a periodic system maintenance function, executed by the fixed interval timer interrupt handler.

A guest fixed interval timer exception occurs on a 0→1 transition of a selected bit from the time base. Note that a guest fixed interval timer exception also occurs if the selected time base bit transitions from 0→1 due to an **mtspr** instruction that writes 1 to that time base bit when its previous value was 0.

The Guest Fixed Interval Timer Period (FP) field of the GTCR selects one of 4 bits from the time base, as shown in *Table 9-6*.

Table 7. Fixed Interval Timer Period Selection

TCR[FP]	Time Base Bit	Period (Time Base Clocks)	Period (32 MHz Clock)	Period (1.6 GHz Clock)	Period (2.3 GHz Clock)
0b00	TBL ₂₁	2 ¹¹ clocks	64.00 μs	1.28 μs	.89 μs
0b01	TBL ₁₇	2 ¹⁵ clocks	1.02 ms	20.48 μs	14.25 μs
0b10	TBL ₁₃	2 ¹⁹ clocks	16.38 ms	327.68 μs	227.95 μs
0b11	TBL ₉	2 ²³ clocks	262.14 ms	5.24 ms	3.65 ms

When a guest fixed interval timer exception occurs, the exception status is recorded by setting the Fixed interval Timer Interrupt Status (FIS) field of the GTSR to 1. A fixed interval timer interrupt occurs if it is enabled by both the Fixed Interval Timer Interrupt Enable (FIE) field of the GTCR and by either MSR[EE] or MSR[GS]. *Section 7.6.13 Fixed-Interval Timer Interrupt* on page 327 provides more information about the handling of Fixed Interval Timer interrupts.

The fixed interval timer interrupt handler software should clear GTSR[FIS] before re-enabling MSR[EE] or MSR[GS], to avoid another fixed interval timer interrupt due to the same exception (unless GTCR[FIE] is cleared instead).

9.7 Watchdog Timer

The watchdog timer provides a method for system error recovery in the event that the program running on the A2 core has stalled and cannot be interrupted by the normal interrupt mechanism. The watchdog timer can be configured to cause a critical-class watchdog timer interrupt upon the expiration of a single period of the watchdog timer. It can also be configured to invoke a core-initiated reset upon the expiration of a second period of the watchdog timer.

A watchdog timer exception occurs on a 0→1 transition of a selected bit from the time base. Note that a watchdog timer exception also occurs if the selected time base bit transitions from 0→1 due to an **mtspr** instruction that writes 1 to that time base bit when its previous value was 0.

The Watchdog Timer Period (WP) field of the TCR selects one of 4 bits from the time base, as shown in *Table 9-8*.

Table 8. Watchdog Timer Period Selection

TCR[WP]	Time Base Bit	Period (Time Base Clocks)	Period (32 MHz Clock)	Period (1.6 GHz Clock)	Period (2.3 GHz Clock)
0b00	TBL ₁₃	2 ¹⁹ clocks	16.38 ms	327.68 μs	227.95 μs
0b01	TBL ₉	2 ²³ clocks	262.14 ms	5.24 ms	3.65 ms
0b10	TBL ₇	2 ²⁵ clocks	1.05 s	20.97 ms	14.59 ms
0b11	TBL ₁	2 ³¹ clocks	67.11 s	1.34 s	.93 s

The action taken upon a watchdog timer time-out depends upon the status of the Enable Next Watchdog (ENW) and Watchdog Timer Interrupt Status (WIS) fields of the TSR at the time of the time-out. When TSR[ENW] = 0, the next watchdog timer exception is “disabled”, and the only action to be taken upon the watchdog timer time-out is to set TSR[ENW] to 1. By clearing TSR[ENW], software can guarantee that the time until the next *enabled* watchdog timer exception is *at least* one full Watchdog Timer period (and a maximum of *two* full watchdog timer periods).

When TSR[ENW] = 1, the next watchdog timer exception is enabled, and the action to be taken upon the time-out depends on the value of TSR[WIS]. If TSR[WIS] = 0, then the action is to set TSR[WIS] to 1, at which time a watchdog timer interrupt occurs if enabled by both the Watchdog Timer Interrupt Enable (WIE) field of the TCR and by either the Critical Interrupt Enable (CE) or Guest State (GS) fields of the MSR. The watchdog timer interrupt handler software should clear TSR[WIS] before re-enabling MSR[CE] or MSR[GS], to avoid another watchdog timer interrupt due to the same exception (unless TCR[WIE] is cleared instead).

Section 7.6.14 Guest Fixed-Interval Timer Interrupt on page 327 provides more information about the handling of watchdog timer interrupts.

If TSR[ENW,WIS] is already 0b11 at the time of the next watchdog timer time-out, the action to take depends on the value of the Watchdog Reset Control (WRC) field of the TCR. If TCR[WRC] is nonzero, then a core reset request occurs (see *Core Reset Request and Status Signals* on page 149 for more information about core behavior when a watchdog timer reset request is activated).

Note that once software has set TCR[WRC] to a nonzero value, it cannot be reset by software; this feature prevents errant software from disabling the watchdog timer reset capability.

Table 9-9 summarizes the action to be taken upon a watchdog timer time-out according to the values of TSR[ENW] and TSR[WIS].

Table 9. Watchdog Timer Exception Behavior

TSR[ENW]	TSR[WIS]	Action upon Watchdog Timer Exception
0	0	Set TSR[ENW] to 1.
0	1	Set TSR[ENW] to 1.
1	0	Set TSR[WIS] to 1. If watchdog timer interrupts are enabled (TCR[WIE] = 1 and either MSR[CE] = 1 or MSR[GS] = 1), then interrupt.
1	1	Initiate the watchdog timer reset request specified by TCR[WRC].

A typical system usage of the watchdog timer function is to enable the watchdog timer interrupt and the watchdog timer reset function in the TCR (and MSR), and to start out with both TSR[ENW] and TSR[WIS] clear. Then, a recurring software loop of reliable duration (or perhaps the interrupt handler for a periodic interrupt such as the fixed interval timer interrupt) performs a periodic check of system integrity. Upon successful completion of the system check, software clears TSR[ENW], thereby ensuring that a minimum of one full watchdog timer period and a maximum of two full watchdog timer periods must expire before an enabled watchdog timer exception occurs.

If for some reason the recurring software loop is not successfully completed (and TSR[ENW] thus not cleared) during this period of time, then an enabled watchdog timer exception occurs. This sets TSR[WIS], and a watchdog timer interrupt occurs (if enabled by both TCR[WIE] and MSR[CE] or MSR[GS]). The occurrence of a watchdog timer interrupt in this kind of system is interpreted as a “system error”, insofar as the system was for some reason unable to complete the periodic system integrity check in time to avoid the enabled watchdog timer exception. The action taken by the watchdog timer interrupt handler is of course system-dependent, but typically the software attempts to determine the nature of the problem and correct it if possible. If and when the system attempts to resume operation, the software typically clears both TSR[WIS] and TSR[ENW], thus providing a minimum of another full watchdog timer period for a new system integrity check to occur.

Finally, if for some reason the watchdog timer interrupt is disabled or the Watchdog Timer interrupt handler is unsuccessful in clearing TSR[WIS] and TSR[ENW] before another watchdog timer exception, then the next exception causes a processor reset request to occur, if enabled by the TCR[WRC] field.

Figure 9-2 illustrates the sequence of watchdog timer events that occurs according to this typical system usage.

9.8 Guest Watchdog Timer

The guest watchdog timer provides a method for system error recovery in the event that the program running on the A2 core has stalled and cannot be interrupted by the normal interrupt mechanism. The guest watchdog timer can be configured to cause a critical-class watchdog timer interrupt upon the expiration of a single period of the watchdog timer. It can also be configured to invoke a core-initiated reset upon the expiration of a second period of the watchdog timer.

A guest watchdog timer exception occurs on a 0→1 transition of a selected bit from the time base. Note that a guest watchdog timer exception also occurs if the selected time base bit transitions from 0→1 due to an **mtspr** instruction that writes 1 to that time base bit when its previous value was 0.

The Guest Watchdog Timer Period (WP) field of the GTCR selects one of 4 bits from the time base, as shown in Table 9-8.

Table 10. Guest Watchdog Timer Period Selection

TCR[WP]	Time Base Bit	Period (Time Base Clocks)	Period (32 MHz Clock)	Period (1.6 GHz Clock)	Period (2.3 GHz Clock)
0b00	TBL ₁₃	2 ¹⁹ clocks	16.38 ms	327.68 μs	227.95 μs
0b01	TBL ₉	2 ²³ clocks	262.14 ms	5.24 ms	3.65 ms
0b10	TBL ₇	2 ²⁵ clocks	1.05 s	20.97 ms	14.59 ms
0b11	TBL ₁	2 ³¹ clocks	67.11 s	1.34 s	.93 s

The action taken upon a watchdog timer time-out depends upon the status of the Enable Next Watchdog (ENW) and Watchdog Timer Interrupt Status (WIS) fields of the GTSR at the time of the time-out. When $GTSR[ENW] = 0$, the next watchdog timer exception is “disabled”, and the only action to be taken upon the watchdog timer time-out is to set $GTSR[ENW]$ to 1. By clearing $GTSR[ENW]$, software can guarantee that the time until the next *enabled* watchdog timer exception is *at least* one full Watchdog Timer period (and a maximum of *two* full watchdog timer periods).

When $GTSR[ENW] = 1$, the next watchdog timer exception is enabled, and the action to be taken upon the time-out depends on the value of $GTSR[WIS]$. If $GTSR[WIS] = 0$, then the action is to set $GTSR[WIS]$ to 1, at which time a watchdog timer interrupt occurs if enabled by both the Watchdog Timer Interrupt Enable (WIE) field of the TCR and by either the Critical Interrupt Enable (CE) or Guest State (GS) fields of the MSR. The watchdog timer interrupt handler software should clear $GTSR[WIS]$ before re-enabling $MSR[CE]$ or $MSR[GS]$, to avoid another watchdog timer interrupt due to the same exception (unless $GTCR[WIE]$ is cleared instead). *Section 7.6.14 Guest Fixed-Interval Timer Interrupt* on page 327 provides more information about the handling of watchdog timer interrupts.

If $GTSR[ENW, WIS]$ is already 0b11 at the time of the next watchdog timer time-out, the action to take depends on the value of the Watchdog Reset Control (WRC) field of the GTCR. If $GTCR[WRC]$ is nonzero, then a core reset request occurs (see *Core Reset Request and Status Signals* on page 149 for more information about core behavior when a watchdog timer reset request is activated).

Note that once software has set $GTCR[WRC]$ to a nonzero value, it cannot be reset by software; this feature prevents errant software from disabling the watchdog timer reset capability.

Table 9-9 summarizes the action to be taken upon a watchdog timer time-out according to the values of $GTSR[ENW]$ and $GTSR[WIS]$.

Table 11. Guest Watchdog Timer Exception Behavior

TSR[ENW]	TSR[WIS]	Action upon Watchdog Timer Exception
0	0	Set $GTSR[ENW]$ to 1.
0	1	Set $GTSR[ENW]$ to 1.
1	0	Set $GTSR[WIS]$ to 1. If watchdog timer interrupts are enabled ($GTCR[WIE] = 1$ and either $MSR[CE] = 1$ or $MSR[GS] = 1$), then interrupt.
1	1	Initiate the watchdog timer reset request specified by $GTCR[WRC]$.

A typical system usage of the watchdog timer function is to enable the watchdog timer interrupt and the watchdog timer reset function in the GTCR (and MSR), and to start out with both $GTSR[ENW]$ and $GTSR[WIS]$ clear. Then, a recurring software loop of reliable duration (or perhaps the interrupt handler for a periodic interrupt such as the fixed interval timer interrupt) performs a periodic check of system integrity. Upon successful completion of the system check, software clears $GTSR[ENW]$, thereby ensuring that a minimum of one full watchdog timer period and a maximum of two full watchdog timer periods must expire before an enabled watchdog timer exception occurs.

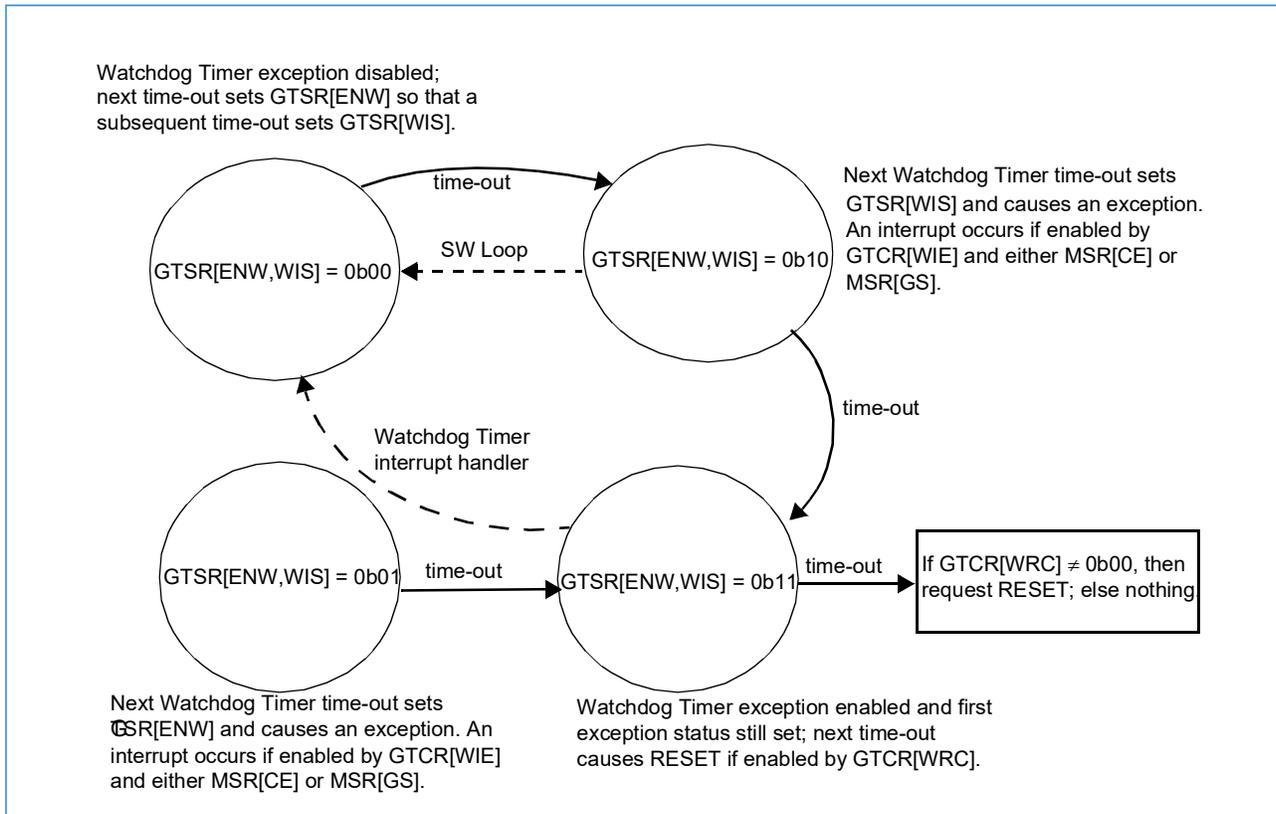
If for some reason the recurring software loop is not successfully completed (and $GTSR[ENW]$ thus not cleared) during this period of time, then an enabled watchdog timer exception occurs. This sets $GTSR[WIS]$, and a watchdog timer interrupt occurs (if enabled by both $GTCR[WIE]$ and $MSR[CE]$ or $MSR[GS]$). The occurrence of a watchdog timer interrupt in this kind of system is interpreted as a “system error”, insofar as the system was for some reason unable to complete the periodic system integrity check in time to avoid the enabled watchdog timer exception. The action taken by the watchdog timer interrupt handler is of course system-dependent, but typically the software attempts to determine the nature of the problem and correct it if

possible. If and when the system attempts to resume operation, the software typically clears both GTSR[WIS] and GTSR[ENW], thus providing a minimum of another full watchdog timer period for a new system integrity check to occur.

Finally, if for some reason the watchdog timer interrupt is disabled or the Watchdog Timer interrupt handler is unsuccessful in clearing GTSR[WIS] and GTSR[ENW] before another watchdog timer exception, then the next exception causes a processor reset request to occur, if enabled by the GTCR[WRC] field.

Figure 9-2 illustrates the sequence of watchdog timer events that occurs according to this typical system usage.

Figure 4. Guest Watchdog State Machine



9.9 Timer Control Register (TCR)

The TCR is a privileged SPR that controls DEC, UDEC, FIT, and watchdog timer operation. The TCR is read into a GPR using `mfspr` and is written from a GPR using `mtspr`.

The Watchdog Timer Reset Control (WRC) field of the TCR is cleared to 0 by processor reset (see *Initialization* on page 149). Each bit of this 2-bit field is set only by software and is cleared only by hardware. For each bit of the field, after software has written it to 1, that bit remains 1 until a processor reset occurs. This is to prevent errant code from disabling the watchdog timer reset function.

The Auto-Reload Enable (ARE) field of the TCR is also cleared to zero by a processor reset. This disables the auto-reload feature of the DEC.

Register Short Name:	TCR	Read Access:	Hypv
Decimal SPR Number:	340	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	WP	0b00	<p><u>Watchdog Timer Period</u> Specifies one of 4 bit locations of the TimeBase used to signal a Watchdog Timer exception on a transition from 0 to 1.</p> <p>00 2¹⁹ time base clocks 01 2²³ time base clocks 10 2²⁵ time base clocks 11 2³¹ time base clocks</p>
34:35	WRC	0b00	<p><u>Watchdog Timer Reset Control</u> 00 NoReset: No Watchdog Timer reset request will occur 01 Reset1 request 10 Reset2 request 11 Reset3 request Note: - Type of reset request to cause upon Watchdog Timer exception with TSR[ENW,WIS]=0b11. - These bits are set only by software. Once a 1 has been written to one of these bits, that bit remains a 1 until a reset request occurs. This is to prevent errant code from disabling the Watchdog reset function.</p>
36	WIE	0b0	<p><u>Watchdog Timer Interrupt Enable</u> 0 Disable Watchdog Timer interrupt 1 Enable Watchdog Timer interrupt</p>
37	DIE	0b0	<p><u>Decrementer Interrupt Enable</u> 0 Disable Decrementer interrupt 1 Enable Decrementer interrupt</p>
38:39	FP	0b00	<p><u>Fixed-Interval Timer Period</u> Specifies one of 4 bit locations of the TimeBase used to signal a Fixed-Interval Timer exception on a transition from 0 to 1.</p> <p>00 2¹¹ time base clocks 01 2¹⁵ time base clocks 10 2¹⁹ time base clocks 11 2²³ time base clocks</p>
40	FIE	0b0	<p><u>Fixed-Interval Timer Interrupt Enable</u> 0 Disable Fixed Interval Timer interrupt 1 Enable Fixed Interval Timer interrupt</p>
41	ARE	0b0	<p><u>Auto-Reload Enable</u> 0 Disable auto reload 1 Enable auto reload</p>
42	UDIE	0b0	<p><u>User Decrementer Interrupt Enable</u> 0 Disable User Decrementer interrupt 1 Enable User Decrementer interrupt</p>
43:50	///	0x0	<u>Reserved</u>

Bit(s):	Field Name:	Init	Description
51	UD	0b0	<u>User Decrementer Available</u> 0 mtspr or mfspr to the UDEC register causes a Illegal Instruction exception 1 mtspr or mfspr to the UDEC register succeeds Note: Changing this bit requires a CSI for the next instruction to see the new context
52:63	///	0x0	<u>Reserved</u>

9.10 Guest Timer Control Register (GTCR)

The GTCR

9.11 Timer Status Register (TSR)

The TSR is a privileged SPR that records the status of DEC, UDEC, FIT, and watchdog timer events. The fields of the TSR are generally set to 1 only by hardware and cleared to 0 only by software. Hardware cannot clear any fields in the TSR, nor can software set any fields. Software can read the TSR into a GPR using **mfspr**. Clearing the TSR is performed using **mtspr** by placing a 1 in the GPR source register in all bit positions that are to be cleared in the TSR, and a 0 in all other bit positions. The data written from the GPR to the TSR is not direct data, but a mask. A 1 clears the bit and a 0 leaves the corresponding TSR bit unchanged.

Register Short Name:	TSR	Read Access:	Hypv
Decimal SPR Number:	336	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	WC,AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	ENW	0b0	<u>Enable Next Watchdog Timer</u> 0 Action taken on next Watchdog Timer exception will be to set TSR{ENW} 1 Action taken on next Watchdog Timer exception is governed by TSR{WIS}
33	WIS	0b0	<u>Watchdog Timer Interrupt Status</u> 0 A Watchdog Timer event has not occurred. 1 A Watchdog Timer event has occurred. When (MSR{CE}=1 or MSR{GS}=1) and TCR{WIE}=1, a Watchdog Timer interrupt is taken.
34:35	WRS	0b00	<u>Watchdog Timer Reset Status</u> 00 No Reset: No Watchdog Timer reset has occurred 01 Reset1: A Watchdog Timer initiated Reset1 reset occurred 10 Reset2: A Watchdog Timer initiated Reset2 reset occurred 11 Reset3: A Watchdog Timer initiated Reset3 reset occurred
36	DIS	0b0	<u>Decrementer Interrupt Status</u> A Decrementer event has occurred
37	FIS	0b0	<u>Fixed-Interval Timer Interrupt Status</u> A Fixed-Interval Timer event has occurred

Bit(s):	Field Name:	Init	Description
38	UDIS	0b0	<u>User Decrementer Interrupt Status</u> A User Decrementer event has occurred
39:63	///	0x0	<u>Reserved</u>

9.12 Guest Timer Status Register (GTSR)

The GTSR

9.13 Freezing the Timer Facilities

The debug mechanism provides a means for temporarily “freezing” the timers upon a debug exception. Specifically, the time base and decremter can be prevented from incrementing and decrementing, respectively, whenever a debug exception is recorded in the Debug Status Register (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 *Debug Facilities* on page 389 for more information about freezing the timers.

9.14 Selection of the Timer Clock Source

The source clock of the timers is selected by the Timer Clock Select () field of Execution Unit Configuration Register 0 (XUCR0). When set to zero, XUCR0[TCS] selects the CPU clock. This is the highest frequency timer clock source.

When set to one, XUCR0[TCS] selects an A2 core input (*an_ac_tb_update_pulse*) as the timer clock. The input is sampled by a latch clocked by the CPU clock, and so cannot cycle any faster than half the frequency of the CPU clock. Both rising and falling edges of the external timer pulse are sampled, so that the actual timer clock will be twice the frequency of the external update pulse.

9.15 Selecting the Timer Clock Frequency Divide Value

The Timer Clock Divide field of Execution Unit Configuration Register 4 allows selection of four possible divide values for the timer update pulse frequency. XUCR4[TCD] can be used to select the following timer clock divide options: not divided, divide by 4, divide by 8, and divide by 16. The default (reset) value is to select the undivided input timer frequency. The selected divide-by value is applied to both the internal core clock frequency, as well as the external timer input pulse.

9.16 Synchronizing Timers Across Multiple Cores

In applications involving multiple A2 cores, the *an_ac_tb_update_enable* input can be used to stop timer clock pulses from incrementing the timers. The purpose of this function is for software to initialize the timer facilities in all cores before enabling the update pulse; thereby synchronizing all time base counters.

When set to zero, the *an_ac_tb_update_enable* input signal blocks the timer clock from incrementing timer facilities. This control applies to either setting of XUCR0[TCS]; whether the timer clock source is internally or externally generated.



10. Debug Facilities

The debug facilities of the A2O Core include support for several debug modes for debugging during hardware and software development, as well as debug events that allow developers to control the debug process. Debug registers control these debug modes and debug events. The debug registers can be accessed either through software running on the processor or through the JTAG port via the SCOM interface of the A2O Core. Access to the debug facilities through the JTAG port is typically provided by a debug tool such as the RISCWatch development tool from IBM. A trace bus, which enables the tracing of code running in real time, is also provided.

10.1 Implications of Hypervisor on Debug Controls

The Power ISA Embedded.Hypervisor category provides several controls that allow debug operation within the A2 core. All debug events are dependent on the state of EPCR[DUVD] and MSR[GS] to determine if debug events can occur when executing in the hypervisor state (MSR[GS,PR] = 00). When EPCR[DUVD] = 1, debug events are enabled for guest state (MSR[GS] = 1) operation only. This stops debug events intended for code executing in the guest state from occurring when the hypervisor is active. Another control determines if the MSR[DE] bit can be modified when executing in guest state. If MSR[DEP] is set to 1, a guest state operation to MSR[DE] is ignored.

DBSR is a hypervisor accessible debug register. It allows code executing in hypervisor state to set bits in the DBSR. DBSRWR is bit-for-bit compatible with the DBSR register. When a 1 is written to DBSRWR, the corresponding DBSR bit is set. The DBSRWR register is shown in Section 10.7.6 on page 413.

10.2 Support for Development Tools

The RISCWatch product from IBM is an example of a development tool that uses external debug mode, debug events, and the JTAG interface to implement a hardware and software development tool. Registers in the A2 core are not directly accessible to JTAG, but instead are converted to a different serial interface through a chip level SCOM controller.

10.3 Debug Modes

The following sections describe the various debug modes supported by the A2O Core. Each of these debug modes supports a particular type of debug tool or debug task commonly used in embedded systems development. For internal and external debug modes, the various debug event types are enabled by the setting of corresponding fields in Debug Control Register 0 (DBCRO) or Debug Control Register 3 (DBCRC3), and upon their occurrence are recorded in the Debug Status Register (DBSR). The trace debug mode is controlled through various SCOM accessible registers that enable the selection of debug and trigger signals that are sent out on the external trace and trigger buses.

There are three debug modes:

- Internal debug mode
- External debug mode
- Trace debug mode

The *Power ISA* specification deals only with internal debug mode and the relationship of debug interrupts to the rest of the interrupt architecture. Internal debug mode is the mode that involves debug software running on the processor itself, typically in the form of the debug interrupt handler. The other debug modes, are outside the scope of the architecture, and involve special-purpose debug hardware external to the A2O Core, connected either to the JTAG interface (for external debug mode) and/or the external trace array (for trace debug mode). Details of these interfaces and their operation are beyond the scope of this manual.

10.3.1 Internal Debug Mode

Internal debug mode provides access to architected processor resources and supports setting hardware and software breakpoints and monitoring processor status. In this mode, debug events are considered exceptions. Exceptions, in addition to recording their status in the DBSR, generate debug interrupts if and when such interrupts are enabled (Machine State Register (MSR) DE field is 1; see *CPU Interrupts and Exceptions* on page 277 for a description of the MSR and debug interrupts). When a debug interrupt occurs, special debugger software at the interrupt handler can check processor status and other conditions related to the debug event, as well as alter processor resources using all of the instructions defined for the A2O Core.

Internal debug mode relies on this interrupt handling software at the debug interrupt vector to debug software problems. This mode, used while the processor executes instructions, enables debugging of both application programs and operating system software, including all of the noncritical class interrupt handlers.

In this mode, the debugger software can communicate with the outside world through a communications port, such as a serial port, external to the processor core.

To enable internal debug mode, the IDM field of DBCR0 must be set to 1 (DBCR0[IDM] = 1). This mode can be enabled in combination with external debug mode (see *External Debug Mode* below).

10.3.2 External Debug Mode

External debug mode provides access to architected processor resources and supports stopping, starting, and stepping the processor; setting hardware and software breakpoints; monitoring processor status; and other operations. External debug mode is enabled by setting the SCOM-accessible PC Configuration Register 0, Debug Action Select, (PCCR0[DBA]) bits to a valid decode. Each thread has separate PCCR0[DBA] bits, allowing independent control over external debug action selection. In this mode, debug events record their status in the DBSR, and then cause the processor to take actions based on the state of the encoded bits set in PCCR0[DBA]. A description of the external debug actions supported through the PCCR0[DBA] field is shown in *Table 10-1*.

Table 1. PCCR0[DBA] (Debug Action) Definition per Thread (Sheet 1 of 2)

Encode	Action	Description
000	No Action	No action taken.
001	Reserved	
010	Stop N	Stop this thread at the current instruction.
011	Stop Core	Stop all threads at the current instruction.
100	Activate Error Signal	An error bit in this thread's Fault Isolation Register (FIR) will be set. Depending on FIR mask settings, a recoverable or checkstop error signal can be driven outside the core.

Table 1. PCCR0[DBA] (Debug Action) Definition per Thread (Sheet 2 of 2)

Encode	Action	Description
101	Activate External Signal	Activates an external trigger signal (ac_an_debug_trigger pulse) outside the core. Chip-level logic can use this to perform other actions, such as stopping all cores.
110	Activate External Signal and Stop N	Stops this thread at the current instruction and activates the external trigger signal.
111	Activate External Signal and Stop Core	Stops all threads at the current instruction and activates the external trigger signal.

The PCCR0[DBA] bits provide options for stopping a thread or all threads on the core. It also allows sending an error signal to external logic from the local FIR. If chip clock controls are enabled to stop the clock, they can force the core clocks off, facilitating interrogation of all core latches and arrays via scanning. An external trigger signal (single cycle pulse) can also be selected to trigger chip level controls upon activation of the debug compare event. Use of this signal is implementation dependent.

When a debug event occurs with PCCR0[DBA] set to stop the processor (encodes 2, 3, 6, or 7), normal instruction execution stops and architected processor resources and memory can be accessed and altered via the JTAG interface. While in the stop state, external interrupts as well as time base and decremter updates can be temporarily disabled through options in the THRCTL register (see *Thread Control and Status Register (THRCTL) Definition* on page 424). When a debug event occurs with PCCR0[DBA] set to any other value, the stop state is not entered and instruction processing continues.

Storage access control by a memory management  (MMU) remains in effect while in external debug mode; the debugger might need to modify MSR or TLB values to access protected memory.

External debug mode relies only on internal processor resources, and no debug interrupt handling software, so it can be used to debug both system hardware and software problems. This mode can also be used for software development on systems without a control program, or to debug control program problems, including problems within the debug interrupt handler itself, or within any other critical class interrupt handlers.

External debug mode can be enabled in combination with internal debug mode (see *Internal Debug Mode* on page 390). External debug mode takes precedence over internal debug mode. That is, debug events that cause the processor to stop (when enabled by PCCR0[DBA]) enter the stop state rather than generating a debug interrupt, although a debug interrupt might be pending while the processor is in the stop state.

10.3.3 Trace Debug Mode

The A2 core tracing capability is separate from the other debug modes. It can be used independent from, or in conjunction with, the other debug resources. A 64-bit debug bus provides signals to tracing facilities external to the core. Each unit within the core has a SCOM-accessible register for selecting and routing signal groups onto the debug bus. Signal groups from multiple units within the core can be routed onto the debug bus at the same time. Additional controls select and route groups of trigger signals onto a separate trigger bus. Together, the debug and trigger buses provide external tracing facilities with the capability to monitor and trigger on a large number of core signals. See on page 435 for additional information.

10.4 Debug Events

There are several different kinds of debug events, each of which is enabled by a field in DBCR0 or DBCR3 (except for the unconditional debug event) and recorded in the DBSR. *Debug Modes* on page 389 describes the operation that results when a debug event occurs while operating in any of the debug modes.

Table 10-2 lists the various debug events recognized by the A20 Core. Detailed explanations of each debug event type follow the table.

Table 2. Debug Events

Event	Description
Instruction Address Compare (IAC)	Caused by the attempted execution of an instruction for which the address matches the conditions specified by DBCR0, DBCR1, and the IAC1–IAC4 registers.
Data Address Compare (DAC)	Caused by the attempted execution of a load, store, or cache management instruction for which the data storage address matches the conditions specified by DBCR0, DBCR2, DBCR3, and the DAC1–DAC4 registers.
Data Value Compare (DVC)	Caused by the attempted execution of a load or store instruction for which the data storage address matches the conditions specified by DBCR0, DBCR2, and the DAC1–DAC2 registers, and for which the referenced data matches the value specified by the DVC1–DVC2 registers.
Branch Taken (BRT)	Caused by the attempted execution of a branch instruction for which the branch conditions are met (that is, for a branch instruction that results in the redirection of the instruction stream).
Trap (TRAP)	Caused by the attempted execution of a tw , twi , td , or tdi instruction for which the trap conditions are met.
Return (RET)	Caused by the attempted execution of an rfi instruction.
Instruction Complete (ICMP)	Caused by the successful completion of the execution of any instruction.
Interrupt (IRPT)	Caused by the generation of a base class interrupt.
Unconditional (UDE)	Caused by the assertion of an unconditional debug event request through the THRCTL[UDE] bits. The THRCTL register is accessed via the SCOM interface to the A2 core.
Instruction Value Compare (IVC)	Caused by the attempted execution of an instruction for which the instruction data matches the value specified by the IMR and IMMR registers. The IVC debug event is enabled by setting DBCR3[IVC].

10.4.1 Instruction Address Compare (IAC) Debug Event

IAC debug events occur when execution is attempted of an instruction for which the instruction address and other parameters match the IAC conditions specified by DBCR0, DBCR1, and the IAC registers. There are four IAC registers on the A20 Core, IAC1–IAC4. Depending on the IAC mode specified by DBCR1, these IAC registers can be used to specify four independent, exact IAC addresses, or they can be configured in pairs (IAC1/IAC2 and IAC3/IAC4) to match a masked instruction address for which IAC debug events should occur. In 32-bit mode, the lower 32 bits of IAC1–IAC4 are compared against the lower 32 bits of the instruction address.

10.4.1.1 IAC Debug Event Fields

Several fields in DBCR0 and DBCR1 are used to specify the IAC conditions, as follows:

IAC Event Enable Field

DBCRO[IAC1, IAC2, IAC3, IAC4] are the individual IAC event enables for each of the four IAC events: IAC1, IAC2, IAC3, and IAC4. For a given IAC event to occur, the corresponding IAC event enable bit in DBCR0 must be set. When a given IAC event occurs, the corresponding DBSR[IAC1, IAC2, IAC3, IAC4] bit is set.

IAC Mode Field

DBCRI[IAC12M, IAC34M] control the comparison mode for the IAC1/IAC2 and IAC3/IAC4 events, respectively. There are two comparison modes supported by the A2O Core:

- Exact comparison mode (DBCRI[IAC12M/IAC34M] = 0b0)

In this mode, the instruction address is compared to the value in the corresponding IAC register; the IAC event occurs only if the comparison is an exact match. When the processor is operating in 32-bit mode (MSR[CM] = 0), the addresses are masked to compare only bits 32 through 63.

- Address bit match mode (DBCRI[IAC12M/IAC34M] = 0b1)

In this mode, the IAC1 or IAC2 event occurs only if the instruction address matches the value in the IAC1 register, as masked by the value in the IAC2 register. That is, the IAC1 register specifies an address value, and the IAC2 register specifies an *address bit mask* that determines which bit of the instruction address should participate in the comparison to the IAC1 value. For every bit set to 1 in the IAC2 register, the corresponding instruction address bit must match the value of the same bit position in the IAC1 register. For every bit set to 0 in the IAC2 register, the corresponding address bit comparison does not affect the result of the IAC event determination. Similarly, instruction address matches for IAC3 and IAC4 events occur as described previously for IAC1 and IAC2. When the processor is operating in 32-bit mode (MSR[CM] = 0), the addresses are masked to compare only bits 32 through 63.

When the instruction address matches the address bit mask mode conditions, either one or both of the IAC debug event bits is set in the DBSR, as determined by which IAC event enable bits are set in DBCR0. That is, when an address bit mask mode IAC debug event occurs, the setting of DBCRO[IAC1, IAC2] determines whether one or the other or both of the DBSR[IAC1, IAC2] bits are set. In like manner, the setting of DBCRO[IAC3, IAC4] determines how the DBSR[IAC3, IAC4] bits are set. It is a programming error to set the IAC mode field to address bit mask mode for IAC12M or IAC34M without also enabling at least one of the paired IAC event enable bits in DBCR0 (IAC1/IAC2 or IAC3/IAC4 respectively).

- The A2 core does not support the IAC range inclusive comparison mode.
- The A2 core does not support the IAC range exclusive comparison mode.

IAC User/Supervisor Field

DBCRI[IAC1US, IAC2US, IAC3US, IAC4US] are the individual IAC user/supervisor fields for each of the four IAC events. The IAC user/supervisor fields specify the operating mode of the processor in order for the corresponding IAC event to occur. The operating mode is determined by the Problem State field of the Machine State Register (MSR[PR]; see *Section 2.4.2.4 Machine State Register* on page 82). When the IAC user/supervisor field is 0b00, the operating mode does not matter; the IAC debug event can occur independent of the state of MSR[PR]. When this field is 0b10, the processor must be operating in

supervisor state ($\text{MSR}[\text{PR}] = 0$). When this field is 0b11, the processor must be operating in user mode ($\text{MSR}[\text{PR}] = 1$). The IAC user/supervisor field value of 0b01 is reserved.

If the IAC is set to the address bit match mode, it is a programming error (and the results of any instruction address comparison are undefined) if the paired IAC user/supervisor field settings ($\text{DBCR1}[\text{IAC1US}]/\text{DBCR1}[\text{IAC2US}]$ or $\text{DBCR1}[\text{IAC3US}]/\text{DBCR1}[\text{IAC4US}]$) are not set to the same value.

IAC Effective/Real Address Field

$\text{DBCR1}[\text{IAC1ER}, \text{IAC2ER}, \text{IAC3ER}, \text{IAC4ER}]$ are the individual IAC effective/real address fields for each of the four IAC events. The IAC effective/real address fields specify whether the instruction address comparison should be performed using the effective, virtual, or real address (see *Memory Management* on page 169 for an explanation of these different types of addresses). When the IAC effective/real address field is 0b00, the comparison is performed using the effective address only—the IAC debug event can occur independent of the instruction address space ($\text{MSR}[\text{IS}]$). When this field is 0b10, the IAC debug event occurs only if the effective address matches the IAC conditions and is in virtual address space 0 ($\text{MSR}[\text{IS}] = 0$). Similarly, when this field is 0b11, the IAC debug event occurs only if the effective address matches the IAC conditions and is in virtual address space 1 ($\text{MSR}[\text{IS}] = 1$). Note that in these latter two modes, in which the virtual address space of the instruction is considered, it is not the entire virtual address that is considered. The Process ID, which forms the final part of the virtual address, is not considered. Finally, the IAC effective/real address field value of 0b01 is reserved, and corresponds to the Power ISA architected real address comparison mode, which is not supported by the A2O Core.

If the IAC is set to the address bit match mode, it is a programming error (and the results of any instruction address comparison are undefined) if the paired IAC effective/real field settings ($\text{DBCR1}[\text{IAC1ER}]/\text{DBCR1}[\text{IAC2ER}]$ or $\text{DBCR1}[\text{IAC3ER}]/\text{DBCR1}[\text{IAC4ER}]$) are not set to the same value.

10.4.1.2 IAC Debug Event Processing

When enabled, the occurrence of an IAC debug event is recorded in the corresponding bit of the DBSR. If debug interrupts are not enabled ($\text{MSR}[\text{DE}] = 0$) the imprecise debug event ($\text{DBSR}[\text{IDE}]$) bit is also set. The resulting actions taken by the processor due to the IAC debug event depend on the specific debug configuration.

When operating in external debug mode ($\text{DBCR0}[\text{EDM}] = 1$), the setting of $\text{PCCR0}[\text{DBA}]$ determines the resulting debug actions. If the debug action is a stop, the processor enters the stop state and ceases the processing of instructions. The program counter contains the address of the instruction that caused the IAC match to occur. If the $\text{PCCR0}[\text{DBA}]$ decode does not stop the processor, instruction execution continues, and any additional debug actions are determined by the setting of $\text{DBCR0}[\text{IDM}]$.

When operating in internal debug mode ($\text{DBCR0}[\text{IDM}] = 1$) with debug interrupts enabled ($\text{MSR}[\text{DE}] = 1$), a debug interrupt occurs with Critical Save/Restore Register 0 (CSRR0) set to the address of the instruction that caused the IAC debug event. When operating in internal debug mode with debug interrupts disabled ($\text{MSR}[\text{DE}] = 0$), the debug interrupt does not occur immediately. Instead, instruction execution continues, and a debug interrupt occurs if and when $\text{MSR}[\text{DE}]$ is set to 1. This enables debug interrupts, assuming software has not cleared the IAC debug event status from the DBSR in the meantime. Upon such a “delayed” interrupt, the debug interrupt handler software can query the $\text{DBSR}[\text{IDE}]$ field to determine that the debug interrupt has occurred imprecisely.

10.4.2 Data Address Compare (DAC) Debug Event

DAC debug events occur when execution is attempted of a load, store, or cache management instruction for which the data storage address and other parameters match the DAC conditions specified by DBCR0, DBCR2, DBCR3, and the DAC registers. There are four DAC registers on the A2O Core, DAC1 through DAC4. Depending on the DAC mode specified by DBCR2 and DBCR3, these DAC registers can be used to specify four independent, exact DAC addresses, or they can be configured to operate as a pair (DAC1/DAC2 and DAC3/DAC4). When operating as a pair, they can specify a combination of an address and an *address bit mask* for selective comparison with the data storage address. Note that for all load, store and cache management instructions, the address that is used in the DAC comparison is the effective address as presented to the ERATs.

10.4.2.1 DAC Debug Event Fields

There are several fields in DBCR0, DBCR2, and DBCR3 that are used to specify the DAC conditions, as follows:

DAC Event Enable Field

DBCR0[*DAC1*, *DAC2*, *DAC3*, *DAC4*] are the event enables for the four DAC events. For each of the DAC events, there is one enable for DAC read events (*DAC1R*, *DAC2R*, *DAC3R*, *DAC4R*) and another for DAC write events (*DAC1W*, *DAC2W*, *DAC3W*, *DAC4W*). Load, **dcbt**, **dcbtstp**, **dcbtst**, **dcbtstep**, **dcbtls**, **dcbtstls**, **dcblc**, **icbi**, **icbiep**, **icblc**, **icbt**, and **icbtls** instructions can cause DAC read events; while store, **dcbf**, **dcbfep**, **dcbi**, **dcbst**, **dcbstp**, **dcbz**, and **dcbzep** instructions can cause DAC write events (see *DAC Debug Events Applied to Various Instruction Types* on page 398 for more information about these instructions and the types of DAC debug events they can cause). For a given DAC event to occur, the corresponding DAC event enable bit in DBCR0 for the particular operation type must be set. When a DAC event occurs, the corresponding DBCR[*DAC1R*, *DAC1W*, *DAC2R*, *DAC2W*, *DAC3R*, *DAC3W*, *DAC4R*, *DAC4W*] bit is set. The DBCR bits for DAC1 and DAC2 events are shared by DVC debug events (see *Data Value Compare (DVC) Debug Event* on page 399).

DAC Mode Field

DBCR2[*DAC12M*] and DBCR3[*DAC34M*] control the comparison mode for the DAC1, DAC2, DAC3, and DAC4 events respectively. There are two comparison modes supported by the A2O Core:

- Exact comparison mode (DBCR2[*DAC12M*] = 0b0; DBCR3[*DAC34M*] = 0b0)

In this mode, the data address is compared to the value in the corresponding DAC register, and the DAC event occurs *only if the comparison is an exact match*. When the processor is operating in 32-bit mode (MSR[*CM*] = 0), the addresses are masked to compare only bits 32 through 63.

- Address bit mask mode (DBCR2[*DAC12M*] = 0b1; DBCR3[*DAC34M*] = 0b1)

In this mode, the DAC1 or DAC2 event occurs only if the data address matches the value in the DAC1 register, as masked by the value in the DAC2 register. That is, the DAC1 register specifies an address value, and the DAC2 register specifies an *address bit mask* that determines which bit of the data address should participate in the comparison to the DAC1 value. For every bit set to 1 in the DAC2 register, the corresponding data address bit must match the value of the same bit position in the DAC1 register. For every bit set to 0 in the DAC2 register, the corresponding address bit comparison does not affect the result of the DAC event determination. Similarly, data address matches for DAC3 and DAC4 events occur as described previously for DAC1 and DAC2. When the processor is

operating in 32-bit mode ($\text{MSR}[\text{CM}] = 0$), the addresses are masked to compare only bits 32 through 63.

This comparison mode is useful for detecting accesses to a particular byte address, when the accesses can be of various sizes. For example, if the debugger is interested in detecting accesses to byte address $0x0000_0000_0000_0003$, then these accesses can occur due to a byte access to that specific address, or due to a halfword access to address $0x0000_0000_0000_0002$, or due to a word access to address $0x0000_0000$. By using address bit mask mode and specifying that the low-order two bits of the address should be ignored (that is, setting the address bit mask in DAC2 to $0xFFFF_FFFF_FFFF_FFFC$), the debugger can detect each of these types of access to byte address $0x0000_0000_0000_0003$.

When the data address matches the address bit mask mode conditions, either one or both of the DAC debug event bits corresponding to the operation type (read or write) is set in the DBSR, as determined by which of the corresponding four DAC event enable bits are set in DBCR0. That is, when an address bit mask mode DAC debug event occurs, the setting of $\text{DBCR0}[\text{DAC1R}, \text{DAC1W}, \text{DAC2R}, \text{DAC2W}]$ determines whether one or the other or both of the $\text{DBSR}[\text{DAC1R}, \text{DAC1W}, \text{DAC2R}, \text{DAC2W}]$ bits corresponding to the operation type are set. In like manner, the setting of $\text{DBCR0}[\text{DAC3R}, \text{DAC3W}, \text{DAC4R}, \text{DAC4W}]$ determines how the $\text{DBSR}[\text{DAC3R}, \text{DAC3W}, \text{DAC4R}, \text{DAC4W}]$ bits are set. It is a programming error to set the DAC mode field to address bit mask mode for DAC12M or DAC34M without also enabling at least one of the paired DAC event enable bits in DBCR0 (DAC1/DAC2 or DAC3/DAC4 respectively).

- The A2 core does not support the DAC range inclusive comparison mode.
- The A2 core does not support the DAC range exclusive comparison mode.

DAC User/Supervisor Field

$\text{DBCR2}[\text{DAC1US}, \text{DAC2US}]$ and $\text{DBCR3}[\text{DAC3US}, \text{DAC4US}]$ are the individual DAC user/supervisor fields for the four DAC events. The DAC user/supervisor fields specify the operating mode of the processor in order for the corresponding DAC event to occur. The operating mode is determined by the Problem State field of the Machine State Register ($\text{MSR}[\text{PR}]$; see *Section 2.4.2.4 Machine State Register* on page 82). When the DAC user/supervisor field is $0b00$, the operating mode does not matter—the DAC debug event can occur independent of the state of $\text{MSR}[\text{PR}]$. When this field is $0b10$, the processor must be operating in supervisor state ($\text{MSR}[\text{PR}] = 0$). When this field is $0b11$, the processor must be operating in user mode ($\text{MSR}[\text{PR}] = 1$). The DAC user/supervisor field value of $0b01$ is reserved.

If the DAC is set to the address bit mask mode, it is a programming error (and the results of any data address comparison are undefined) if the paired DAC user/supervisor field settings ($\text{DBCR2}[\text{DAC1US}]$ and $\text{DBCR2}[\text{DAC2US}]$ or $\text{DBCR3}[\text{DAC3US}]$ and $\text{DBCR3}[\text{DAC4US}]$) are not set to the same value.

DAC Effective/Real Address Field

$\text{DBCR2}[\text{DAC1ER}, \text{DAC2ER}]$ and $\text{DBCR3}[\text{DAC3ER}, \text{DAC4ER}]$ are the individual DAC effective/real address fields for the four DAC events. The DAC effective/real address fields specify whether the instruction address comparison should be performed using the effective, virtual, or real address (see *Memory Management* on page 169 for an explanation of these different types of addresses). When the DAC effective/real address field is $0b00$, the comparison is performed using the effective address only; the DAC debug event can occur independent of the data address space ($\text{MSR}[\text{DS}]$). When this field is $0b10$, the DAC debug event occurs only if the effective address matches the DAC conditions and is in virtual address space 0 ($\text{MSR}[\text{DS}] = 0$). Similarly, when this field is $0b11$, the DAC debug event occurs only if the effective address matches the DAC conditions and is in virtual address space 1 ($\text{MSR}[\text{DS}] = 1$). Note that in these latter two modes, in which the virtual address space of the data is considered, it is not the

entire virtual address that is considered. The Process ID, which forms the final part of the virtual address, is not considered. Finally, the DAC effective/real address field value of 0b01 is reserved, and corresponds to the Power ISA architected real address comparison mode, which is not supported by the A20 Core.

If the DAC is set to the address bit mask mode, it is a programming error (and the results of any data address comparison are undefined) if the paired DAC effective/real address field settings (DBCR2[*DAC1ER*] and DBCR2[*DAC2ER*] or DBCR3[*DAC3ER*] and DBCR3[*DAC4ER*]) are not set to the same value.

Data Value Compare Mode Field

DBCR2[*DVC1M*, *DVC2M*] are the data value compare (DVC) mode enable bits. This field must be disabled (by being set to 0x00) for the corresponding DAC debug event to be enabled. Other settings of the *DVC1M* or *DVC2M* fields enable specific DVC operations, which also require the corresponding DVC byte enable (DBCR2[*DVC1BE*, *DVC2BE*]) fields to be set to a nonzero value. Data value compare events cannot occur on cache management instructions. See *Data Value Compare (DVC) Debug Event* on page 399 for more information about DVC events.

10.4.2.2 DAC Debug Event Processing

When enabled, the occurrence of a DAC debug event is recorded in the corresponding bit of the DBSR. If debug interrupts are not enabled (MSR[*DE*] = 0), the imprecise debug event (DBSR[*IDE*]) bit is also set. The resulting actions taken by the processor due to the DAC debug event depend on the specific debug configuration.

When operating in external debug mode (DBCR0[*EDM*] = 1) the setting of PCCR0[*DBA*] determines the resulting debug actions. If the debug action is a stop, the processor enters the stop state and ceases the processing of instructions. The program counter contains the address of the instruction that caused the DAC match to occur. If the PCCR0[*DBA*] decode does not stop the processor, instruction execution continues, and any additional debug actions are determined by the setting of DBCR0[*IDM*].

When operating in internal debug mode (DBCR0[*IDM*] = 1) with debug interrupts enabled (MSR[*DE*] = 1), a debug interrupt occurs with Critical Save/Restore Register 0 (CSRR0) set to the address of the instruction that caused the DAC debug event. When operating in internal debug mode with debug interrupts disabled (MSR[*DE*] = 0), the debug interrupt does not occur immediately. Instead, instruction execution continues, and a debug interrupt occurs if and when MSR[*DE*] is set to 1. This enables debug interrupts, assuming software has not cleared the DAC debug event status from the DBSR in the meantime. Upon such a “delayed” interrupt, the debug interrupt handler software can query the DBSR[*IDE*] field to determine that the debug interrupt has occurred imprecisely.

10.4.2.3 DAC Debug Events Applied to Instructions that Result in Multiple Storage Accesses

Certain misaligned load and store instructions are handled by making multiple, independent storage accesses. Similarly, load and store multiple and string instructions that access more than one register result in more than one storage access. *Load and Store Alignment* on page 159 provides a detailed description of the circumstances that lead to such multiple storage accesses being made as the result of the execution of a single instruction.

Whenever the execution of a given instruction results in multiple storage accesses, the data address of each access is independently considered for whether or not it will cause a DAC debug event.

10.4.2.4 DAC Debug Events Applied to Various Instruction Types

Various special cases apply to the cache management instructions, the store word and doubleword conditional indexed (**stwcx.**, **stdcx.**) instructions, and the load and store string indexed (**lswx**, **stswx**) instructions, with regards to DAC debug events. These special cases are as follows:

dcbz, **dcbzep**, **dcbi**

The **dcbz**, **dcbzep**, and **dcbi** instructions are considered “stores” with respect to both storage access control and DAC debug events. The **dcbz** and **dcbzep** instructions directly change the contents of a given storage location, whereas the **dcbi** instruction can indirectly change the contents of a given storage location by invalidating data that has been modified within the data cache, thereby “restoring” the value of the location to the “old” contents of memory. As “store” operations, they can cause DAC write debug events.

dcbst, **dcbstep**, **dcbf**, **dcbfep**

The **dcbst**, **dcbstep**, **dcbf**, and **dcbfep** instructions are considered “loads” with respect to storage access control because they do not change the contents of a given storage location. They can merely cause the data at that storage location to be moved from the data cache out to memory. However, in a debug environment, the fact that these instructions can lead to write operations on the external interface is typically the event of interest. Therefore, these instructions are considered “stores” with respect to DAC debug events and can cause DAC write debug events.

dcbt, **dcbtep**, **dcbtstep**

The touch instructions are considered “loads” with respect to both storage access control and DAC debug events. However, these instructions are treated as no-ops if they reference caching inhibited storage locations or if they cause data storage or data TLB miss exceptions. Consequently, if a touch instruction is being treated as a no-op for one of these reasons, then it does not cause a DAC read debug event. However, if a touch instruction is not being treated as a no-op for one of these reasons, it can cause a DAC read debug event.

dcba, **icbt**, **dcbtst**

The **dcba** and **icbt** instructions are treated as a no-op by the A2O Core, and thus will not cause a DAC debug event.

icbi, **icbiep**, **icbtls**, **icblc**, **dcbtls**, **dcbtstls**, **dcblic**

These instructions are considered a “load” with respect to both storage access control and DAC debug events, and thus can cause a DAC read debug event.

dci, **ici**

The **dci** and **ici** instructions do not generate an address, but rather they affect the entire data and instruction cache, respectively. Therefore, none of these instructions cause DAC debug events.

stwcx., **stdcx.**

If the execution of a **stwcx.** or **stdcx.** instruction would otherwise have caused a DAC write debug event, but the processor does not have the reservation from a **lwarx** instruction, then the DAC write debug event still occurs.

lswx, stswx

DAC debug events do not occur for **lswx** or **stswx** instructions with a length of 0 ($XER[SI] = 0$), because these instructions do not actually access storage.

10.4.3 Data Value Compare (DVC) Debug Event

DVC debug events occur when execution is attempted of a load or store instruction for which the data storage address and other parameters match the DAC conditions specified by DBCR0, DBCR2, and the DAC registers, and for which the data accessed matches the DVC conditions specified by DBCR2 and the DVC registers. DVC debug events are not supported for floating-point loads and stores. In other words, for a DVC debug event to occur, the conditions for a DAC debug event must first be met, and then the data must also match the DVC conditions. *Data Address Compare (DAC) Debug Event* on page 395 describes the DAC conditions. In addition to the DAC conditions, there are two DVC registers on the A2O Core, DVC1 and DVC2. The DVC registers can be used to specify two independent, 8-byte data values, which are selectively compared against the data being accessed by a given load, store, or cache management instruction.

When a DVC event occurs, the corresponding DBSR[*DAC1R*, *DAC1W*, *DAC2R*, *DAC2W*] bit is set. These same DBSR bits are shared by DAC debug events.

10.4.3.1 DVC Debug Event Fields

In addition to the DAC debug event fields described in *DAC Debug Event Fields* on page 395, and the DVC registers themselves, there are two fields in DBCR2 that are used to specify the DVC conditions, as follows:

DVC Byte Enable Field

DBCR2[*DVC1BE*, *DVC2BE*] are the individual DVC byte enable fields for the two DVC events. Each bit of a given DVC byte enable field corresponds to a byte position within an aligned doubleword of memory. For a given aligned doubleword of memory, the byte offsets (or “byte lanes”) within that doubleword are numbered 0 through 7, starting from the left-most (most significant) byte of the doubleword. Accordingly, bits 0:7 of a given DVC byte enable field correspond to bytes 0:7 of an aligned word of memory being accessed.

For an access to “match” the DVC conditions for a given byte, the access must be actually transferring data on that given byte position *and* the data must match the corresponding byte value within the DVC register.

For each storage access, the DVC comparison is made against the bytes that are being accessed within the doubleword of memory starting at the address specified by the corresponding DAC compare. This is true whether the DAC address is aligned or unaligned, providing that the storage access is an atomic operation. If an unaligned storage address forces byte accesses (see *Section 2.2.1 Storage Operands* on page 60), then the DVC comparison is made against byte offset 7 in the DBCR2 byte enable fields and their corresponding DVC registers.

DVC Mode Field

DBCR2[*DVC1M*, *DVC2M*] are the individual DVC mode fields for the two DVC events. Each one of these fields specifies the particular data value comparison mode for the corresponding DVC debug event. There are three comparison modes supported by the A2O Core:

- DAC mode only (DBCR2[DVC1M, DVC2M] = 0b00)

This mode enables DAC1 and DAC2 compare events providing the respective DBCR0 and DBCR2 DAC settings result in a match condition. In this mode, the corresponding DBCR2[DVC1BE]/DVC1 and DBCR2[DVC2BE]/DVC2 bits are not used for determining if the DAC compare event will occur.

- AND comparison mode (DBCR2[DVC1M, DVC2M] = 0b01)

In this mode, all data byte lanes enabled by a DVC byte enable field must be accessed and must match the corresponding byte data value in the corresponding DVC1 or DVC2 register.

- OR comparison mode (DBCR2[DVC1M, DVC2M] = 0b10)

In this mode, at least one data byte lane that is enabled by a DVC byte enable field must be accessed and must match the corresponding byte data value in the DVC1 or DVC2 register.

- AND-OR comparison mode (DBCR2[DVC1M, DVC2M] = 0b11)

In this mode, at least one of the halfwords that are enabled by the DVC byte enable field must match the corresponding byte data value in the DVC1 or DVC2 register.

10.4.3.2 DVC Debug Event Processing

When enabled, the occurrence of a DVC debug event is recorded in the corresponding DAC1R/W or DAC2R/W bit of the DBSR. If debug interrupts are not enabled (MSR[DE] = 0), the imprecise debug event (DBSR[IDE]) bit is also set. The resulting actions taken by the processor due to the DVC debug event depend on the specific debug configuration.

When operating in external debug mode (DBCR0[EDM] = 1), the setting of PCCR0[DBA] determines the resulting debug actions. If the debug action is a stop, the processor enters the stop state and ceases the processing of instructions. The program counter contains the address of the instruction that caused the DVC debug event, or in the case of a load miss to some other subsequent instruction. If the PCCR0[DBA] decode does not stop the processor, instruction execution continues, and any additional debug actions are determined by the setting of DBCR0[IDM].

When operating in internal debug mode (DBCR0[IDM] = 1) with debug interrupts enabled (MSR[DE] = 1), a debug interrupt occurs. CSRR0 contains the address of the instruction that caused the DVC debug event, or in the case of a load miss to some other subsequent instruction. When operating in internal debug mode with debug interrupts disabled (MSR[DE] = 0), the debug interrupt does not occur immediately. Instead, instruction execution continues, and a debug interrupt occurs if and when MSR[DE] is set to 1. This enables debug interrupts, assuming software has not cleared the DVC debug event status from the DBSR in the meantime. Upon such a “delayed” interrupt, the debug interrupt handler software can query the DBSR[IDE] field to determine that the debug interrupt has occurred imprecisely.

10.4.3.3 DVC Debug Events Applied to Instructions that Result in Multiple Storage Accesses

Certain misaligned load and store instructions are handled by making multiple, independent storage accesses. Similarly, load and store multiple and string instructions that access more than one register result in more than one storage access. *Load and Store Alignment* on page 159 provides a detailed description of the circumstances that lead to such multiple storage accesses being made as the result of the execution of a single instruction.

Whenever the execution of a given instruction results in multiple storage accesses, the address and data of each access is independently considered for whether or not it will cause a DVC debug event.

10.4.3.4 DVC Debug Events Applied to Various Instruction Types

Various special cases apply to the cache management instructions, the store word and doubleword conditional indexed (**stwcx.**, **stdcx.**) instruction, and the load and store string indexed (**lswx**, **stswx**) instructions, with regards to DVC debug events. These special cases are as follows:

stwcx., stdcx.

If the execution of a **stwcx.** or **stdcx.** instruction would otherwise have caused a DVC write debug event, but the processor does not have the reservation from an **lwarx** instruction, then the DVC write debug event still occurs.

lswx, stswx

DVC debug events do not occur for **lswx** or **stswx** instructions with a length of 0 ($XER[SI] = 0$), because these instructions do not actually access storage.

10.4.3.5 DVC Debug Events Applied to Floating-Point Loads and Stores

DVC debug events are not supported for floating-point loads and stores.

10.4.4 Instruction Complete (ICMP) Debug Event

ICMP debug events occur when ICMP debug events are enabled ($DBCRO[ICMP] = 1$), debug interrupts are enabled ($MSR[DE] = 1$), and the A2O Core completes the execution of any instruction.

When operating in external ($DBCRO[EDM] = 1$) debug mode, the occurrence of an ICMP debug event is recorded in $DBSR[ICMP]$. In external debug mode, the setting of $PCCR0[DBA]$ determines the resulting debug actions. If the debug action is a stop, the processor enters the stop state and ceases the processing of instructions. The program counter contains the address of the instruction that would have executed next, had the ICMP debug event not occurred. Note that if the instruction whose completion caused the ICMP debug event was a branch instruction (and the branch conditions were satisfied), then upon entering the stop state the program counter contains the target of the branch, and not the address of the instruction that is sequentially after the branch. Similarly, if the ICMP debug event is caused by the execution of a return (**rfi**, **rfci**, or **rfmci**) instruction, then upon entering the stop state the program counter contains the address being *returned to*, and not the address of the instruction that is sequentially after the return instruction. If the $PCCR0[DBA]$ decode does not stop the processor, instruction execution continues, and any additional debug actions are determined by the setting of $DBCRO[IDM]$.

When operating in internal debug mode ($DBCRO[IDM] = 1$) with debug interrupts enabled ($MSR[DE] = 1$), the occurrence of an ICMP debug event is recorded in $DBSR[ICMP]$ and a debug interrupt occurs with $CSRR0$ set to the address of the instruction that would have executed next, had the ICMP debug event not occurred. Note that there is a special case of $MSR[DE] = 1$ at the time of the execution of the instruction causing the ICMP debug event, but that instruction itself sets $MSR[DE]$ to 0. This special case is described in more detail in *Debug Interrupt* on page 331, in the subsection on the setting of $CSRR0$.

When debug interrupts are disabled ($MSR[DE] = 0$), ICMP debug events cannot occur. Because the code at the beginning of the critical class interrupt handlers (including the debug interrupt itself) must execute at least temporarily with $MSR[DE] = 0$, there would be no way to avoid causing additional ICMP debug events and setting $DBSR[IDE]$, if ICMP debug events were allowed to occur under these conditions.

10.4.5 Branch Taken (BRT) Debug Event

BRT debug events occur when BRT debug events are enabled ($\text{DBCR0}[\text{BRT}] = 1$), debug interrupts are enabled ($\text{MSR}[\text{DE}] = 1$), and execution is attempted of a branch instruction for which the branch conditions are satisfied, such that the instruction stream is redirected to the target address of the branch.

When operating in external ($\text{DBCR0}[\text{EDM}] = 1$) debug mode, the occurrence of a BRT debug event is recorded in $\text{DBSR}[\text{BRT}]$. In external debug mode, the setting of $\text{PCCR0}[\text{DBA}]$ determines the resulting debug actions. If the debug action is a stop, the processor enters the stop state and ceases the processing of instructions. The program counter contains the address of the branch instruction that caused the BRT debug event. If the $\text{PCCR0}[\text{DBA}]$ decode does not stop the processor, instruction execution continues, and any additional debug actions are determined by the setting of $\text{DBCR0}[\text{IDM}]$ as described below.

When operating in internal debug mode ($\text{DBCR0}[\text{IDM}] = 1$) with debug interrupts enabled ($\text{MSR}[\text{DE}] = 1$), the occurrence of a BRT debug event is recorded in $\text{DBSR}[\text{BRT}]$ and causes the instruction execution to be suppressed. A debug interrupt occurs with CSRR0 set to the address of the branch instruction that caused the BRT debug event.

When debug interrupts are disabled ($\text{MSR}[\text{DE}] = 0$), BRT debug events cannot occur. Because taken branches are a very common operation and thus likely to be frequently executed within the critical class interrupt handlers (which typically have $\text{MSR}[\text{DE}]$ set to 0), allowing BRT debug events under these conditions would lead to an undesirable number of delayed (and hence imprecise) debug interrupts.

10.4.6 Trap (TRAP) Debug Event

TRAP debug events occur when TRAP debug events are enabled ($\text{DBCR0}[\text{TRAP}] = 1$) and execution is attempted of a trap (**tw**, **twi**, **td**, **tdi**) instruction for which the trap condition is satisfied.

When enabled, the occurrence of a TRAP debug event is recorded in $\text{DBSR}[\text{TRAP}]$. If debug interrupts are not enabled ($\text{MSR}[\text{DE}] = 0$), the imprecise debug event ($\text{DBSR}[\text{IDE}]$) bit is also set. The resulting actions taken by the processor due to the TRAP debug event depend on the specific debug configuration.

When operating in external debug mode ($\text{DBCR0}[\text{EDM}] = 1$), the setting of $\text{PCCR0}[\text{DBA}]$ determines the resulting debug actions. If the debug action is a stop, the processor enters the stop state and ceases the processing of instructions. The program counter contains the address of the trap instruction that caused the TRAP debug event. If the $\text{PCCR0}[\text{DBA}]$ decode does not stop the processor, instruction execution continues, and any additional debug actions are determined by the setting of $\text{DBCR0}[\text{IDM}]$.

When operating in internal debug mode ($\text{DBCR0}[\text{IDM}] = 1$) with debug interrupts enabled ($\text{MSR}[\text{DE}] = 1$), a debug interrupt occurs with CSRR0 set to the address of the trap instruction that caused the TRAP debug event. When operating in internal debug mode with debug interrupts disabled ($\text{MSR}[\text{DE}] = 0$), the debug interrupt does not occur immediately. Instruction execution is suppressed and a trap exception type of program interrupt occurs instead. A debug interrupt also occurs later, if and when $\text{MSR}[\text{DE}]$ is set to 1. This enables debug interrupts, assuming software has not cleared the TRAP debug event status from the DBSR in the meantime. Upon such a “delayed” interrupt, the debug interrupt handler software can query the $\text{DBSR}[\text{IDE}]$ field to determine that the debug interrupt has occurred imprecisely.

10.4.7 Return (RET) Debug Event

RET debug events occur when RET debug events are enabled ($\text{DBCR0}[\text{RET}] = 1$) and execution is attempted of a noncritical class return (**rfi**) instruction.

When enabled, the occurrence of a RET debug event is recorded in DBSR[RET]. If debug interrupts are not enabled ($MSR[DE] = 0$), the imprecise debug event (DBSR[IDE]) bit is also set. The resulting actions taken by the processor due to the RET debug event depend on the specific debug configuration.

When operating in external debug mode ($DBCR0[EDM] = 1$), the setting of PCCR0[DBA] determines the resulting debug actions. If the debug action is a stop, the processor enters the stop state and ceases the processing of instructions. The program counter contains the address of the return instruction that caused the RET debug event. If the PCCR0[DBA] decode does not stop the processor, instruction execution continues, and any additional debug actions are determined by the setting of DBCR0[IDM].

When operating in internal debug mode ($DBCR0[IDM] = 1$) with debug interrupts enabled ($MSR[DE] = 1$), a debug interrupt occurs with CSRR0 set to the address of the return instruction that caused the RET debug event. When operating in internal debug mode with debug interrupts disabled ($MSR[DE] = 0$), the debug interrupt does not occur immediately. Instead, instruction execution continues, and a debug interrupt occurs if and when $MSR[DE]$ is set to 1. This enables debug interrupts, assuming software has not cleared the RET debug event status from the DBSR in the meantime. Upon such a “delayed” interrupt, the debug interrupt handler software can query the DBSR[IDE] field to determine that the debug interrupt has occurred imprecisely.

10.4.8 Interrupt (IRPT) Debug Event

IRPT debug events occur when IRPT debug events are enabled ($DBCR0[IRPT] = 1$) and a base class interrupt occurs. Critical or machine check class interrupts cannot cause IRPT debug events. Otherwise, a debug interrupt caused by an IRPT debug event would always be imprecise by necessity, because an interrupt that caused an IRPT debug event would also cause $MSR[DE]$ to be set to 0.

When enabled, the occurrence of an IRPT debug event is recorded in DBSR[IRPT]. If debug interrupts are not enabled ($MSR[DE] = 0$), the imprecise debug event (DBSR[IDE]) bit is also set. The resulting actions taken by the processor due to the IRPT debug event depend on the specific debug configuration.

When operating in external debug mode ($DBCR0[EDM] = 1$), the setting of PCCR0[DBA] determines the resulting debug actions. If the debug action is a stop, the processor enters the stop state and ceases the processing of instructions. The program counter contains the address of the instruction that would have executed next, had the IRPT debug event not occurred. Because the IRPT debug event is caused by the occurrence of an interrupt, by definition this address is that of the first instruction of the interrupt handler for the interrupt type that caused the IRPT debug event. If the PCCR0[DBA] decode does not stop the processor, instruction execution continues, and any additional debug actions are determined by the setting of DBCR0[IDM].

When operating in internal debug mode ($DBCR0[IDM] = 1$) with debug interrupts enabled ($MSR[DE] = 1$), a debug interrupt occurs with CSRR0 set to the address of the instruction that would have executed next, had the IRPT debug event not occurred. Because the IRPT debug event is caused by the occurrence of some other interrupt, by definition this address is that of the first instruction of the interrupt handler for the interrupt type that caused the IRPT debug event. When operating in internal debug mode with debug interrupts disabled ($MSR[DE] = 0$), the debug interrupt does not occur immediately. Instead, instruction execution continues, and a debug interrupt occurs if and when $MSR[DE]$ is set to 1. This enables debug interrupts, assuming software has not cleared the IRPT debug event status from the DBSR in the meantime. Upon such a “delayed” interrupt, the debug interrupt handler software can query the DBSR[IDE] field to determine that the debug interrupt has occurred imprecisely.

10.4.9 Unconditional Debug Event (UDE)

UDE debug events occur when the core's *an_ac_uncond_dbg_event* input is activated. The UDE debug event is the only event that does not have a corresponding enable field in either DBCR0 or DBCR3.

The occurrence of a UDE debug event is recorded in DBSR[UDE]. If debug interrupts are not enabled (MSR[DE] = 0), the imprecise debug event (DBSR[IDE]) bit is also set. The resulting actions taken by the processor due to the UDE debug event depend on the specific debug configuration.

When operating in external debug mode (DBCR0[EDM] = 1), the setting of PCCR0[DBA] determines the resulting debug actions. If the debug action is a stop, the processor enters the stop state and ceases the processing of instructions. The program counter contains the address of the instruction that would have executed next, had the UDE debug event not occurred. If the PCCR0[DBA] decode does not stop the processor, instruction execution continues, and any additional debug actions are determined by the setting of DBCR0[IDM].

When operating in internal debug mode (DBCR0[IDM] = 1) with debug interrupts enabled (MSR[DE] = 1), a Debug interrupt occurs with CSRR0 set to the address of the instruction that would have executed next, had the UDE debug event not occurred. When operating in internal debug mode with debug interrupts disabled (MSR[DE] = 0), the debug interrupt does not occur immediately. Instead, instruction execution continues, and a debug interrupt occurs if and when MSR[DE] is set to 1. This enables debug interrupts, assuming software has not cleared the UDE debug event status from the DBSR in the meantime. Upon such a "delayed" interrupt, the debug interrupt handler software can query the DBSR[IDE] field to determine that the debug interrupt has occurred imprecisely.

10.4.10 Instruction Value Compare (IVC) Debug Event

The instruction value compare function provides a method of comparing an instruction against a set of mask registers and performing selected actions when a match condition occurs. The masks are implemented through the Instruction Match (IMR) and Instruction Match Mask (IMMR) registers. The IMR contains the instruction compare data, and the IMMR is used to hold a 32-bit mask. A match occurs when the instruction data bitwise ANDed against the IMMR equals the IMR also bitwise ANDed against the IMMR. An instruction value compare debug event is enabled by setting DBCR3[IVC].

When enabled, the occurrence of an IVC debug event is recorded in DBSR[IVC]. If debug interrupts are not enabled (MSR[DE] = 0), the imprecise debug event (DBSR[IDE]) bit is also set. The resulting actions taken by the processor due to the IVC debug event depend on the specific debug configuration.

When operating in external debug mode (DBCR0[EDM] = 1), the setting of PCCR0[DBA] determines the resulting debug actions. If the debug action is a stop, the processor enters the stop state and ceases the processing of instructions. The program counter contains the address of the instruction that caused the IVC match to occur. If the PCCR0[DBA] decode does not stop the processor, instruction execution continues, and any additional debug actions are determined by the setting of DBCR0[IDM].

When operating in internal debug mode (DBCR0[IDM] = 1) with debug interrupts enabled (MSR[DE] = 1), a debug interrupt occurs with CSRR0 set to the address of the instruction that caused the IVC debug event. When operating in internal debug mode with debug interrupts disabled (MSR[DE] = 0), the debug interrupt does not occur immediately. Instead, instruction execution continues, and a debug interrupt occurs if and when MSR[DE] is set to 1. This enables debug interrupts, assuming software has not cleared the IVC debug event status from the DBSR in the meantime. Upon such a "delayed" interrupt, the debug interrupt handler software can query the DBSR[IDE] field to determine that the debug interrupt has occurred imprecisely.

10.4.11 Debug Event Summary

Table 10-3 summarizes each of the debug event types, and the effect of the debug modes and MSR[DE] on their occurrence.

Table 3. Debug Event Summary

MSR [DE]	External Debug Mode	Internal Debug Mode	Debug Events										
			IAC	DAC	DVC	ICMP	BRT	TRAP	RET	IRPT	UDE	IVC	
1	Note 1	Note 1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Note 2	Note 2	Yes	Yes
0	Note 1	Note 1	Yes	Yes	Yes	No	No	Yes	Note 2	Note 2	Yes	Yes	

Notes:

1. The occurrence of debug events (to activate bits in the DBSR) is not dependent on the setting of DBCR0[EDM] or DBCR0[IDM]. Actions taken in response to the debug event, however, depend on the values of MSR[DE] along with the state of internal and external debug modes. When both DBCR0[EDM] and DBCR0[IDM] are active, the external debug mode actions take precedence (that is, the processor stops before the debug interrupt occurs).
2. RET and IRPT debug events can only occur for noncritical class interrupts.

10.5 Debug Reset

Software can initiate an immediate reset operation by setting DBCR0[RST] to a nonzero value. The A2 core decodes this field and activates one of three external reset request signals. The exact meaning associated with these reset requests, and any actions taken in response to them, are chip or system dependent. Core reset requests are described in *Section 4 Initialization* on page 149.

10.6 Debug Timer Freeze

To maintain the semblance of “real time” operation while a system is being debugged, DBCR0[FT] can be set to 1, which causes all of the timers for that thread to stop incrementing or decrementing for as long as a debug event bit is set in the DBSR, or until DBCR0[FT] is set to 0. See *Timer Facilities* on page 371 for more information about the operation of the A2O Core timers.

10.7 Debug Registers

Various Special Purpose Registers (SPRs) are used to enable the debug modes, to configure and record debug events, and to communicate with debug tool hardware and software. These debug registers can be accessed either through software running on the processor or by the SCOM interface of the A2O Core through the Ram instruction stuffing facilities.

For the debug facilities on the A2 core, all control and status registers (DBCR0 - DBCR3, DBSR) and the instruction value registers (IMMR, IMR) are replicated per thread. The data registers used to hold other compare values (IAC1 - IAC4, DAC1 - DAC4, DVC1 - DVC2) are implemented per core, and therefore need to be shared when debug operations are run simultaneously by multiple threads.

Programming Note: It is the responsibility of software to synchronize the context of any changes to the debug facility registers. Specifically, when changing the contents of any of the debug facility registers, software must execute an **isync** instruction both *before* and *after* the changes to these registers, to ensure

that all preceding instructions use the *old* values of the registers, and that all succeeding instructions use the *new* values. In addition, when changing any of the debug facility register fields related to the DAC debug events, DVC debug events, or both, software must execute an **msync** instruction *before* making the changes, to ensure that all storage accesses complete using the *old* context of these register fields.

10.7.1 Debug Control Register 0 (DBCR0)

DBCR0 is an SPR that is used to enable debug modes and events, reset the processor, and control timer operation when debugging. DBCR0 can be written from a GPR using **mtspr** and can be read into a GPR using **mfspr**.

Register Short Name:	DBCR0	Read Access:	Hypv
Decimal SPR Number:	308	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	dcfg

Bit(s):	Field Name:	Init	Description
32	EDM ^{RO}	0b0	<u>External Debug Mode</u> ^{RO} Reports the state of external debug mode 0 external debug mode is disabled 1 external debug mode is enabled. External debug mode is set, and the corresponding debug action selected, from the decoded value of PCCR0[DBA] bits.
33	IDM	0b0	<u>Internal Debug Mode</u> Enable internal debug mode. If MSR[DE]=1, then the occurrence of a debug event or the recording of an earlier debug event in the Debug Status Register when MSR[DE]=0 or DBCR0[IDM]=0 will cause a Debug interrupt
34:35	RST	0b00	<u>Reset</u> 00 No Action 01 Reset1 10 Reset2 11 Reset3
36	ICMP	0b0	<u>Instruction Completion Debug Event</u> 0 ICMP debug events are disabled 1 ICMP debug events are enabled when MSR[DE]=1
37	BRT	0b0	<u>Branch Taken Debug Event</u> 0 BRT debug events are disabled 1 BRT debug events are enabled when MSR[DE]=1
38	IRPT	0b0	<u>Interrupt Taken Debug Event Enable</u> 0 IRPT debug events are disabled 1 IRPT debug events are enabled
39	TRAP	0b0	<u>Trap Debug Event Enable</u> 0 TRAP debug events cannot occur 1 TRAP debug events can occur
40	IAC1	0b0	<u>Instruction Address Compare 1 Debug Event Enable</u> 0 IAC1 debug events cannot occur 1 IAC1 debug events can occur

Bit(s):	Field Name:	Init	Description
41	IAC2	0b0	<u>Instruction Address Compare 2 Debug Event Enable</u> 0 IAC2 debug events cannot occur 1 IAC2 debug events can occur
42	IAC3	0b0	<u>Instruction Address Compare 3 Debug Event Enable</u> 0 IAC3 debug events cannot occur 1 IAC3 debug events can occur
43	IAC4	0b0	<u>Instruction Address Compare 4 Debug Event Enable</u> 0 IAC4 debug events cannot occur 1 IAC4 debug events can occur
44:45	DAC1	0b00	<u>Data Address Compare 1 Debug Event Enable</u> 00 Disabled: DAC1 debug events cannot occur 01 Store Only: DAC1 debug events can occur only if a store-type data storage access 10 Load Only: DAC1 debug events can occur only if a load-type data storage access 11 Any: DAC1 debug events can occur on any data storage access
46:47	DAC2	0b00	<u>Data Address Compare 2 Debug Event Enable</u> 00 Disabled: DAC2 debug events cannot occur 01 Store Only: DAC2 debug events can occur only if a store-type data storage access 10 Load Only: DAC2 debug events can occur only if a load-type data storage access 11 Any: DAC2 debug events can occur on any data storage access
48	RET	0b0	<u>Return Debug Event Enable</u> 0 RET debug events cannot occur 1 RET debug events can occur
49:58	///	0x0	<u>Reserved</u>
59:60	DAC3	0b00	<u>Data Address Compare 3 Debug Event Enable</u> 00 Disabled: DAC3 debug events cannot occur 01 Store Only: DAC3 debug events can occur only if a store-type data storage access 10 Load Only: DAC3 debug events can occur only if a load-type data storage access 11 Any: DAC3 debug events can occur on any data storage access
61:62	DAC4	0b00	<u>Data Address Compare 4 Debug Event Enable</u> 00 Disabled: DAC4 debug events cannot occur 01 Store Only: DAC4 debug events can occur only if a store-type data storage access 10 Load Only: DAC4 debug events can occur only if a load-type data storage access 11 Any: DAC4 debug events can occur on any data storage access
63	FT	0b0	<u>Freeze Timers on Debug Event</u> 0 Enable clocking of timers 1 Disable clocking of timers if any DBSR bit is set (except MRR)

10.7.2 Debug Control Register 1 (DBCR1)

DBCR1 is an SPR that is used to configure IAC debug events. DBCR1 can be written from a GPR using **mtsp** and can be read into a GPR using **mf spr**.

Register Short Name:	DBCR1	Read Access:	Hypv
Decimal SPR Number:	309	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	IAC1US	0b00	<u>Instruction Address Compare 1 User/Supervisor Mode</u> 00 Enabled: IAC1 debug events can occur 01 Reserved 10 Enabled PR0: IAC1 debug events can occur only if MSR[PR]=0 11 Enabled PR1: IAC1 debug events can occur only if MSR[PR]=1
34:35	IAC1ER	0b00	<u>Instruction Address Compare 1 Effective/Real Mode</u> 00 Effective: IAC1 debug events are based on effective addresses 01 Not Implemented: 10 Effective IS0: IAC1 debug events are based on effective addresses and if MSR[IS]=0 11 Effective IS1: IAC1 debug events are based on effective addresses and if MSR[IS]=1
36:37	IAC2US	0b00	<u>Instruction Address Compare 2 User/Supervisor Mode</u> 00 Enabled: IAC2 debug events can occur 01 Reserved 10 Enabled PR0: IAC2 debug events can occur only if MSR[PR]=0 11 Enabled PR1: IAC2 debug events can occur only if MSR[PR]=1
38:39	IAC2ER	0b00	<u>Instruction Address Compare 2 Effective/Real Mode</u> 00 Effective: IAC2 debug events are based on effective addresses 01 Not Implemented 10 Effective IS0: IAC2 debug events are based on effective addresses and if MSR[IS]=0 11 Effective IS1: IAC2 debug events are based on effective addresses and if MSR[IS]=1
40	///	0b0	<u>Reserved</u>
41	IAC12M	0b0	<u>Instruction Address Compare 1/2 Mode</u> 0 Exact address compare 1 Address bit match
42:47	///	0x0	<u>Reserved</u>
48:49	IAC3US	0b00	<u>Instruction Address Compare 3 User/Supervisor Mode</u> 00 Enabled: IAC3 debug events can occur 01 Reserved 10 Enabled PR0: IAC3 debug events can occur only if MSR[PR]=0 11 Enabled PR1: IAC3 debug events can occur only if MSR[PR]=1
50:51	IAC3ER	0b00	<u>Instruction Address Compare 3 Effective/Real Mode</u> 00 Effective: IAC3 debug events are based on effective addresses 01 Not Implemented 10 Effective IS0: IAC3 debug events are based on effective addresses and if MSR[IS]=0 11 Effective IS1: IAC3 debug events are based on effective addresses and if MSR[IS]=1

Bit(s):	Field Name:	Init	Description
52:53	IAC4US	0b00	<u>Instruction Address Compare 4 User/Supervisor Mode</u> 00 Enabled: IAC4 debug events can occur 01 Reserved 10 Enabled PR0: IAC4 debug events can occur only if MSR[PR]=0 11 Enabled PR1: IAC4 debug events can occur only if MSR[PR]=1
54:55	IAC4ER	0b00	<u>Instruction Address Compare 4 Effective/Real Mode</u> 00 Effective: IAC4 debug events are based on effective addresses 01 Not Implemented 10 Effective IS0: IAC4 debug events are based on effective addresses and if MSR[IS]=0 11 Effective IS1: IAC4 debug events are based on effective addresses and if MSR[IS]=1
56	///	0b0	<u>Reserved</u>
57	IAC34M	0b0	<u>Instruction Address Compare 3/4 Mode</u> 0 Exact address compare 1 Address bit match
58:63	///	0x0	<u>Reserved</u>

10.7.3 Debug Control Register 2 (DBCR2)

DBCR2 is an SPR that is used to configure DAC and DVC debug events. DBCR2 can be written from a GPR using **mtspr** and can be read into a GPR using **mfspr**.

Register Short Name:	DBCR2	Read Access:	Hypv
Decimal SPR Number:	310	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	DAC1US	0b00	<u>Data Address Compare 1 User/Supervisor Mode</u> 00 Enabled: DAC1 debug events can occur 01 Reserved: 10 Enabled PR0: DAC1 debug events can occur only if MSR[PR]=0 11 Enabled PR1: DAC1 debug events can occur only if MSR[PR]=1
34:35	DAC1ER	0b00	<u>Data Address Compare 1 Effective/Real Mode</u> 00 Effective: DAC1 debug events are based on effective addresses 01 Not Implemented 10 Effective DS0: DAC1 debug events are based on effective and if MSR[DS]=0 11 Effective DS1: DAC1 debug events are based on effective and if MSR[DS]=1
36:37	DAC2US	0b00	<u>Data Address Compare 2 User/Supervisor Mode</u> 00 Enabled: DAC2 debug events can occur 01 Reserved 10 Enabled PR0: DAC2 debug events can occur only if MSR[PR]=0 11 Enabled PR1: DAC2 debug events can occur only if MSR[PR]=1

Bit(s):	Field Name:	Init	Description
38:39	DAC2ER	0b00	<u>Data Address Compare 2 Effective/Real Mode</u> 00 Effective: DAC2 debug events are based on effective addresses 01 Not Implemented 10 Effective DS0: DAC2 debug events are based on effective and if MSR[DS]=0 11 Effective DS1: DAC2 debug events are based on effective and if MSR[DS]=1
40	///	0b0	<u>Reserved</u>
41	DAC12M	0b0	<u>Data Address Compare 1/2 Mode</u> 0 Exact: Exact address compare 1 Bit Match: Address bit match
42:43	///	0b00	<u>Reserved</u>
44:45	DVC1M	0b00	<u>Data Value Compare 1 Mode</u> 00 DVC Disabled: DAC1 debug events can occur 01 DVC All: DAC1 debug events can occur only when all bytes specified by DVC1BE in the data value of the data storage access match their corresponding bytes in DVC1 10 DVC Any: DAC1 debug events can occur only when at least one of the bytes specified by DVC1BE in the data value of the data storage access matches its corresponding byte in DVC1 11 DVC HW: DAC1 debug events can occur only when all bytes specified in DVC1BE within at least one of the halfwords of the data value of the data storage access matches their corresponding bytes in DVC1
46:47	DVC2M	0b00	<u>Data Value Compare 2 Mode</u> 00 DVC Disabled: DAC2 debug events can occur 01 DVC All: DAC2 debug events can occur only when all bytes specified in by DVC2BE in the data value of the data storage access match their corresponding bytes in DVC2 10 DVC Any: DAC2 debug events can occur only when at least one of the bytes specified by DVC2BE in the data value of the data storage access matches its corresponding byte in DVC2 11 DVC HW: DAC2 debug events can occur only when all bytes specified in DVC2BE within at least one of the halfwords of the data value of the data storage access matches their corresponding bytes in DVC2
48:55	DVC1BE	0x0	<u>Data Value Compare 1 Byte Enables</u> Specifies which bytes in the aligned data value being read or written by the storage access are compared to the corresponding bytes in DVC1
56:63	DVC2BE	0x0	<u>Data Value Compare 2 Byte Enables</u> Specifies which bytes in the aligned data value being read or written by the storage access are compared to the corresponding bytes in DVC2

10.7.4 Debug Control Register 3 (DBCR3)

DBCR3 is an SPR that is used to configure DAC and DVC debug events and to enable IVC debug events. DBCR3 can be written from a GPR using **mtspr** and can be read into a GPR using **mfspir**.

Register Short Name:	DBCR3	Read Access:	Hypv
Decimal SPR Number:	848	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	DAC3US	0b00	<u>Data Address Compare 3 User/Supervisor Mode</u> 00 Enabled: DAC3 debug events can occur 01 Reserved 10 Enabled PR0: DAC3 debug events can occur only if MSR[PR]=0 11 Enabled PR1: DAC3 debug events can occur only if MSR[PR]=1
34:35	DAC3ER	0b00	<u>Data Address Compare 3 Effective/Real Mode</u> 00 Effective: DAC3 debug events are based on effective addresses 01 Not Implemented 10 Effective DS0: DAC3 debug events are based on effective and if MSR[DS]=0 11 Effective DS1: DAC3 debug events are based on effective and if MSR[DS]=1
36:37	DAC4US	0b00	<u>Data Address Compare 4 User/Supervisor Mode</u> 00 Enabled: DAC4 debug events can occur 01 Reserved 10 Enabled PR0: DAC4 debug events can occur only if MSR[PR]=0 11 Enabled PR1: DAC4 debug events can occur only if MSR[PR]=1
38:39	DAC4ER	0b00	<u>Data Address Compare 4 Effective/Real Mode</u> 00 Effective: DAC4 debug events are based on effective addresses 01 Not Implemented 10 Effective DS0: DAC4 debug events are based on effective and if MSR[DS]=0 11 Effective DS1: DAC4 debug events are based on effective and if MSR[DS]=1
40	///	0b0	<u>Reserved</u>
41	DAC34M	0b0	<u>Data Address Compare 3/4 Mode</u> 0 Exact address compare 1 Address bit match
42:62	///	0x0	<u>Reserved</u>
63	IVC	0b0	<u>Instruction Value Compare Event</u> 0 Instruction value compare events disabled 1 Instruction value compare events enabled

10.7.5 Debug Status Register (DBSR)

The DBSR contains the status of debug events and information about the type of the most recent reset. The status bits are set by the occurrence of debug events, while the reset type information is updated upon the occurrence of any of the three reset types.

The DBSR is read into a GPR using **mf spr**. Clearing the DBSR is performed using **mt spr** by placing a 1 in the GPR source register in all bit positions that are to be cleared in the DBSR, and a 0 in all other bit positions. The data written from the GPR to the DBSR is not direct data, but a mask. A 1 clears the bit and a 0 leaves the corresponding DBSR bit unchanged.

Register Short Name:	DBSR	Read Access:	Hypv
Decimal SPR Number:	304	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	WC
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	IDE	0b0	<u>Imprecise Debug Event</u> Set to 1 if MSRDE=0 and a debug event causes its respective Debug Status Register bit to be set to 1
33	UDE	0b0	<u>Unconditional Debug Event</u> Set to 1 if an Unconditional debug event occurred
34:35	MRR	0b00	<u>Most Recent Reset</u> Set to one of three values when a reset occurs: 00 No Action 01 Reset1 10 Reset2 11 Reset3
36	ICMP	0b0	<u>Instruction Complete Debug Event</u> Set to 1 if an Instruction Completion debug event occurred and DBCR0[ICMP]=1
37	BRT	0b0	<u>Branch Taken Debug Event</u> Set to 1 if a Branch Taken debug event occurred and DBCR0[BRT]=1
38	IRPT	0b0	<u>Interrupt Taken Debug Event</u> Set to 1 if an Interrupt Taken debug event occurred and DBCR0[IRPT]=1
39	TRAP	0b0	<u>Trap Instruction Debug Event</u> Set to 1 if a Trap Instruction debug event occurred and DBCR0[TRAP]=1
40	IAC1	0b0	<u>Instruction Address Compare 1 Debug Event</u> Set to 1 if an IAC1 debug event occurred and DBCR0[IAC1]=1
41	IAC2	0b0	<u>Instruction Address Compare 2 Debug Event</u> Set to 1 if an IAC2 debug event occurred and DBCR0[IAC2]=1
42	IAC3	0b0	<u>Instruction Address Compare 3 Debug Event</u> Set to 1 if an IAC3 debug event occurred and DBCR0[IAC3]=1
43	IAC4	0b0	<u>Instruction Address Compare 4 Debug Event</u> Set to 1 if an IAC4 debug event occurred and DBCR0[IAC4]=1
44	DAC1R	0b0	<u>Data Address Compare 1 Read Debug Event</u> Set to 1 if a read-type DAC1 debug event occurred and DBCR0[DAC1]=0b10 or DBCR0[DAC1]=0b11

Bit(s):	Field Name:	Init	Description
45	DAC1W	0b0	<u>Data Address Compare 1 Write Debug Event</u> Set to 1 if a write-type DAC1 debug event occurred and DBCR0[DAC1]=0b01 or DBCR0[DAC1]=0b11
46	DAC2R	0b0	<u>Data Address Compare 2 Read Debug Event</u> Set to 1 if a read-type DAC2 debug event occurred and DBCR0[DAC2]=0b10 or DBCR0[DAC2]=0b11
47	DAC2W	0b0	<u>Data Address Compare 2 Write Debug Event</u> Set to 1 if a write-type DAC2 debug event occurred and DBCR0[DAC2]=0b01 or DBCR0[DAC2]=0b11
48	RET	0b0	<u>Return Debug Event</u> Set to 1 if a Return debug event occurred and DBCR0[RET]=1
49:58	///	0x0	<u>Reserved</u>
59	DAC3R	0b0	<u>Data Address Compare 3 Read Debug Event</u> Set to 1 if a read-type DAC3 debug event occurred and DBCR0[DAC3]=0b10 or DBCR0[DAC3]=0b11
60	DAC3W	0b0	<u>Data Address Compare 3 Write Debug Event</u> Set to 1 if a write-type DAC3 debug event occurred and DBCR0[DAC3]=0b01 or DBCR0[DAC3]=0b11
61	DAC4R	0b0	<u>Data Address Compare 4 Read Debug Event</u> Set to 1 if a read-type DAC4 debug event occurred and DBCR0[DAC4]=0b10 or DBCR0[DAC4]=0b11
62	DAC4W	0b0	<u>Data Address Compare 4 Write Debug Event</u> Set to 1 if a write-type DAC4 debug event occurred and DBCR0[DAC4]=0b01 or DBCR0[DAC4]=0b11
63	IVC	0b0	<u>Instruction Value Compare Event</u> Set to 1 if an IVC debug event occurred with DBCR3[IVC]=1

10.7.6 Debug Status Register Write Register (DBSRWR)

The DBSRWR is a write only register with the same format as the DBSR. It can be used to set the corresponding DBSR bit when running in the hypervisor state (MSR[PR,GS] = 00) using an **mtspr** instruction.

Register Short Name:	DBSRWR	Read Access:	None
Decimal SPR Number:	306	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	IDE	0b0	<u>Imprecise Debug Event</u> Sets corresponding DBSR bit
33	UDE	0b0	<u>Unconditional Debug Event</u> Sets corresponding DBSR bit
34:35	MRR	0b00	<u>Most Recent Reset</u> Sets corresponding DBSR bit

Bit(s):	Field Name:	Init	Description
36	ICMP	0b0	<u>Instruction Complete Debug Event</u> Sets corresponding DBSR bit
37	BRT	0b0	<u>Branch Taken Debug Event</u> Sets corresponding DBSR bit
38	IRPT	0b0	<u>Interrupt Taken Debug Event</u> Sets corresponding DBSR bit
39	TRAP	0b0	<u>Trap Instruction Debug Event</u> Sets corresponding DBSR bit
40	IAC1	0b0	<u>Instruction Address Compare 1 Debug Event</u> Sets corresponding DBSR bit
41	IAC2	0b0	<u>Instruction Address Compare 2 Debug Event</u> Sets corresponding DBSR bit
42	IAC3	0b0	<u>Instruction Address Compare 3 Debug Event</u> Sets corresponding DBSR bit
43	IAC4	0b0	<u>Instruction Address Compare 4 Debug Event</u> Sets corresponding DBSR bit
44	DAC1R	0b0	<u>Data Address Compare 1 Read Debug Event</u> Sets corresponding DBSR bit
45	DAC1W	0b0	<u>Data Address Compare 1 Write Debug Event</u> Sets corresponding DBSR bit
46	DAC2R	0b0	<u>Data Address Compare 2 Read Debug Event</u> Sets corresponding DBSR bit
47	DAC2W	0b0	<u>Data Address Compare 2 Write Debug Event</u> Sets corresponding DBSR bit
48	RET	0b0	<u>Return Debug Event</u> Sets corresponding DBSR bit
49:58	///	0x0	<u>Reserved</u>
59	DAC3R	0b0	<u>Data Address Compare 3 Read Debug Event</u> Sets corresponding DBSR bit
60	DAC3W	0b0	<u>Data Address Compare 3 Write Debug Event</u> Sets corresponding DBSR bit
61	DAC4R	0b0	<u>Data Address Compare 4 Read Debug Event</u> Sets corresponding DBSR bit
62	DAC4W	0b0	<u>Data Address Compare 4 Write Debug Event</u> Sets corresponding DBSR bit
63	IVC	0b0	<u>Instruction Value Compare Event</u> Sets corresponding DBSR bit

10.7.7 Instruction Address Compare Registers (IAC1–IAC4)

The four IAC registers specify the addresses upon which IAC debug events should occur. Each of the IAC registers can be written from a GPR using **mtspr**, and can be read into a GPR using **mfspr**.

Register Short Name:	IAC1	Read Access:	Hypv
-----------------------------	------	---------------------	------

Decimal SPR Number:	312	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	IAC1	0x0	<u>Instruction Address Compare 1</u> A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified, or to blocks of addresses specified by the combination of the IAC1 and IAC2. Since all instruction addresses are required to be word-aligned, the two low-order bits of the Instruction Address Compare Registers are reserved and do not participate in the comparison to the instruction address.
62:63	///	0b00	<u>Reserved</u>

Register Short Name:	IAC2	Read Access:	Hypv
Decimal SPR Number:	313	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	IAC2	0x0	<u>Instruction Address Compare 2</u> A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified, or to blocks of addresses specified by the combination of the IAC1 and IAC2. Since all instruction addresses are required to be word-aligned, the two low-order bits of the Instruction Address Compare Registers are reserved and do not participate in the comparison to the instruction address.
62:63	///	0b00	<u>Reserved</u>

Register Short Name:	IAC3	Read Access:	Hypv
Decimal SPR Number:	314	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	IAC3	0x0	<u>Instruction Address Compare 3</u> A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified, or to blocks of addresses specified by the combination of the IAC3 and IAC4. Since all instruction addresses are required to be word-aligned, the two low-order bits of the Instruction Address Compare Registers are reserved and do not participate in the comparison to the instruction address.
62:63	///	0b00	<u>Reserved</u>

Register Short Name:	IAC4	Read Access:	Hypv
Decimal SPR Number:	315	Write Access:	Hypv

Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	IAC4	0x0	<u>Instruction Address Compare 4</u> A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified, or to blocks of addresses specified by the combination of the IAC3 and IAC4. Since all instruction addresses are required to be word-aligned, the two low-order bits of the Instruction Address Compare Registers are reserved and do not participate in the comparison to the instruction address.
62:63	///	0b00	<u>Reserved</u>

10.7.8 Data Address Compare Registers (DAC1–DAC2)

The four DAC registers specify the addresses upon which DAC debug events, or DVC debug events, or both should occur. Each of the DAC registers can be written from a GPR using **mtspr** and can be read into a GPR using **mfspr**.

Register Short Name:	DAC1	Read Access:	Hypv
Decimal SPR Number:	316	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DAC1	0x0	<u>Data Address Compare 1</u> A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified, or to blocks of addresses specified by the combination of the DAC1 and DAC2.

Register Short Name:	DAC2	Read Access:	Hypv
Decimal SPR Number:	317	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DAC2	0x0	<u>Data Address Compare 2</u> A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified, or to blocks of addresses specified by the combination of the DAC1 and DAC2.

Register Short Name:	DAC3	Read Access:	Hypv
Decimal SPR Number:	849	Write Access:	Hypv

Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DAC3	0x0	<u>Data Address Compare 3</u> A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified, or to blocks of addresses specified by the combination of the DAC3 and DAC4.

Register Short Name:	DAC4	Read Access:	Hypv
Decimal SPR Number:	850	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DAC4	0x0	<u>Data Address Compare 4</u> A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified, or to blocks of addresses specified by the combination of the DAC3 and DAC4.

10.7.9 Data Value Compare Registers (DVC1–DVC2)

The DVC registers specify the data values upon which DVC debug events should occur. Each of the DVC registers can be written from a GPR using **mtspr** and can be read into a GPR using **mfspr**.

Register Short Name:	DVC1	Read Access:	Hypv
Decimal SPR Number:	318	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DVC1	0x0	<u>Data Value Compare 1</u> A DAC1R, DAC1W debug event may be enabled to occur upon loads or stores of a specific data value specified in DVC1. DBCR2[DVC1M] and DBCR2[DVC1BE] control how the contents of the DVC1 is compared with the value.

Register Short Name:	DVC2	Read Access:	Hypv
Decimal SPR Number:	319	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DVC2	0x0	<u>Data Value Compare 2</u> A DAC2R, DAC2W debug event may be enabled to occur upon loads or stores of a specific data value specified in DVC2. DBCR2[DVC2M] and DBCR2[DVC2BE] control how the contents of the DVC2 is compared with the value.

10.7.10 Instruction Address Register (IAR)

The IAR indicates the address of the current instruction at the completion point, or of the last instruction that has passed the completion point.

Register Short Name:	IAR	Read Access:	Hypv
Decimal SPR Number:	882	Write Access:	Hypv
Initial Value:	0xffffffffffffc	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	bcfg

Bit(s):	Field Name:	Init	Description
0:61	IAR	0x3fffffff ffffff	<u>Instruction Address Register</u> Indicates the address of the current instruction at the completion point, or last instruction which has past the completion point. The completion point is the point at which all interrupts has been process and the instruction is guaranteed to complete.
62:63	///	0b00	<u>Reserved</u>

10.7.11 Instruction Match Mask Registers (IMMR)

The IMR and IMMR registers are used together to specify bits compared against an instruction, to determine if an instruction value compare (IVC) debug event should occur. A match occurs when the instruction data bitwise ANDed with the IMMR equals the IMR bitwise ANDed with the IMMR. The IMMR register can be written from a GPR using **mtspr** and can be read into a GPR using **mfspir**.

Register Short Name:	IMMR	Read Access:	Hypv
Decimal SPR Number:	881	Write Access:	Hypv
Initial Value:	0x00000000ffffff	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	MASK	0xffffffff	Instruction Mask

10.7.12 Instruction Match Registers (IMR)

The IMR and IMMR registers are used together to specify bits compared against an instruction, to determine if an instruction value compare (IVC) debug event should occur. A match occurs when the instruction data bitwise ANDed with the IMMR equals the IMR bitwise ANDed with the IMMR. The IMR register can be written from a GPR using **mtspir** and can be read into a GPR using **mfspir**.

Register Short Name:	IMR	Read Access:	Hypv
Decimal SPR Number:	880	Write Access:	Hypv
Initial Value:	0x00000000ffffff	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	MATCH	0xffffffff	Instruction Match

10.8 Debug configuration and control functions

PC Configuration Register 0 (PCCR0) is used to enable various debug modes, set or disable additional debug functions, and provides status and control for the Recoverable Error Counter. PCCR0 is a SCOM-accessible register with RW, WOAND and WOOR access. It is connected to the PC unit debug configuration ring, and is configurable through scanning during the POR sequence. See *PC Configuration Register 0 (PCCR0) Definition* on page 421 for the register layout description.

10.8.1 Core debug modes

PCCR0[EnabDebugMode] places the core in *debug mode*. Debug mode provides an enable for the trace and trigger logic throughout the core (primarily through the ACT pin), as well as gating signals controlling the following debug functions:

- single-stepping instructions
- THRCTL[DisabAsynCrpts, DisabTimebase, DisabDecrem]
- PCCR0[EnableFastClockstop]

Setting PCCR0[EnabRamOperations] enables *Ram mode* operations through the RAMI, RAMC and RAMD registers. Specifically, it enables the ACT pin to these registers; and gates the RamMode, RamExecute and other bits of the RAMC register (i.e. MSR override enables, FlushThread).

External Debug Mode (EDM) is activated whenever the PCCR0[Tx_DBA] bits have a non-zero value. External debug mode provides a means for an architected debug compare event (such as an Instruction Address Compare (IAC)) to allow an unarchitected debug action to be performed in place of a debug interrupt. The DBCR0[EDM] bit indicates whether or not the thread is in external debug mode. The debug actions enabled by the decoded PCCR0[Tx_DBA] bits are described below:

- 000 - No action.
- 001 - Reserved (no action).
- 010 - Stop this thread.
- 011 - Stop all threads.
- 100 - Activate a debug event error (sets FIR bit) for this thread.
- 101 - Activate external signal (*ac_an_debug_trigger*). This is a single cycle pulse.
- 110 - Activate external signal and stop this thread.
- 111 - Activate external signal and stop all threads.

10.8.2 Additional debug functions

Setting PCCR0[EnabErrorInjection] gates ERRINJ register outputs to force errors for debug and recovery testing purposes.

an_ac_debug_stop is a core input signal used by external debug tools to simultaneously stop all core threads. PCCR0[EnabExtDebugStop] gates the external debug stop input, and must be set in order for the debug stop function to work.

PCCR0[DisabRamXstopReport] blocks the reporting of checkstop errors outside the core (*ac_an_checkstop* or *ac_an_local_checkstop*) when Ram mode is active. A checkstop error still sets a bit in the FIR, as well as the RAMC[Checkstop] bit.

PCCR0[EnabFastClockstop] is used to force all core thold signals active on a checkstop error, thereby quickly stopping all clocks. This allows debug tools to access core facilities through scanning close to the point of a failure. This function is also gated by debug mode (PCCR0[EnabDebugMode] set).

Power management controls activate run tholds in order to stop clocks, and put the core in a power-savings state. PCCR0[DisabPowerSavings], when set, blocks the run tholds from being set. All other power management operations and status (i.e. *ac_an_rvwinkle_mode*) will still occur, but latches controlled by the run tholds will remain clocked and active.

PCCR0[DisabOverrunChks] disables all overrun checking and associated status. This includes Ram overrun checking (indicated by RAMC[Overrun]), as well as instruction stepping overrun (indicated by THRCTL[Instr-StepOverrun]).

10.8.3 Recoverable error counter

The recoverable error counter is a 4 bit counter that increments whenever an unmasked recoverable error occurs. Upon the count value reaching 15 an error signal is activated to the FIR logic, and the next recoverable error will cause the counter to wrap back to 0.

The count value can be read to obtain the current value, or written (RW access only) to preset or clear it.

10.8.4 PC Configuration Register 0 (PCCR0) Definition

Table 4. PC Configuration Register 0

Note: Bits 57 through 59 are specific to thread 1, and are unimplemented on single-threaded versions of A20.

Short Name	PCCR0	Access	RW, WOAND, WOOR
Register Address	x'33' RW x'34' WOAND x'35' WOOR	Scan Ring Initial Value	dcfg 0x0000000000000000
Bit Num	Function	Init	Description
0:31	Reserved (Unimplemented)	0	
32	EnabDebugMode	0	This bit places the core in debug mode. It is used to enable debug logic such as the trace and trigger mux controls and buses. Enabling debug mode allows various debug functions to be performed (i.e. instruction stepping and miscellaneous debug controls such as THRCTL[DisabAsynclrpts, DisabTimebaselrpts, DisabDeclrpts]).
33	EnabRamOperations	0	This bit enables Ram mode operation through the RAMI, RAMC and RAMD registers. It is gated with various RAMC control bits, such as: RamMode, RamExecute, EnabMsrOverrides and FlushThread.
34	EnabErrorInjection	0	This bit enables control signals set in the ERRINJ register to force errors in order to test error recovery methods.
35	EnabExtDebugStop	0	When set, this bit enables the input signal <i>an_ac_debug_stop</i> to stop all threads.
36	DisabRamXstopReport	0	Setting this bit will block the reporting of checkstop errors outside of the core when in Ram mode. A checkstop error would still be indicated by the RAMC _{Checkstop} bit, and the core's local FIR.
37	EnabFastClockstop	0	This bit enables a checkstop error to directly force all core tholds active, thereby quickly stopping clocks. Note: The core must be in debug mode (PC Configuration Register 0, bit 32 = 1) for this bit to be valid.
38	DisabPowerSavings	0	This bit blocks power-savings controls from raising the run tholds, and thereby forcing off the associated latch clocks. Other power-savings control signals (i.e. <i>ac_an_rvwinkle_mode</i>) will still be active, but all core latch clocks will remain enabled.
39	DisabOverrunChks	0	Setting this bit stops overrun checking logic from blocking Ram or instruction step operations when an overrun condition is detected. Also, overrun conditions will not be indicated through the appropriate status bits (RAMC[Overrun] and THRCTL[InstrStepOverrun]).
40:43	Reserved (Spare)	0	

Table 4. PC Configuration Register 0

Note: Bits 57 through 59 are specific to thread 1, and are unimplemented on single-threaded versions of A20.

44:47	Reserved (Unimplemented)	0	
48:51	RecovErrorCounter	0	This 4 bit counter increments whenever an unmasked recoverable error occurs. When the count value reaches 15, an error bit will be set in FIR0. The count value can be read to obtain the current value, or written to preset or clear it. Note: Write access to the Recoverable Error Counter is only supported through the "RW" SCOM address.
52	Reserved (Unimplemented)	0	Additional actions that can be selected when a debug compare event occurs for the indicated thread (sets DBCR0[EDM] status bit). Debug Action Select: 000 - No action 001 - Reserved (no action) 010 - Stop Specified Thread 011 - Stop All Threads
53:55	T0_DBA	000	
56	Reserved (Unimplemented)	0	100 - Activate the thread's debug event error (Sets FIR bit) 101 - Activate External Signal (<i>ac_an_debug_trigger</i> pulse) 110 - Activate External Signal and Stop Specified Thread 111 - Activate External Signal and Stop All Threads
57:59	T1_DBA	000	
60:63	Reserved (Unimplemented)	0	

10.9 Thread control and status

The *Thread Control and Status Register* (THRCTL) allows debug control of thread operations such as start/stop, single-step, and the monitoring of thread status bits. THRCTL is a SCOM-accessible register with RW, WOAND and WOOR access. It is connected to the PC unit boot configuration ring, and is configurable through scanning during the POR sequence. See *Thread Control and Status Register (THRCTL)* on page 86 for register layout description.

10.9.1 Thread control functions

THRCTL[Tx_Stop] causes instruction fetching to stop, and the thread to enter the stopped state. When in the stopped state the following events occur:

- The IU stops fetching instructions; the next instruction address is preserved and used upon restart
- Any instructions at the completion point will complete, all other instructions are flushed. If a stop is attempted in the middle of a microcode sequence, the hardware will allow the microcode sequence to complete before stopping.
- The corresponding THRCTL[Tx_Run] status bit is cleared to indicate that the thread is stopped.
- Snoop invalidate and TLB invalidate requests from the A20/L2 interface are still handled as normal.

Besides activation through a SCOM write to the THRCTL register, the THRCTL[Tx_Stop] bit is also set through the following stop conditions:

- An enabled checkstop error
- A debug compare event (when the PCCR0[Tx_DBA] decoded bits are configured to stop threads)
- Execution of a **dnh** instruction when enabled by CCR4[EN_DNH]
- An **attn** instruction when enabled by CCR2[EN_ATTEN]

In order to single-step instructions, the core should first have debug mode active (PCCR0[EnabDebugMode] set), with the designated thread quiesced using the THRCTL[Tx_Stop] control. Each time THRCTL[Tx_Step] is set, the IU will issue one instruction. Upon completion of the stepped instruction, the THRCTL[Tx_Step] bit is cleared. If the thread is not stopped when a single-step is initiated (THRCTL[Tx_Step] set while THRCTL[Tx_Run] still active), then an overrun error will be detected and indicated by the THRCTL[InstrStepOverrun] bit.

Additional THRCTL bits are used to disable certain interrupt and timer controls while the core is stopped for debug. These functions require that the core is in debug mode (PCCR0[EnabDebugMode] set), and that the thread is stopped due to a pervasive stop request (THRCTL[Tx_Stop], *an_ac_debug_stop*, or *an_ac_pm_thread_stop*). In each case, the function will be re-enabled whenever the thread returns to a running state (including activation during a single-step pulse). The THRCTL bit and corresponding disable actions are shown below:

- THRCTL[DisabAsynclrpts] - disables asynchronous interrupts for stopped thread
- THRCTL[DisabTimebase] - blocks incrementing of the Timebase when *all* threads are stopped
- THRCTL[DisabDecrem] - blocks decremter count when the thread is stopped

10.9.1.1 Example procedure to perform instruction stepping

- a. Put core in debug mode. [SCOM write PCCR0(32)='1'] .
- b. Stop thread 0 by setting the T0_Stop bit. [SCOM write THRCTL(32)='1'] .
- c. Verify thread 0 has quiesced by reading the T0_Run bit. [SCOM read THRCTL(40)='0'].
- d. SCOM write THRCTL(36)='1' (T0_Step bit) to activate a single instruction step pulse.
(THRCTL(36 and 40) are both reset to '0' when the stepped instruction completes execution.)
- e. Repeat SCOM writes to THRCTL(36) until all stepped instructions have been executed

10.9.2 Thread status

THRCTL[Tx_Run] indicates that the thread is active when set, and quiesced when cleared. Core controls that cause a thread to stop or start execution, such as THRCTL[Tx_Stop], *an_ac_pm_thread_stop*, CCR0[WE] and the TENC/TENS registers, indicate a thread's quiesced/running status through THRCTL[Tx_Run].

Note: The *an_ac_pm_fetch_halt* input blocks new instructions from being fetched but does not cause a thread to stop. The THRCTL[Tx_Run] bit is not affected by the state of *an_ac_pm_fetch_halt*.

Thread stop status is provided by various bits in order to isolate the specific reason a thread is not running. THRCTL[DebugStopInput] is active when the core is stopped due to the external stop input signal. This means that both the *an_ac_debug_stop* input, and its enable (PCCR0[EnabExtDebugStop]) control are active. THRCTL[Tx_StopRequest] is a 4 bit field which indicates why the corresponding thread is stopped. The individual bits of this field provide the following thread stop status:

- Power management (Set when a power-savings instruction is executed. Also set when either the *an_ac_pm_thread_stop* or *an_ac_pm_fetch_halt* inputs are active)
- An enabled checkstop error was detected
- A debug related operation. Either execution of the **dnh** instruction, or an external mode debug compare event (stop configured by PCCR0[Tx_DBA] decode)
- An **attn** instruction

The THRCTL[InstrStepOverrun] bit indicates that single stepping was attempted on *any* thread, and that thread was still running. The overrun condition occurs when THRCTL[Tx_Step] is set, and the corresponding THRCTL[Tx_Run] bit is still active. When an overrun condition is detected, the instruction step operation will be blocked (no execution attempted) and THRCTL[Tx_Step] will be cleared.

THRCTL[RamErrorStatus] is activated whenever a Ram operation results in setting one of the RAMC status bits (RAMC[Unsupported, Overrun, Interrupt, Checkstop]). It provides cumulative status for multiple Ram operations when the RAMC status bits are not read between each new Rammed instruction.

10.9.3 Thread Control and Status Register (THRCTL) Definition

Table 5. Thread Control and Status Register

Note: Bits 33, 37, 41, 48 through 51 are specific to thread 1, and are unimplemented on single-threaded versions of A2o.

Short Name	THRCTL	Access	RW, WOAND, WOOR
Register Address	x'30' RW x'31' WOAND x'32' WOOR	Scan Ring Initial Value	bcfg 0x0000000000000000
Bit Num	Field Name	Init	Description
0:31	Reserved (Unimplemented)	0	
32	T0_Stop	0	When set, this thread will stop instruction fetch and enter a stopped state. Instructions currently in the pipeline will continue to completion. When reset, program execution resumes at the next instruction address available before stopping.
33	T1_Stop	0	In addition to a SCOM write, these bits may be set by the following conditions: <ul style="list-style-type: none"> An enabled checkstop error Either execution of a dnh instruction, or a debug compare event (when PCCR0[Tx_DBA] bits are configured to stop the thread upon occurrence of the compare event). An attn instruction when configured by CCR2[EN_ATTN].
34:35	Reserved (Unimplemented)	0	
36	T0_Step	0	Writing a '1' to this location causes one instruction for this thread to be issued. This bit will be reset upon completion of the stepped instruction.
37	T1_Step	0	Note: Prior to activating Tx_Step, the corresponding thread should be stopped (Tx_Run=0) to avoid an overrun occurring. Instruction stepping can be performed when the thread is stopped due to activation of either THRCTL[Tx_Stop] or the <i>an_ac_debug_stop</i> input. Note: The core must be in debug mode (PC Configuration Register 0, bit 32 = 1) for the single-step signals to be valid.
38:39	Reserved (Unimplemented)	0	
40	T0_Run	0	Status bit indicating that the thread is in a running state when set.
41	T1_Run	0	When '0', the thread is stopped. This bit is read only. Writes will have no effect.
42	Reserved (Unimplemented)	0	
43	DebugStopInput	0	Status bit indicating that an external debug stop request is active. An external debug stop occurs when the <i>an_ac_debug_stop</i> input signal is active and enabled through the PCCR0[EnabExtDebugStop] bit. The external debug stop input affects all core threads. This bit is read only. Writes will have no effect.

Table 5. Thread Control and Status Register

Note: Bits 33, 37, 41, 48 through 51 are specific to thread 1, and are unimplemented on single-threaded versions of A20.

44:47	T0_StopRequest	0	<p>These bits summarize the T0 stop requests that have been activated.</p> <p>44 - A power management related stop. This could be the result of a power-savings (wait) instruction, the <i>an_ac_pm_thread_stop</i> or <i>an_ac_pm_fetch_halt</i> inputs. This bit is read only. Writes will have no effect.</p> <p>45 - An enabled checkstop error has been detected.</p> <p>46 - A debug related stop; either through executing a dnh instruction, or through an external mode debug compare event.</p> <p>47 - An attn instruction (enabled by CCR2[EN_ATTN]) occurred.</p> <p>Bits 45 through 47 are writeable; they are activated by the same stop controls which cause THRCTL[T0_Stop] to be set.</p>
48:51	T1_StopRequest	0	<p>These bits summarize the T1 stop requests that have been activated.</p> <p>48 - A power management related stop. This could be the result of a power-savings (wait) instruction, the <i>an_ac_pm_thread_stop</i> or <i>an_ac_pm_fetch_halt</i> inputs. This bit is read only. Writes will have no effect.</p> <p>49 - An enabled checkstop error has been detected.</p> <p>50 - A debug related stop; either through executing a dnh instruction, or through an external mode debug compare event.</p> <p>51 - An attn instruction (enabled by CCR2[EN_ATTN]) occurred.</p> <p>Bits 49 through 51 are writeable; they are activated by the same stop controls which cause THRCTL[T1_Stop] to be set.</p>
52	DisabAsynclrpts	0	<p>This bit provides a global disable to any thread's asynchronous interrupts as long as the associated thread is stopped through pervasive (THRCTL[Tx_Stop], <i>an_ac_debug_stop</i> or <i>an_ac_pm_thread_stop</i>) controls. The asynchronous interrupts are re-enabled whenever the thread is put in a running state (this includes activation during a single-step pulse).</p> <p>Note: The core must be in debug mode (PC Configuration Register 0, bit 32 = 1) for this signal to be valid.</p>
53	DisabTimebase	0	<p>Setting this bit will block incrementing of the Timebase whenever all threads are stopped through pervasive (THRCTL[Tx_Stop], <i>an_ac_debug_stop</i> or <i>an_ac_pm_thread_stop</i>) controls. The Timebase count will continue whenever any thread is in a running state (this includes activation during a single-step pulse).</p> <p>Note: The core must be in debug mode (PC Configuration Register 0, bit 32 = 1) for this signal to be valid.</p>
54	DisabDecrem	0	<p>Setting this bit blocks the counting of any thread's decremter, as long as that thread is stopped through pervasive (THRCTL[Tx_Stop], <i>an_ac_debug_stop</i> or <i>an_ac_pm_thread_stop</i>) controls. Decrementer counting will be re-enabled whenever the thread is put in a running state (this includes activation during a single-step pulse).</p> <p>Note: The core must be in debug mode (PC Configuration Register 0, bit 32 = 1) for this signal to be valid.</p>
55:61	Reserved (Spare)	0	
62	InstrStepOverrun	0	<p>This bit is set when instruction stepping on any thread results in an overrun condition. The overrun occurs when THRCTL[Tx_Stop] is activated while the corresponding thread is still running (THRCTL[Tx_Run]=1).</p> <p>As a result of the detected overrun, the instruction-step operation will be blocked (no execution attempted) and THRCTL[Tx_Stop] will be cleared.</p>

Table 5. Thread Control and Status Register

Note: Bits 33, 37, 41, 48 through 51 are specific to thread 1, and are unimplemented on single-threaded versions of A2o.

63	RamErrorStatus	0	<p>This bit is activated whenever a Ram operation results in setting one of the RAMC status bits (RAMC[Unsupported, Overrun, Interrupt, Check-stop]).</p> <p>It provides cumulative status for multiple Ram operations when the RAMC status bits are not read between new Rammed instruction.</p>
----	----------------	---	---

10.10 Instruction Stuffing

The core provides a method of forcing instructions into the processor pipeline through a set of SCOM accessible registers, and a special operating mode called *Ram mode*. When in Ram mode, instructions can be inserted into a selected thread's pipeline, allowing code execution, as well as access to GPRs, FPRs and SPRs. This section describes instruction stuffing and the associated Ram facilities, provides an overview of the Ram process, as well as an example Ram procedure.

10.10.1 Ram registers

The registers listed below are used to support Ram operations. Refer to the indicated sections for the register layout and bit descriptions.

Ram Command (RAMC) Register	<i>Section 15.2.3.9</i> beginning on page 726
Ram Data (RAMD) Register	<i>Section 15.2.3.8</i> beginning on page 725
Ram Data High (RAMDH) Register	<i>Section 15.2.3.8</i> beginning on page 725
Ram Data Low (RAMDL) Register	<i>Section 15.2.3.8</i> beginning on page 725
Ram Instruction (RAMI) Register	<i>Section 15.2.3.9</i> beginning on page 726
Ram Instruction and Command (RAMIC) Register	<i>Section 15.2.3.9</i> beginning on page 726
Shadowed Ram Data (SRAMD) Register	<i>Section 14.5</i> beginning on page 529

10.10.2 Ram instructions

The RAMI register specifies the 32 bit Ram instruction field.

While in Ram mode the next instruction put in the pipeline when the RAMC[Execute] bit is activated comes from the RAMI register. An instruction that alters the next instruction address (i.e. branch) will not be executed in place of the Rammed instruction.

In order to save state, GPRs and FPRs that are targets of any Rammed instructions must be saved off prior to executing Ram, and restored upon completion. Alternatively, a set of scratch registers may be specified in place of the source and target registers for an instruction by setting the appropriate RAMC instruction field extension bits (RAMC(32:35)). See *Using scratch registers as temporary storage* on page 427.

10.10.2.1 Supported Ram instructions

Almost any valid A2O instruction can be inserted in the pipeline through Ram. The only instructions which are not supported are those executed as microcoded ops (see *A2 Core Instructions by Mnemonic* on page 736); this also includes instructions such as unaligned loads or stores, which are processed through microcode as individual byte accesses.

While in Ram mode, the next instruction to be Rammed comes from the RAMI register. Any instruction causing a jump to a different code location will update the IFAR, which will redirect software once Ram mode is exited.

Some examples of problem instructions while in Ram mode are:

- branches - next instruction will not come from the branch target address
- return from interrupts - next instruction will not come from the associated SRR0 address
- any instruction or event (ie. timer, external) that would cause an interrupt to be taken

Depending on the situation, executing the types of instructions listed above might still be desirable. Ram could be used to point to a specific code location; once Ram mode is disabled instruction execution would then commence from the value specified in the IFAR.

10.10.2.2 Using scratch registers as temporary storage

The fixed point and floating point units each implement a set of four 64-bit scratch registers (GPR(32:35), FPR(32:35)), which are available as temporary storage when selecting registers for Rammed instructions. Fields in the RAMC register (Extend_InstrTgt1, Extend_InstrSrc1, Extend_InstrSrc2, Extend_InstrSrc3) specify that extended target and source registers should be used in place of the architected GPR or FPR facilities. In the Rammed instruction, r0 through r3 must be used along with the RAMC extended target and source bits, in order to correctly specify the scratch register number (i.e. r0 = GPR(32), r1 = FPR(33), etc).

The fixed point instruction summary table (see *A2 Core Instructions by Mnemonic* on page 736) contains columns for the Tgt1, Src1, Src2 and Src3 fields. For each instruction, these columns indicate how the target and source registers specified in the instruction opcode are implemented. As an example, to use GPR(32) as the source register for RB in the **nor** instruction, specify r0 in place of RB in the opcode and set RAMC[Extend_InstrSrc2] during the Ram operation. Scratch register GPR(32) will be selected as the Src2 register when the **nor** instruction is executed.

For floating point opcodes, the RAMC extended target and source registers are mapped as follows: Extend_InstrTgt1 to an instruction target register, Extend_InstrSrc1 to RA and FRA, Extend_InstrSrc2 to RB and FRB, Extend_InstrSrc3 to FRC. As an example, to use FPR(32) as the source register for FRB in the **fmr** instruction, specify r0 in place of FRB in the opcode and set RAMC[Extend_InstrSrc2] during the Ram operation. Scratch register FPR(32) will be selected as the FRB value when the **fmr** instruction is executed.

It is valid to set a RAMC extended target/source bit active when the corresponding target/source register is not used by the Rammed instruction. The unused RAMC[Extend_InstrTgt1 and Extend_InstrSrc1-3] fields will be ignored by the hardware. This allows setting the RAMC extended target and source fields to be simplified to just 0xF, whenever all of the target and source fields for the Rammed instruction use scratch registers.

Note: Care should be taken with instructions that resolve the RA field to 0, such as the load immediate (i.e. “li RT,value” resolves to “addi RT,0,value”). If the RAMC[Extend_InstrSrc1] field is set, this would actually specify R32 as the source 1 field, producing inaccurate results.

10.10.3 Ram control bits

The RAMC register provides control bits that enable Ram Mode, select a thread and start execution of the Rammed instruction. It also includes additional debug controls for overriding specific processor functions. The following functions are used to control Ram operations:

- **RAMC[RamMode]** - Ram mode is used to gate Ram related controls and functions, and must be active for a Ram operation to be initiated.
- **RAMC[RamThread]** - This field determines which thread the Rammed instruction will be executed on. For A2O it is a one bit field, selecting thread 0 when inactive and thread 1 when set.
- **RAMC[RamExecute]** - The execute pulse initiates a new Ram operation. Data in RAMI at the time it is activated determines the Rammed instruction. Other RAMC bits can alter how the instruction is implemented by hardware.
- **MSR overrides** - These controls enable substitution of certain MSR bits for the Rammed thread. This capability enables instructions and access to SPRs for debug, where normal program permissions would restrict that access. The RAMC override replaces the MSR output signal, but does not alter the state of the corresponding MSR bit. The RAMC fields controlling this function are shown below:
 - EnabMsrOverrides** - Enables overriding of MSR bits for the Rammed instruction. The state of the individual MSR overrides will be used in place of the actual MSR bits.
 - Override_MSR[PR]** - Used in place of the corresponding MSR[PR] bit.
 - Override_MSR[GS]** - Used in place of the corresponding MSR[GS] bit.
 - Override_MSR[DE]** - Used in place of the corresponding MSR[DE] bit.
- **Miscellaneous debug control** - potential workaround for when an instruction fails to complete. This function is **reserved for engineering use only**, and are not supported for general debug.
 - FlushThread** - When activated, completion logic will flush the current instruction.

10.10.4 Ram status bits

RAMC status bits indicate if the previously executed Ram instruction has completed, and whether or not any errors occurred.

Since subsequent writes to the RAMC register can replace active status bits, one of the following methods should be used in order to determine if a series of Rammed instructions completed successfully:

1. Read the RAMC register after each Ram operation and verify correct status (i.e. RAMC(60:62) = 0; RAMC(63) = 1).
2. On subsequent Ram operations use the RAMC's OR mask address to just activate the RAMC[Execute] bit. The value of other RAMC bits, such as Mode, Thread, and MSR Overrides will remain in the state written through the first RAMC write. Status bits set during a Rammed instruction will be preserved between SCOM write operations that use the OR mask address. A RAMC read to verify status can then be performed after completion of a series of Ram operations.

Note: This method requires separate SCOM write operations to RAMI and RAMC as 32 bit registers. The combined 64 bit RAMIC register is only accessible at a RW address.

3. After a series of Ram operations, read the THRCTL register to verify that the RamErrorStatus bit is cleared. THRCTL[RamErrorStatus] is set when a Ram operation activates an error status bit (RAMC[Unsupported,Overrun, Interrupt, Checkstop]). Since it is not affected by RAMC writes that initiate new Ram operations, it provides cumulative status.

The status of Ram operations are determined by the register bits described below:

- **RAMC[Done]** - The Done bit indicates that the previous Rammed instruction has completed and the associated target register data is valid in RAMD. In addition to a SCOM write, the Done bit will be cleared each time the RAMC[Execute] bit is activated to start a new Ram operation.
- **RAMC[Unsupported]** - An invalid Ram instruction was issued. Operations not supported for Ramming are:
 - Any instruction defined to be implemented via microcode. Refer to the instruction summary tables (*Appendix A-1 A2 Core Instructions by Mnemonic* on page 736).
 - Any instruction requiring microcode intervention in order to complete. This includes unaligned load or store instructions which are processed through microcode as individual byte accesses.

The unsupported Rammed instructions are treated as no-ops.

- **RAMC[Overrun]** - The Rammed instruction created an overrun condition. As a result the instruction will be blocked (not issued). An overrun is indicated when any of the following occurs:
 - Executing a Ram instruction while the thread is still running (THRCTL[Tx_RUN] is active).
 - Executing a second Ram instruction before RAMC[Done] was activated for the first Ram instruction.
 - Ram Mode ends while a Rammed instruction is in process (RAMC[Done] has not yet gone active).

Note: When a Rammed instruction is issued it sets an overrun counter, which is cleared when the corresponding Ram done is returned. If a fail scenario occurs where the Rammed instruction does not complete, it is possible that the overrun counter would remain set, and the RAMC[Overrun] bit would not clear. In such a situation the active overrun condition would block additional attempts to issue Ram operations.

There are two methods to reset the overrun counter and force the RAMC[Overrun] bit inactive: 1) clearing RAMC[RamMode] will reset the overrun counter; 2) setting PCCR0[DisabOverrunChks] will disable all overrun checking and allow Ram operations to continue.

- **RAMC[Interrupt]** - This bit indicates that the Rammed instruction resulted in an enabled exception. Examination of relevant facilities (i.e. SRR1, GSRR1, etc) is required in order to determine the cause of the interrupt.

Entering Ram Mode does not fence interrupts. If disabling interrupts is required, the user must ensure the appropriate MSR bits have been cleared. Also, several bits in the THRCTL register support disabling of asynchronous interrupts and timer facilities while a thread is stopped - refer to *Section 10.9 Thread control and status* on page 422.

- **RAMC[Checkstop]** - This bit indicates that while Ram mode was active, one of the FIRs contained an enabled checkstop error. The cause of the error may or may not be due to a Rammed instruction. Examination of the FIRs and other facilities is required in or to determine the reason the error occurred.

When PCCR0[DisabRamXstopReport] is set (with Ram mode active), the reporting of checkstop errors outside of the core is disabled. Checkstop errors are still indicated through the FIRs and RAMC[Checkstop] bit.

- **THRCTL[RamErrorStatus]** - This bit is activated whenever a Ram operation results in setting one of the RAMC status bits (RAMC[Unsupported, Overrun, Interrupt, Checkstop]).

It provides cumulative status for multiple Ram operations when the RAMC status bits are not read between new Rammed instructions.

10.10.5 Ram data

The result of any Rammed instruction is written to the RAMD register, and can be recovered using the appropriate SCOM address (RAMD, RAMDH or RAMDL).

The Ram facilities provide two methods for loading GPRs with data. One method requires breaking data up into 16 bit groups and using Power ISA immediate instructions; the other allows 64 bit writes through the Shadowed RAMD (SRAMD) register.

In an example of the first method shown below, 64 bits of data (0x0123456789ABCDEF) is loaded into R2 through a series of **ori** and rotate instructions.

```
ori  r2,r2,0x0123
ori  r2,r2,0x4567
rldicr r2,r2,32,31
ori  r2,r2,0x89AB
ori  r2,r2,0xCDEF
```

The second method uses a SCOM write to the RAMD register to directly load 64 bits into the SRAMD SPR. Then by Ramming a **mfspir** instruction, the data can be moved into a GPR.

```
SCOM write: Addr=0x2D, Data=0x0123456789ABCDEF // data written to both RAMD and SRAMD
SCOM write: Addr=0x28, Data=0x7C5EDAA600098000 // mfspr r2,sramd
```

Note: A SCOM write to any RAMD address (RAMD, RAMDH or RAMDL) will load SRAMD with the full 64 bit RAMD value (RAMD(0:63)). SRAMD changes value only upon a SCOM operation that writes the RAMD register. It will not change value due to RAMD being written with results of a Rammed instruction.

10.10.6 Ram Mode overview

10.10.6.1 Basic Ram process

To perform instruction stuffing on the A2O core, the following basic steps should be used:

1. The thread to be Rammed must be put in a stopped state (THRCTL[Tx_STOP]). Only the thread selected for instruction stuffing needs to be stopped. The other thread can continue instruction execution at normal speed.
2. Ram operations must be enabled (PCCR0[EnabRamOperations]).
3. The instruction to be Rammed is loaded in the RAMI register
4. Execution of the Rammed instruction is initiated by writing the RAMC register. At a minimum, the following RAMC fields must be in a valid state:
 - RAMC[RamMode] = 1
 - RAMC[RamThread] set to correct thread number
 - RAMC[RamExecute] = 1
5. The RAMC[Done] bit indicates when the Rammed instruction has completed. Additional status bits (RAMC[Unsupported, Overrun, Interrupt, Checkstop]) are available to determine if the previously Rammed instruction completed successfully.

- Output data from the Rammed instruction will be available in the RAMD register upon completion of the Ram operation (when RAMC[Done] set).

The Ram registers provide no address information, and there is no effective address associated with Rammed instructions. The IFAR will contain whatever address was valid at the time the thread was stopped. Each Rammed instruction can be referenced through a unique ITAG value.

10.10.6.2 Example Ram procedure

Table 10-6 below shows a series of SCOM operations which use Ram to backup the MSR and IAR, and then change their value. This example tries to show a basic instruction stuffing procedure, as well as highlight some of the features of the Ram facilities:

- Makes use of scratch registers for read/write operations with the SPRs.
- Sets selected MSR bits by ramming **ori** instructions
- Loads SRAMD, and writes 64 bits of data to a scratch register by Ramming a **mfspr** instruction
- Uses THRCTL[RamErrorStatus] to verify that the Rammed instructions completed without errors

In this example, a single 64 bit write to the RAMIC address was made instead of two 32 bit accesses that load first RAMI, and then starts the Ram operation by writing RAMC. All instructions specify scratch registers for the GPR fields, consequently the extended target/source bits (RAMC(32:35)) could be set to 0xF in all Ram operations. For this series of Rammed instructions, the RAMC value did not have to change, and had the following settings:

- RAMC(32:35)=0xF - all GPR accesses use scratch registers; no RA source field resolved to 0.
- RAMC(44)=1 - Sets RamMode
- RAMC(46)=0 - RamThread specifies thread 0
- RAMC(47)=1 - Activates RamExecute bit
- RAMC(48:51)=0x8 - Enables MSR overrides, and sets hypervisor access

Table 6. Example Ram procedure

SCOM Register	SCOM Address (Access)	SCOM Data	Description
THRCTL	0x30 (RW-write)	0x80000000	Set T0_STOP; ensure RamErrorStatus cleared
THRCTL	0x30 (RW-read)	0x00000000	Verify T0_STOP=1; T0_RUN=0
PCCR0	0x35 (WOOR)	0x40000000	Set EnabRamOperations
RAMIC	0x28 (RW-write)	RAMI=0x7C0000A6 RAMC=0xF0098000	mfmsr r32 (Backup MSR to scratch register R32)
RAMIC	0x28 (RW-write)	RAMI=0x7C2000A6 RAMC=0xF0098000	mfmsr r33 (Backup MSR to scratch register R33)
RAMIC	0x28 (RW-write)	RAMI=0x64218000 RAMC=0xF0098000	oris r33,r33,0x8000 (Ensure MSR[CM] set)
RAMIC	0x28 (RW-write)	RAMI=0x60212000 RAMC=0xF0098000	ori r33,r33,0x2000 (Ensure MSR[FP] set)
RAMIC	0x28 (RW-write)	RAMI=0x7C200124 RAMC=0xF0098000	mtmsr r33 (Load MSR with ORed in bits)
RAMIC	0x28 (RW-write)	RAMI=0x7C32DAA6 RAMC=0xF0098000	mfspir r33, iar (Backup IAR to scratch register R33)
RAMD	0x2D (RW-write)	0x00AABCCDDDEEFF00	Load data into RAMD and SRAMD registers

SCOM Register	SCOM Address (Access)	SCOM Data	Description	
RAMIC	0x28 (RW-write)	RAMI=0x7C5EDAA6 RAMC=0xF0098000	mf spr r34, sramd	(Load scratch register R34 from SRAMD)
RAMIC	0x28 (RW-write)	RAMI=0x7C52DBA6 RAMC=0xF0098000	mt spr iar, r34	(IAR updated from R34)
<p>At this point a new value has been written to the IAR. The thread could be single-stepped from this location, or additional Ram instructions could be executed. The sequence below restores state, ends Ram, and re-starts the thread.</p>				
RAMIC	0x28 (RW-write)	RAMI=0x7C32DBA6 RAMC=0xF0098000	mt spr iar, r33	(Restore IAR from scratch register R33)
RAMIC	0x28 (RW-write)	RAMI=0x7C000124 RAMC=0xF0098000	mt msr r32	(Restore MSR from scratch register R32)
PCCR0	0x34 (WOAND)	0xBFFFFFFF	Uses AND mask to clear only EnabRamOperations bit	
THRCTL	0x31 (WOAND)	0x7FFFFFFF	Uses AND mask to clear only T0_STOP bit	
THRCTL	0x30 (RW-read)	0x00000000	Verify T0_RUN=1; RamErrorStatus=0	

10.11 Direct Access to I-Cache and D-Cache Directories

The Directory Read Control Registers (IUDBG0 and XUDBG0) allow reads of the content of a specific, physical location in the I-cache or D-cache directory and store the data into debug registers. No address translation is done because only the content of a specific cache directory location is accessed. The contents of the instruction or data cache directory entry associated with the selected cache block are placed into IUDBG1 and IUDBG2 for I-cache reads or into XUDBG1 and XUDBG2 for D-cache reads, and the Done bit is set in the associated IUDBG0 or XUDBG0 register. To guarantee that an **mf spr** instruction obtains the results of the directory read, software must first read the Directory Read Control Register Done bit = 1.

Note: On version 1 hardware, all processor threads and all other mechanisms that can possibly generate a coherent cache invalidate to the processor must be quiesced.

Still True?

10.11.1 General Read D-Cache Directory Sequence for L1 D-Cache

#D-Cache Directory written to XUDBG1 and XUDBG2

```
scm write RAMI ← li, Rx, [address(row, way), execute = 1] #data for XUDBG0(49:62)
scm write RAMC ← 0x000X0000 #set Ram mode; Ram thread; Ram execute
scm write RAMI ← mt spr XUDBG0, Rx #data written to XUDBG0 on Ram execute
scm write RAMC ← 0x000X0000 #set Ram mode; Ram thread; Ram execute
```

```
A2: XUDBG1/XUDBG2 ← L1 D-Cache directory(address(row, way)) #hardware read directory
```

#software poll for done = 1

```

scm write RAMI ← mfspr Rx, XUDBG0      #XUDBG0 data written to Rx on Ram execute
scm write RAMC ← 0x000X0000           #set Ram mode; Ram thread; Ram execute
scm write RAMI ← xori Rx, Rx, 0       #XUDBG0 on slowSPR bus; requires second Ram
                                       operation
scm write RAMC ← 0x000X0000           #set Ram mode; Ram thread; Ram execute
scm read RAMD: wait for done = 1      #data valid in RAMD
    
```

#recover XUDBG1 data

```

scm write RAMI ← mfspr Rx, XUDBG1     #XUDBG1 data written to Rx on Ram execute
scm write RAMC ← 0x000X0000           #set Ram mode; Ram thread; Ram execute
scm write RAMI ← xori Rx, Rx, 0       #XUDBG1 on slowSPR bus; requires second Ram
                                       operation
scm write RAMC ← 0x000X0000           #set Ram mode; Ram thread; Ram execute
scm read RAMD: RAMD ← Rx              #XUDBG1 data valid in RAMD
    
```

#recover XUDBG1 data

```

scm write RAMI ← mfspr Rx, XUDBG2     #XUDBG2 data written to Rx on Ram execute
scm write RAMC ← 0x000X0000           #set Ram mode; Ram thread; Ram execute
scm write RAMI ← xori Rx, Rx, 0       #XUDBG2 on slowSPR bus; requires second Ram
                                       operation
scm write RAMC ← 0x000X0000           #set Ram mode; Ram thread; Ram execute
scm read RAMD: RAMD ← Rx              #XUDBG2 data valid in RAMD
    
```

10.11.2 Instruction Unit Debug Register 0 (IUDBG0)

Register Short Name:	IUDBG0	Read Access:	Hypv
Decimal SPR Number:	888	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:49	///	0x0	<u>Reserved</u>
50:51	WAY	0b00	<u>Instruction Cache Directory Way Select</u> Selects way for a instruction cache directory read
52:57	ROW	0x0	<u>Instruction Cache Directory Row Select</u> Selects row for a instruction cache directory read
58:61	///	0b0000	<u>Reserved</u>

Bit(s):	Field Name:	Init	Description
62	EXEC ^{NP}	0b0	<u>Instruction Cache Directory Read Execute</u> ^{NP} 1 Executes a instruction cache directory read
63	DONE	0b0	<u>Instruction Cache Directory Read Done</u> 1 Indicates a instruction cache directory read operation has completed and IUDBG1/IUDBG2 registers are valid

10.11.3 Instruction Unit Debug Register 1 (IUDBG1)

Register Short Name:	IUDBG1	Read Access:	Hypv
Decimal SPR Number:	889	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:52	///	0x0	<u>Reserved</u>
53:55	LRU	0b000	<u>Instruction Cache Directory LRU</u> Indicates value of the LRU in the instruction cache directory
56:59	PARITY	0b0000	<u>Instruction Cache Directory Parity</u> Indicates value of the parity bits in the instruction cache directory
60	ENDIAN	0b0	<u>Instruction Cache Directory Endian</u> 0 Big Endian 1 Little Endian
61:62	///	0b00	<u>Reserved</u>
63	VALID	0b0	<u>Instruction Cache Directory Read Valid</u> 0 directory entry is not valid 1 directory entry is valid

10.11.4 Instruction Unit Debug Register 2 (IUDBG2)

Register Short Name:	IUDBG2	Read Access:	Hypv
Decimal SPR Number:	890	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	///	0b00	<u>Reserved</u>
34:63	TAG	0x0	<u>Instruction Cache Directory Tag</u> Indicates value of the tag bit in the instruction cache directory

10.11.5 Execution Unit Debug Register 0 (XUDBG0)

Register Short Name:	XUDBG0	Read Access:	Hypv
Decimal SPR Number:	885	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:48	///	0x0	<u>Reserved</u>
49:51	WAY	0b000	<u>Data Cache Directory Way Select</u> Selects way for a data cache directory read
52:57	ROW	0x0	<u>Data Cache Directory Row Select</u> Selects row for a data cache directory read
58:61	///	0b0000	<u>Reserved</u>
62	EXEC ^{NP}	0b0	<u>Data Cache Directory Read Execute^{NP}</u> 1 Executes a data cache directory read
63	DONE	0b0	<u>Data Cache Directory Read Done</u> 1 Indicates a data cache directory read operation has completed and XUDBG1/XUDBG2 registers are valid

10.11.6 Execution Unit Debug Register 1 (XUDBG1)

Register Short Name:	XUDBG1	Read Access:	Hypv
Decimal SPR Number:	886	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:44	///	0x0	<u>Reserved</u>
45:48	WATCH	0b0000	<u>Data Cache Directory Watch Bits</u> 0 Directory entry has no watch set 1 Directory entry has watch set
49:55	LRU	0x0	<u>Data Cache Directory LRU</u> Indicates value of the LRU in the data cache directory
56:59	PARITY	0b0000	<u>Data Cache Directory Parity</u> Indicates value of the parity bits in the data cache directory
60:61	///	0b00	<u>Reserved</u>
62	LOCK	0b0	<u>Data Cache Directory Lock Bits</u> 0 Directory entry is unlocked 1 Directory entry is locked
63	VALID	0b0	<u>Data Cache Directory Read Valid</u> 0 directory entry is not valid 1 directory entry is valid

10.11.7 Execution Unit Debug Register 2 (XUDBG2)

Register Short Name:	XUDBG2	Read Access:	Hypv
Decimal SPR Number:	887	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	///	0b0	<u>Reserved</u>
33:63	TAG	0x0	<u>Data Cache Directory Tag</u> Indicates value of the tag bit in the data cache directory

10.12 Support for the Debugger Notify Halt (dnh) instruction

The Power ISA defines the **dnh** instruction, which is used by an external debugger to stop a thread. Additional implementation-dependent fields allow software to pass information to the debugger, or control other functions. This section describes how A2O supports this instruction through implementation specific registers and controls.

CCR4[EN_DNH] enables the function, and must be set in order for the **dnh** instruction to execute. Attempts to execute a **dnh** instruction when this bit is 0 will result in an illegal instruction exception.

10.12.1 DNH Data Register (DNHDR) Definition

Register Short Name:	DNHDR	Read Access:	Hypv
Decimal SPR Number:	855	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	dcfg

Bit(s):	Field Name:	Init	Description
32:47	///	0x0	<u>Reserved</u>
48:52	DUI	0x0	<u>Debug Halt Information</u> This data is captured from the DUI field when a dnh instruction is executed.
53	///	0b0	<u>Reserved</u>
54:63	DUIS	0x0	<u>Secondary Debug Halt Information</u> This data is captured from the DUIS field when a dnh instruction is executed.

The DNH Data Register (DNHDR) is a per-thread SPR that is loaded from both DUI and DUIS fields upon execution of the **dnh** instruction.

Execution of the **dnh** instruction halts the corresponding thread by activating the THRCTL[Tx_Stop] bit. A debugger can poll the THRCTL register to determine which threads are stopped, along with status bits that indicate why. See *Thread control and status* on page 422.

The DUI and DUIS fields are passed to the debugger through the DNHDR SPR. In addition, the DNHDR outputs corresponding to DUIS bits will be reserved for specific hardware functions. The current plan for using the DUIS field is shown below:

- DUIS(0:5) - Routed to core output pins. Allows implementation of chip-specific debug functions.
- DUIS(6:7) - Reserved for additional on-core debug functions.
- DUIS(8:9) - These bits are encoded, and are used to execute a subset of the EDM debug actions. (Refer to *External Debug Mode* in Section 10.8.1 on page 420)
 - 00 - No action.
 - 01 - Stop all threads.
 - 10 - Activate external signal (ac_an_debug_trigger). This is a single cycle pulse.
 - 11 - Activate external signal and stop all threads.

10.13 Trace and Trigger Bus

An 64-bit debug bus is brought out of the core for use by a trace array or other debug functions implemented at the chip level. A 12-bit bus providing trace trigger information is also made available. Together, the trace and trigger signals provide debug data and the control signals used for starting and stopping the trace array.

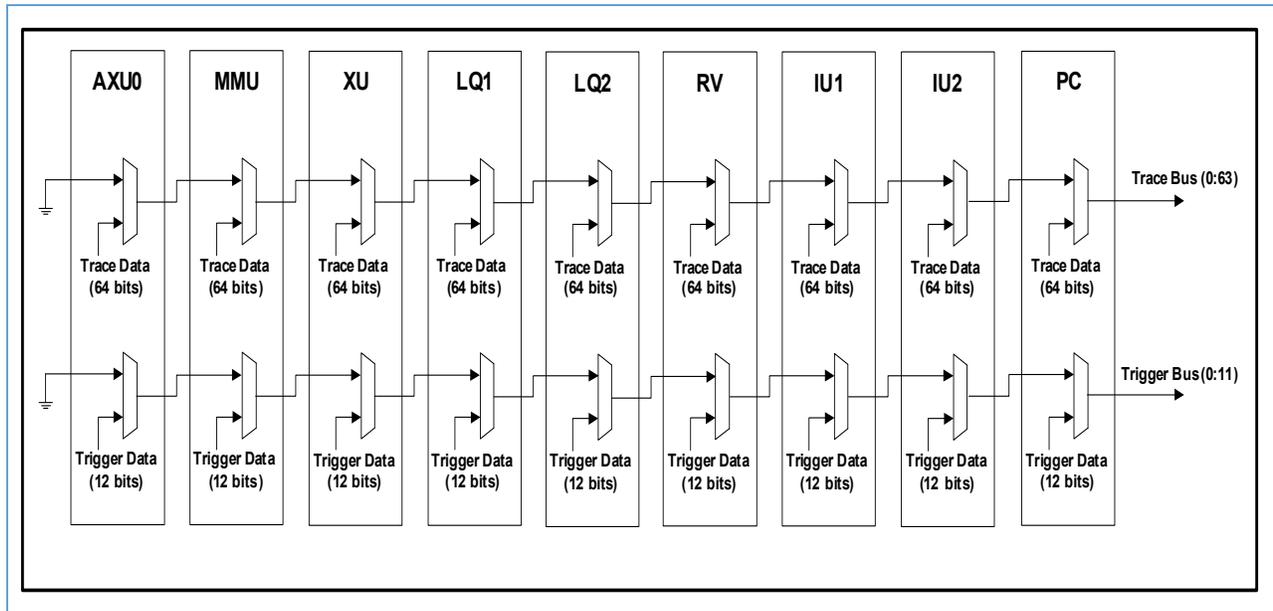
Each unit has one or more debug multiplexer components, with each multiplexer requiring 16 bits of control for selecting the debug and trigger groups from among its signals. A single 32-bit SCOM register contains two sets of debug multiplexer control bits for controlling multiplexers in different units or within the same unit. Tables describing each unit's debug select register and corresponding debug and trigger groups are shown in *Section C Debug and Trigger Groups* on page 773.

10.13.1 Trace and Trigger Bus Overview

Each core unit shares a pass-through trace bus. As depicted in *Figure 10-1*, the order of trace bus data flow from unit to unit is: AXU0 => MMU => XU => LQ => RV => IU=> PC. Most units implement one debug mux component; the IU and LQ implement two. Each unit has an input for the trace bus from the previous unit, and a pass-through multiplexer (trace bus on-ramp) to control sending the trace bus of the previous unit or its local trace signals onto the output trace bus. The pass-through multiplexer control for the data is managed as four, 16-bit groups. The trigger bus structure is similar to the trace bus, and maintains the same unit-to-unit data flow. Each unit can contribute up to 12 bits of trigger data onto the trigger bus. Control bits select between the trigger bus of the previous unit, or a unit's local trigger signals as two, 6-bit groups.

On power-up, the debug mux select registers are initialized to have the pass-through state selected for the trace and trigger buses by default. Also, PCCR0[EnabDebugMode] initializes with debug mode inactive, which holds the trace and trigger bus latches in a non-clocked state. Setting PCCR0[EnabDebugMode] enables the trace and trigger buses for the whole core.

Figure 3. Pass-Through Trace and Trigger Bus Overview

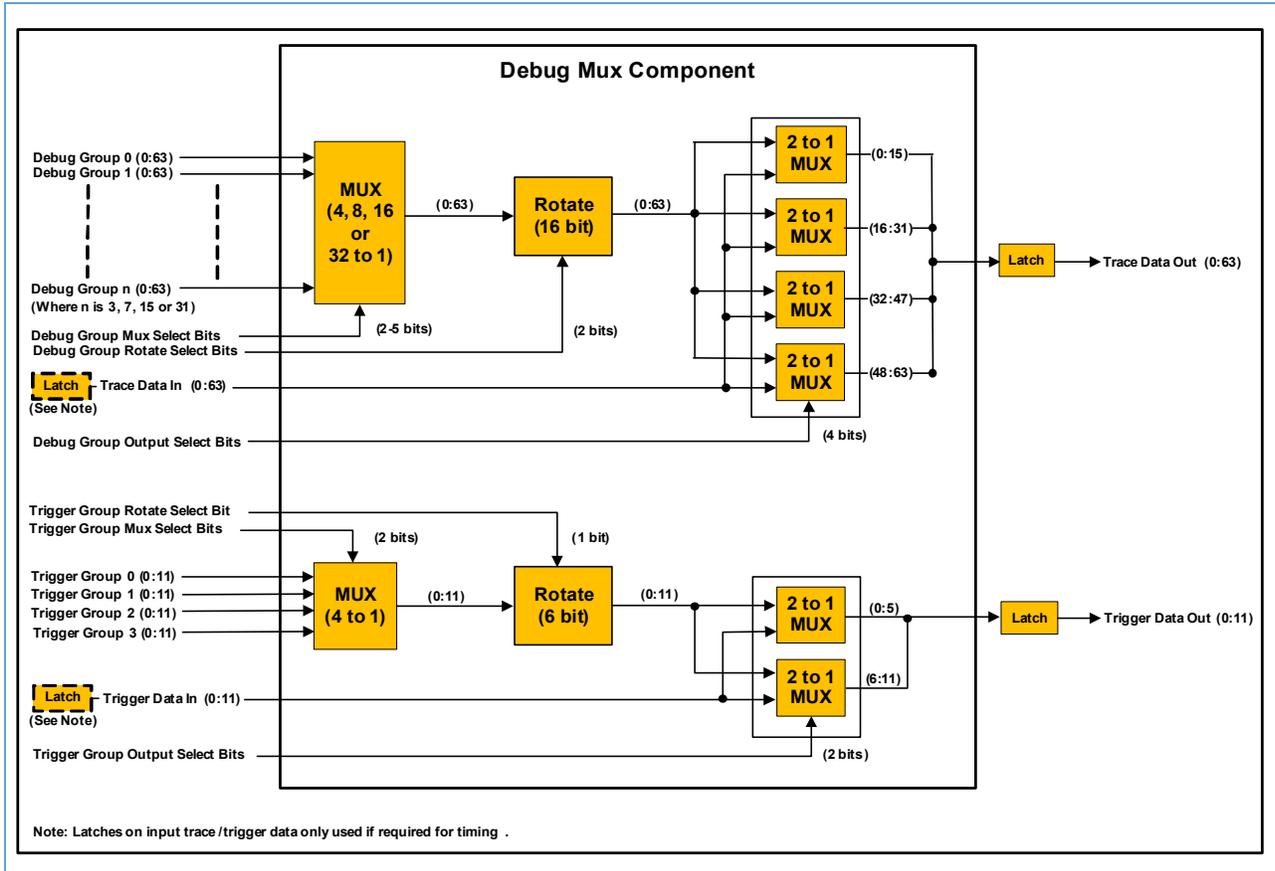


10.13.2 Unit Level Trace and Trigger Bus Implementation

This section describes a unit level implementation of the trace and trigger pass-through bus. *Figure 10-2* shows an implementation that provides selection between multiple 64 bit debug groups. A set of four standard debug_mux components allow selection of 4:1, 8:1, 16:1 or 32:1 multiplexers for the 64-bit debug group signals. After the debug multiplexer, the 64 bits of unit level signals can be rotated as 16-bit groups, before being multiplexed with the input trace bus. Four 2:1 multiplexers select between the unit's debug signals, or the previous unit's input trace bus on 16-bit boundaries. The trigger bus is implemented with a fixed 4:1 multiplexer to select between four 12-bit trigger groups. The output of the trigger mux can be rotated as 6-bit groups. The unit's local trigger signals then connect to a pair of 2:1 multiplexers, which choose between the unit's trigger signals, or signals from the previous unit's input trigger bus.

The trace and trigger signals from each debug multiplexer output are latched before connection to the next downstream debug multiplexer component. Latches on the input trace and trigger data will be implemented only if required for timing. Each debug multiplexer requires between 13 (4:1 debug multiplexer) and 16 (32:1 debug multiplexer) control bits, which are connected to a SCOM-accessible debug select register.

Figure 4. Unit Trace and Trigger Bus, and debug_mux Component Description



10.13.3 Debug Select Registers

Each 32-bit SCOM accessible debug select register can control up to two debug multiplexer components. The debug multiplexer controls for core units are split up as follows:

- **ASR** AXU0 debug multiplexer 1 and RV debug multiplexer 1
- **ISR** IU debug multiplexer 1 and IU debug multiplexer 2
- **LDSR** LQ debug multiplexer 1 and LQ debug multiplexer 2
- **MDSR** MMU debug multiplexer 1 and PC debug multiplexer 1
- **XDSR** XU debug multiplexer 1

Descriptions of the debug select registers, and the debug and trigger groups connected to each debug multiplexer component are provided in *Section C Debug and Trigger Groups* on page 773.



11. Performance Events and Event Selection

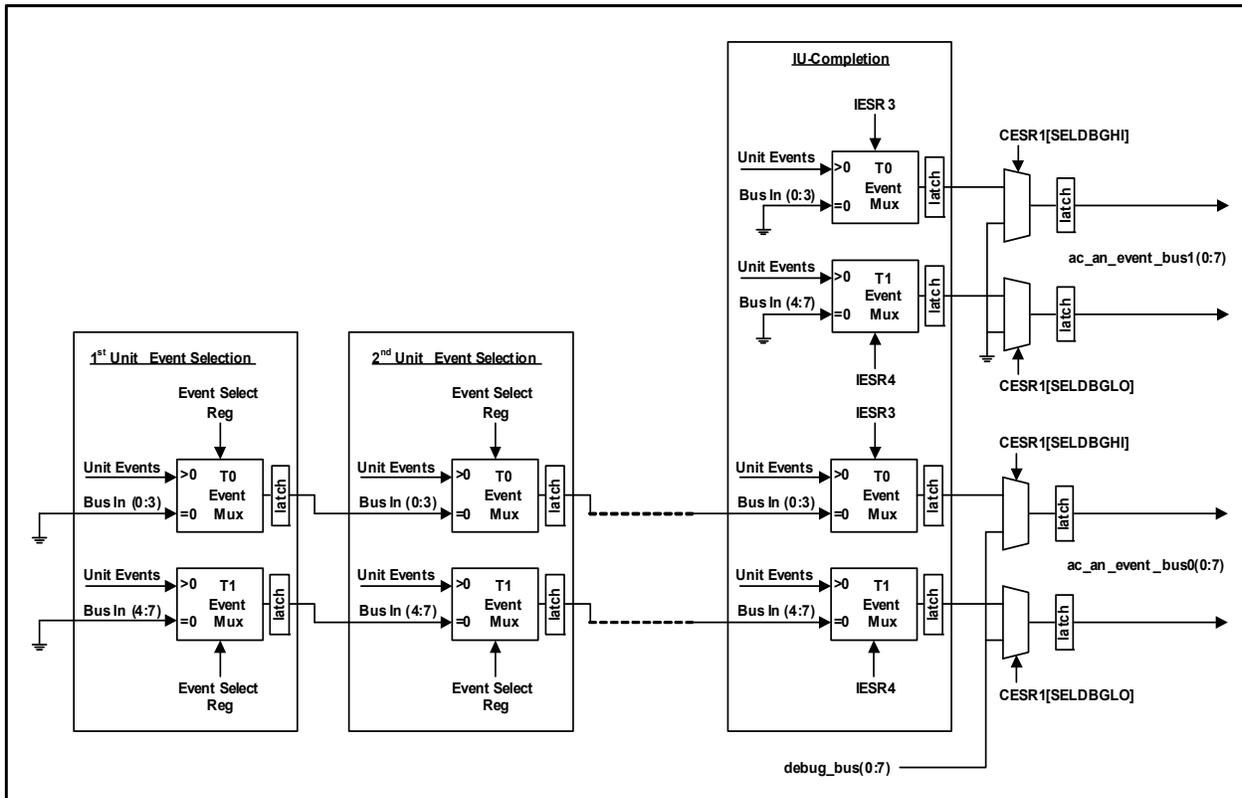
11.1 Event Bus Overview

Two 8-bit event buses, *ac_an_event_bus0* and *ac_an_event_bus1*, are brought out of the core for use by an external performance monitor unit. This section provides an overview of the event bus components and structure, and how performance event signals are selected.

Within the core, the performance event bus is implemented in a pass-through design (refer to *Figure 11-1*). Units implement event multiplexer components (*tri_event_mux1t*) which select between a unit's internal signals, or performance event bits from the upstream event bus. Since each multiplexer component supports four bits, units implement one per thread. The multiplexer component connected to bits 0 through 3 of the event bus selects thread 0 events, while the multiplexer connected to bits 4 through 7 selects events on thread 1.

Completion is the last unit to select events prior to the event bus exiting the core. Both speculative and completion events selected by the multiplexer are continuously driven onto *ac_an_event_bus0*. When two instructions complete on the same cycle, selected events active in the I1 completion unit are driven out onto *ac_an_event_bus1*.

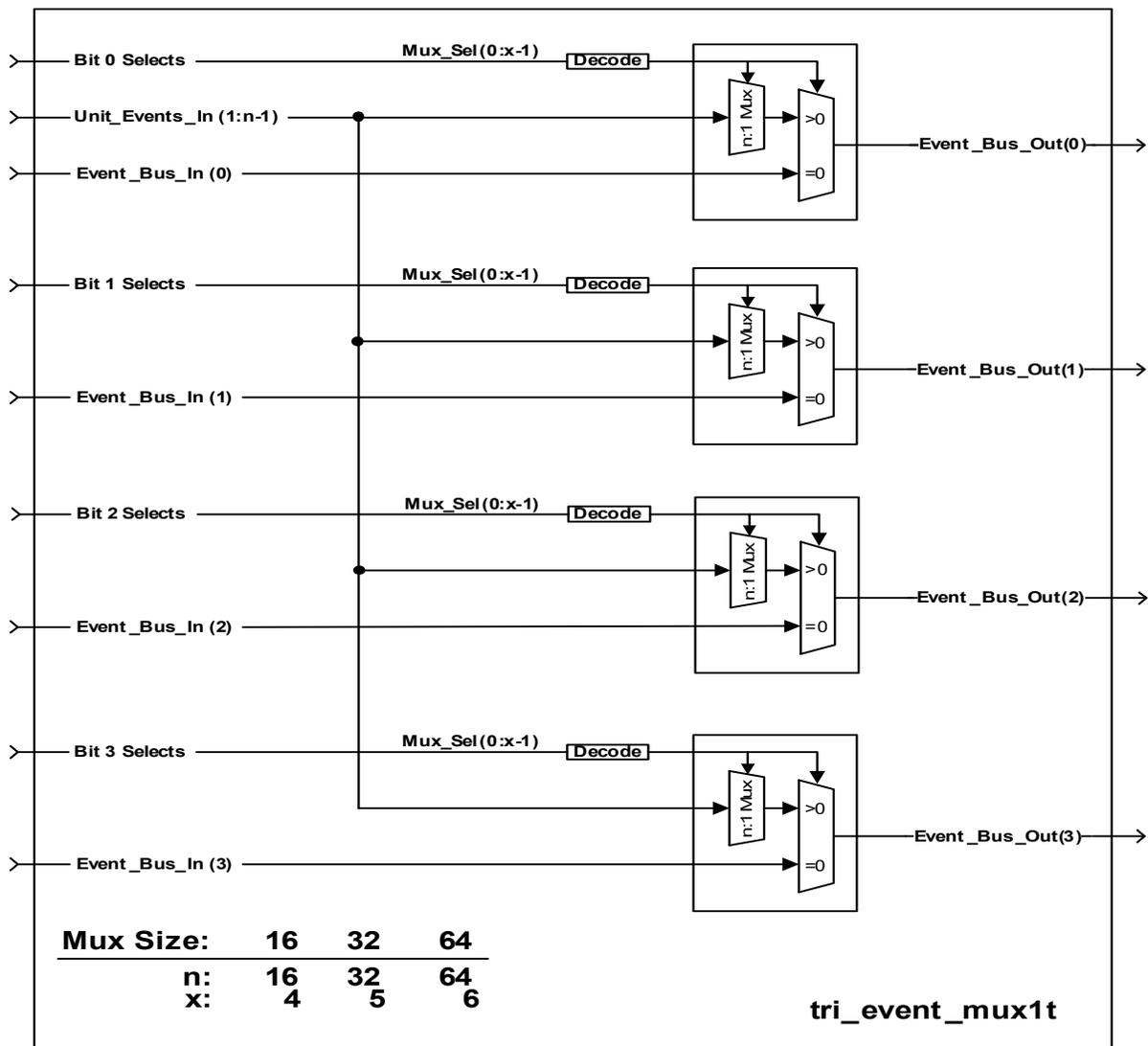
Figure 3. Performance event selection overview



The event multiplexer component is configurable to support 16, 32 or 64 performance event signals from the unit, and drive out the selected events on its four output bits. If the incoming select bits decode to 0 for an output bit, the signal from the corresponding input event bus bit is passed through to that output. Any select bit decode greater than zero selects a signal from the unit's internal performance events for the output. For additional information on how performance event components are used by each unit refer to *Figure 11-2 A2 unit event multiplexer component* and *Table 11-1 Unit Event Mux Summary* below.

Each event multiplexer component uses one or two SPRs, called *Event Select Registers*, to provide the decode values that select performance event signals for each output bit. The 32 and 64 event configurations require two event select registers; only one register is required for the 16 event configuration. For each performance event bus bit, only one unit event multiplexer can have a non-zero decode value. To avoid selecting multiple events and potentially blocking the desired performance event signal, all other event select register values for that event bus bit must be set to zero.

Figure 4. A2O unit event multiplexer component



As an additional debug feature, any signals that can be multiplexed onto `ac_an_debug_bus(0:7)` can be forwarded onto the external event bus, and counted by performance monitor counters. When `CESR1[SELD-BGHI]` is active, `ac_an_debug_bus(0:3)` will be driven onto `ac_an_event_bus0(0:3)` in place of the performance event signals. In a similar manner `CESR1[SELDBGLO]` selects `ac_an_debug_bus(4:7)` in place of the corresponding performance event bits. When `ac_an_event_bus0` is overridden in this way, signals on `ac_an_event_bus1` are forced inactive. Refer to *Section 10.12* on page 399 for information on the Trace Bus.

Table 1. Unit Event Mux Summary

Unit	T0 Event Mux Size and Event Select Register	T1 Event Mux Size and Event Select Register	Comments
AXU0	16, A0ESR	16, A0ESR	
IU	64, IESR1 32, IESR3	64, IESR2 32, IESR4	IU events Completion events
LSU	64, LESR1 16, PESR	64, LESR2 16, PESR	LSU events Prefetcher events
MMU	64, MESR1	64, MESR2	
RV	32, RESR1	32, RESR2	
XU	16, XESR1 16, XESR2	16, XESR1 16, XESR2	XU events Branch events

Refer to *Section 11.3.3* beginning on page 447 for tables describing each unit's performance events, and to *Section 11.4* beginning on page 461 for the event select register tables.

11.2 A20 Performance Event Controls

This section describes different performance analysis modes of operation, and the CESR1 bits used to control them.

11.2.1 Enabling performance event and trace bus latches

The default core power-on reset state disables clocking of latches used for performance event and debug bus logic. Several bits of CESR1 need to be set prior to working with these functions in order to enable them.

Setting `CESR1[ENABPERF]` is required in order to enable latches associated with performance event signals, event multiplexer staging and the core event bus outputs. *This bit must be set prior to working with any unit performance events*, so that the event related latches are enabled.

In addition, if the trace-trigger bus logic is required, then `CESR1[ENABTRACEBUS]` should be set in order to enable debug-bus-related latches.

Note: `CESR1[ENABTRACEBUS]` performs a similar function to `PCCR0[EnabDebugMode]`, except that it only enables trace-trigger-bus-related latches without activating any additional debug mode controls.

11.2.2 Performance analysis operating modes

Each unit enables performance monitor event signals based on three count mode bits selected by the CESR1[COUNTMODES] field. If a thread's mode of operation matches a selected count mode, then its performance events will be enabled for counting. The three modes and their respective CESR1 bits are shown below:

- CESR1(33) - Count events in problem mode
- CESR1(34) - Count events in guest supervisor mode
- CESR1(35) - Count events in hypervisor mode

Setting CESR1[INSTTRACE] places the core in instruction trace mode. In this mode, the IU, XU and LSU continuously output relevant instruction data onto the trace bus, where it is written to system memory for post-processing and analysis. CESR1[INSTTRACETID] determines which thread is selected for core instruction tracing. See *A2 support for core instruction trace* on page 468.

Note: Support for the instruction trace function is TBD.

Two per-thread control bits, CESR1[PMAE_Tx, PMAO_Tx], provide control and status for instruction sampling operations. See *A2 support for instruction sampling* on page 471.

11.2.3 Options for selecting debug bus signals

As shown in *Figure 11-1* on page 441, signals from the debug bus can be multiplexed onto the external event bus instead of performance events. This provides a way to use performance monitors to count signal activity for debug purposes. Any debug signals driven onto ac_an_debug_bus(0:7) can be selected in place of the corresponding core event multiplexer outputs.

When active, CESR1[SELDBGHI] selects signals from ac_an_debug_bus(0:3) in place of the corresponding performance event signals, and drives them onto ac_an_event_bus0(0:3). In the same way CESR1[SELDBGLO] can be used to output ac_an_debug_bus(4:7) onto ac_an_event_bus0(4:7). When ac_an_event_bus0 is overridden in this way, signals on ac_an_event_bus1 are forced inactive.

11.2.4 Core Event Select Register

Register Short Name:	CESR1	Read Access:	Priv
Decimal SPR Number:	912	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Slow	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	ENABPERF	0b0	<u>Enable Performance Event Latches</u> When set, latches used to redrive performance event signals and the event bus are enabled. This bit must be set prior to making performance measurements.
33:35	COUNTMODES	0b000	<u>Performance Event Count Modes</u> This field determines which count modes are valid for the selected performance events. More than one count mode bit at a time may be enabled. 33 = count events when in Problem mode. 34 = count events when in Guest Supervisor mode. 35 = count events when in Hypervisor mode.

Bit(s):	Field Name:	Init	Description
36	ENABTRACEBUS	0b0	<u>Enable Trace-Trigger Bus Latches</u> When set, latches used to redrive the Trace-Trigger bus and trace related signals are enabled. This bit is an alternate method of enabling the Trace-Trigger bus, similar to PCCR0[Enable Debug Mode]. Unlike PCCR0[Enable Debug Mode] it does not enable additional debug mode functions in the THRCTL or PCCR0 registers.
37	///	0b0	<u>Reserved</u>
38	SELDBGHI	0b0	<u>Select Trace Bits on Event Bus (0:3)</u> Selects ac_an_debug_bus(0:3) to drive out on ac_an_event_bus0(0:3) in place of performance event signals. Any debug signal muxed onto these bits can be counted by the PMU.
39	SELDBGLO	0b0	<u>Select Trace Bits on Event Bus (4:7)</u> Selects ac_an_debug_bus(4:7) to drive out on ac_an_event_bus0(4:7) in place of performance event signals. Any debug signal muxed onto these bits can be counted by the PMU.
40	INSTTRACE	0b0	<u>Instruction Trace Mode Enable</u> This bit activates mux selects and controls used to perform the instruction trace function. Note: Support for the instruction trace function is TBD.
41	INSTTRACETID	0b0	<u>Instruction Trace Mode Thread ID</u> Indicates which thread is selected for instruction tracing. T0='0', T1='1' Note: Support for the instruction trace function is TBD.
42:43	///	0b00	<u>Reserved</u>
44	PMAE_T0	0b0	<u>Performance Monitor Alert Enable, T0</u> 0 Performance Monitor Alerts on thread 0 are disabled. 1 Performance Monitor Alerts on thread 0 are enabled. Software can set or clear this bit. Hardware will clear this bit upon activation of a performance monitor alert.
45	PMAO_T0	0b0	<u>Performance Monitor Alert Occurred, T0</u> 0 A Performance Monitor Alert has not occurred on thread 0 since this bit was last cleared. 1 A Performance Monitor Alert has occurred on thread 0 since this bit was last cleared. Software can set or clear this bit. Hardware will set this bit upon activation of a performance monitor alert.
46	PMAE_T1	0b0	<u>Performance Monitor Alert Enable, T1</u> 0 Performance Monitor Alerts on thread 1 are disabled. 1 Performance Monitor Alerts on thread 1 are enabled. Software can set or clear this bit. Hardware will clear this bit upon activation of a performance monitor alert.
47	PMAO_T1	0b0	<u>Performance Monitor Alert Occurred, T1</u> 0 A Performance Monitor Alert has not occurred on thread 1 since this bit was last cleared. 1 A Performance Monitor Alert has occurred on thread 1 since this bit was last cleared. Software can set or clear this bit. Hardware will set this bit upon activation of a performance monitor alert.
48:63	///	0x0	<u>Reserved</u>

11.3 Unit Performance Events

11.3.1 Performance Monitor Event Tags and Count Modes

The event tags and count modes are summarized in *Table 11-2*. *Cycle Counting* refers to counting the number of cycles a performance monitor signal is active or inactive. *Event Counting* refers to counting the number of occurrences of an event.

In the following sections, performance event tables for each unit are included which describe each event and how they are selected by the unit's event multiplexer select register. The performance monitor event tags are shown in the first column; a tag (B, C, E, S, or V) is used to specify how the event should be counted.

Table 2. Performance monitor event tags

Tag	Definition	Cycle Counting	Event Counting
B	Signals are useful for counting both cycles and events. Since an edge detector is used for event counting, signals of this type must have events separated by at least one cycle.	Cycle	Edge
C	Signals are only useful for counting cycles. Events may not be counted because these signals may have multi-cycle events that occur on consecutive cycles.	Cycle	n/a
E	Signals are only useful for counting events. The cycles that this signal is active may not accurately represent the cycles that the associated function is actually occurring.	n/a	Edge
S	Signals are single cycle signals that represent single cycle events. Since cycles and events are synonymous, counting cycles is sufficient to determine the number of cycles or events.	Cycle	n/a
V	Signals are single cycle events but they may occur on consecutive cycles. Thus, in order to count the number of events the number of cycles the signal is active must be used.	n/a	Cycle

11.3.2 Performance event table description

The layout and information in the unit performance events tables is described below:

- Column 1 includes two lines: the performance event name and the associated tag values explained in *Table 11-2*
- Column 2 describes the performance event
- Column 3 indicates if the event is speculative or non-speculative.
- Column 4 lists the multiplexer decode value required to select the thread 0 performance event and drive it out on event bus bits 0 through 3. The unit's event select registers control which performance event is placed onto a particular event bus bit. A decode of 0 always selects the performance event signal driven into the mux from the previous unit.
- Column 5 lists the multiplexer decode value required to select the thread 1 performance event and drive it out on event bus bits 4 through 7. The unit's event select registers control which performance event is placed onto a particular event bus bit. A decode of 0 always selects the performance event signal driven into the mux from the previous unit..

11.3.3 Unit performance event tables

11.3.3.1 AXU0 performance events table

Table 3. AXU0 Performance Events Table (Use A0ESR for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Speculative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
AXU Instruction Commit Tag: S	A valid AXU (non-load/store) instruction is in EX6, past the last flush point.			
AXU CR Commit Tag: S	A valid AXU CR updater instruction is in EX6, past the last flush point.			
AXU Idle Tag: S	No valid AXU instruction is in the EX6 stage.			
FP Div/Sqrt In Progress Tag: B	A Floating-Point Divide or Square Root sequence is in progress. Also includes single-precision versions.			
Denormal Operand Flush Tag: S	A B operand of a Floating Point instruction caused a Denormal Operand flush2ucode. Microcode prenormalization sequence will follow.			
AXU uCode Instr Commit Tag: S	A valid AXU instruction from a ucode sequence is in EX6, past the last flush point. The last instruction of the sequence is not counted.			
FP Exception Tag: E	FX bit of the FPSCR			
FP Enabled Exception Tag: E	FEX bit of the FPSCR			

11.3.3.2 IU performance events table

Table 4. IU Performance Events Table (Use IESR1 and IESR2 for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Speculative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
IL1 Miss Cycles Tag: C	Number of cycles a thread is waiting for a reload from the L2. -Not when CI=1. -Not when thread held off for a reload that another thread is waiting for. -Still counts even if flush has occurred.			
IL1 Reloads Dropped Tag: E	Number of times a reload from the L2 is dropped, per thread -Not when CI=1 -Does not count when not loading cache due to a back invalidate to that address			

Table 4. IU Performance Events Table (Use IESR1 and IESR2 for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Specu- lative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
Reload Collisions Tag: C	Number of cycles a ready thread is held off due to the L1 Cache being reloaded -Could occur on multiple threads per cycle			
IU0 Redirected Tag: C	Number of cycles IU0 is flushed for any reason (CP, UC, BP, etc.)			
BP IU2 Redirect	Number of times Branch Predict / BTB Redirect occurred in IU2			
BP IU3 Redirect	Number of times Branch Predict Redirect occurred in IU3			
BP IU4 Redirect	Number of times Branch Predict Redirect occurred in IU4			
UC Full Flush	Number of times Microcode flushed due to its buffers being full.			
Prefetch	Number of times prefetch occurred.			
IERAT Miss Tag: B	Number of times IERAT Miss occurs -Can only occur on one thread per cycle			
ICache Fetch Tag: B	Number of times ICache read completes for instruction -Does not count if flushed before IU2 -Counts whether cache hit or miss -Can only occur on one thread per cycle			
Instructions Fetched Tag: B	Number of instructions fetched, divided by 4 (only counts every 4 instructions) -Uses a counter so fetches of 1, 2, or 3 instructions are not lost -Includes CI=0 or 1, hit or miss (any instruction that comes through IU2)			
reserved				
L2 Back Invalidates Tag: B	Back invalidate from L2 -Per core, not per thread			
L2 Back Invalidates - Hits Tag: B	Back invalidate from L2, and data was contained within the instruction cache. -Per core, not per thread -Does not count if hits cacheline for which we are waiting for a reload			
BHT overrides BTB Tag: C	BHT predictor overrides and corrects BTB predictor			
IBuff Empty Tag: C	Instruction buffers are empty			
IU5 Stall Tag: C	Any IU5 Stall			
IU6 Stall Tag: C	Any IU6 Stall			
IU6 Dispatch Fx0 Tag: C	Instruction dispatched to RV - FX0 unit divided by 2 (only counts every 2 instructions) -Uses a counter so every other fetch is not lost	Y		
IU6 Dispatch Fx1 Tag: C	Instruction dispatched to RV - FX1 unit divided by 2 (only counts every 2 instructions) -Uses a counter so every other fetch is not lost	Y		

Table 4. IU Performance Events Table (Use IESR1 and IESR2 for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Speculative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
IU6 Dispatch LQ Tag: C	Instruction dispatched to RV - LQ unit divided by 2 (only counts every 2 instructions) -Uses a counter so every other fetch is not lost	Y		
IU6 Dispatch Axu0 Tag: C	Instruction dispatched to RV - Axu0 unit divided by 2 (only counts every 2 instructions) -Uses a counter so every other fetch is not lost	Y		
IU6 Dispatch Axu1 Tag: C	Instruction dispatched to RV - Axu1 unit divided by 2 (only counts every 2 instructions) -Uses a counter so every other fetch is not lost	Y		
IU5 CPL credit stall Tag: C	Rename stalled due to no CPL credit	Y		
IU5 GPR credit stall Tag: C	Rename stalled due to no GPR credit	Y		
IU5 CR credit stall Tag: C	Rename stalled due to no CR credit	Y		
IU5 LR credit stall Tag: C	Rename stalled due to no LR credit	Y		
IU5 CTR credit stall Tag: C	Rename stalled due to no CTR credit	Y		
IU5 XER credit stall Tag: C	Rename stalled due to no XER credit	Y		
IU5 BR hold stall	Rename stalled due to branch mispredict till completion empties	Y		
IU5 AXU hold stall	Rename stalled due to AXU unit	Y		
IU6 Fx0 credit stall Tag: C	Rename stalled due to no Fx0_credit (thread credit or total credit)	Y		
IU6 Fx1 credit stall Tag: C	Rename stalled due to no Fx1_credit (thread credit or total credit)	Y		
IU6 lq_cmdq credit stall Tag: C	Rename stalled due to no LQ_cmdq credit (thread credit or total credit)	Y		
IU6 sq_cmdq credit stall Tag: C	Rename stalled due to no SQ_cmdq credit (thread credit or total credit)	Y		
IU6 Axu0 credit stall Tag: C	Rename stalled due to no Axu0 credit (thread credit or total credit)	Y		
IU6 Axu1 credit stall Tag: C	Rename stalled due to no Axu1 credit (thread credit or total credit)	Y		

Table 5. IU Completion Performance Events Table (Use IESR3 corresponding mux selects)**Note:** Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Specu- lative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
PPC Commit (C)	Number of instructions committed. uCode sequences count as one instruction.		1	1
uCode Commit (S)	Number of uCode sequences committed.		2	2
Any Flush (C)	Number of cycles flush is asserted to the IU		3	3
Opcode Match (C)	Number of opcode matches		4	4
External Interrupt Pending (C)	Count number of cycles the interrupt signal into the processor is asserted before the completion logic redirects program flow to the interrupt vector		5	5
Critical External Interrupt Pending (C)	Count number of cycles the interrupt signal into the processor is asserted before the completion logic redirects program flow to the interrupt vector		6	6
Performance Monitor Inter- rupt Pending (C)	Count number of cycles the interrupt signal into the processor is asserted before the completion logic redirects program flow to the interrupt vector		7	7
External Interrupt Taken (S)	Number of interrupts taken		8	8
Critical External Interrupt Taken (S)	Number of interrupts taken		9	9
Performance Monitor Inter- rupt Taken (S)	Number of interrupts taken		10	10
FX1 Completion	Number of PPC instructions completed on FX1		11	11
LQ0 Completion	Number of PPC instructions completed on LQ0 Completion Bus		12	12

11.3.3.3 LSU performance events table**Table 6. LQ Performance Events Table** (Use LESR1 and LESR2 for corresponding mux selects)**Note:** Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Specu- lative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
Committed Stores Tag: V	Number of completed store commands. -Microcoded instructions will count more than once. -Does not count syncs,tlb ops,dcbz,icswx, or data cache management instructions. -Includes stcx, but does not wait for stcx complete response from the L2. -Includes cache-inhibited stores.			

Table 6. LQ Performance Events Table (Use LESR1 and LESR2 for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Speculative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
Committed Store Misses Tag: V	Number of completed store commands that missed the L1 Data Cache. -Microcoded instructions may be counted more than once. -Does not count syncs,tlb ops,dcbz,icswx, or data cache management instructions. -Includes stcx, but does not wait for stcx complete response from the L2. -Does not includes cache-inhibited stores.			
Committed Load Misses Tag: V	Number of completed load commands that missed the L1 Data Cache. -Microcoded instructions may be counted more than once. -Does not count dcbt[st][ls][ep]. -Include larx. -Does not includes cache-inhibited loads.			
Committed Cache-Inhibited Load Misses Tag: V	Number of completed cache-inhibited load commands. -Microcoded instructions may be counted more than once. -Does not count dcbt[st][ls][ep]. -Does not includes cacheable loads.			
Committed Cacheable Loads Tag: V	Number of completed cacheable load commands. -Microcoded instructions may be counted more than once. -Does not count dcbt[st][ls][ep]. -Include larx. -Does not includes cache-inhibited loads.			
Committed DCBT Misses Tag: V	Number of completed dcbt[st][ls][ep] commands that missed the L1 Data Cache. -Does not include touch ops that were dropped due to the following: 1) Unsupported TH(CT) fields. 2) Translated to cache-inhibited. 3) Exception detected on dcbt[st][ep].			
Committed DCBT Hits Tag: V	Number of completed dcbt[st][ls][ep] commands that hit the L1 Data Cache. -Does not include touch ops that were dropped due to the following: 1) Unsupported TH(CT) fields. 2) Translated to cache-inhibited. 3) Exception detected on dcbt[st][ep].			
Committed AXU Loads Tag: V	Number of completed AXU loads. AXU refers to the unit attached on the AXU interface (i.e a floating point unit). -Cacheable and cache-inhibited loads are counted.			
Committed AXU Stores Tag: V	Number of completed AXU stores. AXU refers to the unit attached on the AXU interface (i.e a floating point unit). -Cacheable and cache-inhibited stores are counted.			
Committed STCX Tag: V	Number of completed STCX instructions. Does not wait for the stcx complete response from the L2.			
Committed WCLR Tag: V	Number of completed WCLR instructions.			
Committed WCLR L=0x1 Tag: V	Number of completed WCLR instructions that set the Watchlost indicator.			

Table 6. LQ Performance Events Table (Use LESR1 and LESR2 for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Specu- lative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
Committed LDAWX Tag: V	Number of completed LDAWX. instructions.			
Unsupported Alignment Flush Tag: V	Number of flushes due to an unsupported alignment. -This is a speculative count. -Includes speculative flushes to microcode. -Includes speculative flushes to the alignment interrupt due to unaligned larx, stcx, icswx, ldawx or XUCR0[FLSTA]=1 or XUCR0[AFLSTA]=1.			
Reload Resource Conflict Restart Tag: V	Number of restarts due to a resource conflict on a reload. 1) 1st half of Cacheable Reload colliding with dcbt[st]ls or ldawx.			
Store Queue Resource Con- flict Restart Tag: V	Number of restarts due to Store Commit colliding with Load 1) Store Queue Requested an RV Issue Hole			
STQ: Store Queue Commit Attempt Tag: V	Number of attempted Store Commits 1) Store Commit collided with Load Issue			
STQ: Store Queue Commit Tag: V	Number of Store Commits.			
STQ: Committed Load for- ward from Store Queue Tag: V	Number of Loads that forwarded from the Store Queue			
STQ: Younger Load Colliding Against Multiple Older Stores Tag: V	Number of younger loads restarting due to colliding against multiple older stores 1) Speculative Count			
STQ: Younger Load Restarted Due to Older Store Tag: V	Number of younger guarded loads restarting due to colliding against an older guarded store 1) Speculative Count 2) Younger Guarded Load Request collided against an older guarded Store 3) Younger Load Request hit against an older CP_NEXT store instruction (i.e icbi, sync, stcx, icswx., mftgpr, mfdp) 4) Younger Load Request Address hit multiple older entries 5) Younger Load Request Address hit against an older store but endian- ness differs 6) Younger Guarded Load Request Address hit against an older store 7) Younger Load Request Address hit against an older store type with no data associated 8) Younger Loadmiss Request Cacheline Address hit against older store type			
DERAT: Non-Speculative Erat Misses Sent to MMU Tag: V	Number of Non-Speculative ERAT Misses sent to the MMU 1) Speculative Count			
DERAT: Total Erat Misses Sent to the MMU Tag: V	Number of ERAT Misses sent to the MMU 1) Speculative Count			

Table 6. LQ Performance Events Table (Use LESR1 and LESR2 for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Speculative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
DERAT: Total ERAT Translation Attempts Tag: V	Number of Data ERAT translation attempts. These include instruction stream attempts and prefetcher attempts 1) Speculative Count			
ODQ: Committed Younger Loadhit collided with Older Store Flush Tag: V	Number of Loadhits flushed due to older stores colliding			
ODQ: Committed Back-Invalidate-hit-Loadhit Flushed Tag: V	Number of Back-Invalidate hits against a loadhit that caused a flush.			
ODQ: Committed Older I=1 Load hit Younger I=1 Load Flush Tag: V	Number of Cache Inhibited Older Load hit against a Cache Inhibited Younger Load			
ODQ: Committed Older Loadmiss hit against Younger Load Forward Tag: V	Number of Older Loadmisses hitting against a Younger Load Forward			
LDQ: Committed Back-Invalidate-hit-Loadmiss Flushed Tag: V	Number of Back-Invalidate hits against a loadmiss that caused a flush.			
LDQ: Committed Load was Gathered Tag: V	Number of Loadmisses that were handled by the Load Gather Queue			
LDQ: Load Gather Queue Full Restart Tag: B	Number of restart due to not able to gather and outstanding loadmiss to the same cacheline. 1) Speculative Count			
LDQ: Reload Update Attempts Tag: V	Number of Reload update attempts to the L1 Data Cache			
LDQ: Reload Update Hole Request Tag: V	Number of Reload Queue Requests an RV Issue Hole.			
CTL: Speculative Flushes Tag: V	Number of Speculative Flushes in the LSU due to a loadmiss.			
CTL: Prefetching Issued Tag: V	Number of Issued Prefetches.			
CTL: Prefetch Dropped Tag: V	Number of Issued Prefetches that were dropped. Prefetches are dropped due to the following reasons: 1) Loadmiss Queue Full 2) Load Type request to Cacheline Real Address is already outstanding 3) Store Type requests to Cacheline Real Address are outstanding in STQ			

Table 6. LQ Performance Events Table (Use LESR1 and LESR2 for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Specu- lative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
CTL: Prefetch Hit L1 Data Cache Tag: V	Number of Issued Prefetches that hit the L1 Data Cache. These requests are considered complete and wont be sent to the L2			
CTL: Prefetch Missed Data ERAT Tag: V	Number of Issued Prefetches that missd the Data ERAT.			
Committed Duplicate LDAWX. Tag: V	Number of completed LDAWX. which sets CR=001 XER[SO].			
Inter-Thread Directory Access Flush Tag: V	Number of flushes due to a thread setting/clearing cacheline directory contents (i.e. valid,lock,thread watch bits) and different thread accesses same cacheline. Also, count of non-committed WCLR L[0]=0 in pipe and different thread has a directory access in EX3. -This is a speculative count.			
Committed WCHKALL Tag: V	Number of completed WCHKALL instructions.			
Committed Successful WCH- KALL Tag: V	Number of completed WCHKALL instructions that returned CR=000 XER[SO].			
Load Miss Queue Full Restart Tag: V	Number of restart due to the Load Miss Queue being full. Load Miss Queue Full is determined when all 8 entries are in use and new load miss is flushed. Also, count of load miss command sequence wrapped flushes. -This is a speculative count.			
Hit Against Outstanding Load Restart Tag: V	Number of restarts due to a cache instruction (i.e load,store, or cache management) hit against an outstanding load miss. -XUCR0[CLS]=0Cacheline check is down to the 64Byte boundary, else check is down to the 128Byte boundary. -This is a speculative count.			
Hit Against Outstanding I=G=1 Request Restart Tag: V	Number of restarts due to a cache instruction (i.e load,store, or cache management) hit against an outstanding guarded cache-inhibited request in the load miss queue or in the store queue. -This is a speculative count.			
LARX Finished Tag: V	Number of completed LARX instructions. -Waits for reload from the L2			
Inter-Thread Store Set Watch Lost Indicator Tag: V	Number of Watch Lost indicator sets due to a different thread storing to a watched line by another thread.			
Reload Set Watch Lost Indi- cator Tag: V	Number of Watch Lost indicator sets due to a reload evicting watched line.			
Back-Invalidate Set Watch Lost Indicator Tag: V	Number of Watch Lost indicator sets due to a back-invalidate to a watched line.			
L1 Data Cache Back- Invalidate Tag: V	Number of back-invalidates sent to the L1 Data Cache.			

Table 6. LQ Performance Events Table (Use LESR1 and LESR2 for corresponding mux selects)
Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Speculative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
L1 Data Cache Back-Invalidate Hits Tag: V	Number of back-invalidates sent to the L1 Data Cache that invalidated a line.			
L1 Cache Parity Error Detected Tag: V	Number of parity errors detected in the L1 Directories and Caches. -Includes both Instruction and Data Directories and Caches. -Does not count more than one per cycle, although up to 4 may occur simultaneously.			
Load Latency Memory Sub-system Tag: B	Number of cycles load miss queue entry 0 is in use. Can be used to determine how often load miss queue entry 0 is used and how many cycles its in use.			

Table 7. LSU Prefetcher Performance Events Table (Use PESR for corresponding mux selects)
Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Speculative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
PFETCH: Prefetcher Event 0 Tag: V	Prefetcher Event 0, Event is defined in the prefetcher documentation			
PFETCH: Prefetcher Event 1 Tag: V	Prefetcher Event 1, Event is defined in the prefetcher documentation			
PFETCH: Prefetcher Event 2 Tag: V	Prefetcher Event 2, Event is defined in the prefetcher documentation			
PFETCH: Prefetcher Event 3 Tag: V	Prefetcher Event 3, Event is defined in the prefetcher documentation			
PFETCH: Prefetcher Event 4 Tag: V	Prefetcher Event 4, Event is defined in the prefetcher documentation			
PFETCH: Prefetcher Event 5 Tag: V	Prefetcher Event 5, Event is defined in the prefetcher documentation			
PFETCH: Prefetcher Event 6 Tag: V	Prefetcher Event 6, Event is defined in the prefetcher documentation			
PFETCH: Prefetcher Event 7 Tag: V	Prefetcher Event 7, Event is defined in the prefetcher documentation			

Table 7. LSU Prefetcher Performance Events Table (Use PESR for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Specu- lative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode

11.3.3.4 MMU performance events table

Table 8. MMU Performance Events Table (Use MESR1 and MESR2 for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Specu- lative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
tlb_hit_direct_ierat Tag: E	TLB hit direct entry (instruction, ind=0 entry hit for fetch)	No/Yes ¹	A0,A1	0/1
tlb_miss_direct_ierat Tag: E	TLB miss direct entry (instruction, ind=0 entry missed for fetch)	No/Yes ¹	A0,A1	2/3
tlb_miss_indirect_ierat Tag: E	TLB miss indirect entry (instruction, ind=1 entry missed for fetch, results in i-tlb exception)	No	A0,A1	4
htw_hit_ierat Tag: E	H/W tablewalk hit (instruction, ptereload with PTE.V=1 for fetch)	No	A0,A1	5
htw_miss_ierat Tag: E	H/W tablewalk miss (instruction, ptereload with PTE.V=0 for fetch, results in PT fault exception -> isi)	No	A0,A1	6
tlb_hit_direct_derat Tag: E	TLB hit direct entry (data, ind=0 entry hit for load/store/cache op)	No/Yes	B0,B1	0/1
tlb_miss_direct_derat Tag: E	TLB miss direct entry (data, ind=0 entry miss for load/store/cache op)	No/Yes	B0,B1	2/3
tlb_miss_indirect_derat Tag: E	TLB miss indirect entry (data, ind=1 entry missed for load/store/cache op, results in d-tlb exception)	No	B0,B1	4
htw_hit_derat Tag: E	H/W tablewalk hit (data, ptereload with PTE.V=1 for load/store/cache op)	No	B0,B1	5
htw_miss_derat Tag: E	H/W tablewalk miss (data, ptereload with PTE.V=0 for load/store/cache op, results in PT fault exception -> dsi)	No	B0,B1	6
ierat_miss_latency Tag: B	IERAT miss (edge) or latency (level) (total ierat misses or latency)	No/Yes ¹	B0,B1	8/9
derat_miss_latency Tag: B	DERAT miss (edge) or latency (level) (total derat misses or latency)	No/Yes	B0,B1	10/11

Table 8. MMU Performance Events Table (Use MESR1 and MESR2 for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Speculative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
ierat_miss_total Tag: E	IERAT miss total (part of direct entry search total)	No/Yes ¹	A0	8/9
derat_miss_total Tag: E	DERAT miss total (part of direct entry search total)	No/Yes	A0	10/11
tlb_miss_direct_total Tag: E	TLB miss direct entry total (total TLB ind=0 misses)	No/Yes	A0	12/13
tlb_hit_firstsize_total Tag: E	TLB hit direct entry first page size (first mmucr2 size)	No/Yes	A0	14/15
tlb_hit_indirect_total Tag: E	TLB indirect entry hits total (=page table searches)	No	B0	12
htw_ptereload_total Tag: E	H/W tablewalk successful installs total (with no PTfault, TLB ineligible, or LRAT miss)	No	B0	13
lrat_translation_total Tag: E	LRAT translation request total (for GS=1 tlbwe and ptereload)	No	B0	14
lrat_miss_total Tag: E	LRAT misses total (for GS=1 tlbwe and ptereload)	No	B0	15
pt_fault_total Tag: E	Page table faults total (PTE.V=0 for ptereload, resulting in isi/dsi)	No	B1	12
pt_inelig_total Tag: E	TLB ineligible total (all TLB ways are iprot=1 for ptereloads, resulting in isi/dsi)	No	B1	13
tlbwec_fail_total Tag: E	tlbwe conditional failed total (total tlbwe WQ=01 with no reservation match)	No	B1	14
tlbwec_success_total Tag: E	tlbwe conditional success total (total tlbwe WQ=01 with reservation match)	No	B1	15
tlbilx_local_source_total Tag: E	tlbilx local invalidations sourced total (sourced tlbilx on this core total)	No	A1	12
tlbivax_local_source_total Tag: E	tlbivax invalidations sourced total (sourced tlbivax on this core total)	No	A1	13
tlbivax_snoop_total Tag: E	tlbivax snoops total (total tlbivax snoops received from bus, local bit = don't care)	No	A1	14
tlb_flush_req_total Tag: B	TLB flush requests total (TLB requested flushes due to TLB busy or instruction hazards)	No	A1	15

NOTES:

1. The speculative nature of eras and tlb requests is defined by a bit sent with the request to the tlb. This bit may be tied to a constant value (TBD) for instruction side requests, causing both instruction fetch events to be identical.

2. Speculative as defined here means that speculative tlb hits will reload erat entries, but speculative tlb misses will not result in a hardware page table translation (i.e. only non-speculative causes this).

11.3.3.5 RV performance events table

Table 9. RV Performance Events Table (Use RESR1 and RESR2 for corresponding mux selects)				
Note: Refer to the unit performance events table column descriptions in <i>Section 11.3.2</i> on page 446.				
Event Name Tag: B/C/E/S/V	Description	Specu- lative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
FX0 RV Empty (S)	No valid entries in the RV queue	Y		
FX0 RV Issued OoO (S)	An instruction issued Out of Order	Y		
FX0 RV Above Watermark (S)	The number of instruction in the RV queue exceeds the watermark as set by the RVCR0. This is separate from RV credits. Watermark defaults to Full-1 (count when full).	Y		
FX0 Instr Issued (S)	Instruction is issued from the RV queue.	Y		
FX0 Ordered Hold (S)	An ordered instruction is in the RV queue waiting on another ordered instruction. This can be due to a shared resource.	Y		
FX0 COrdered Hold (S)	An instruction is waiting to be issued in completion order	Y		
FX0 Dep Hold (S)	Instruction(s) are valid but none are issued this cycle due to source dependencies or any other holds	Y		
FX1 RV Empty (S)	No valid entries in the RV queue	Y		
FX1 RV Issued OoO (S)	An instruction issued Out of Order	Y		
FX1 RV Above Watermark (S)	The number of instruction in the RV queue exceeds the watermark as set by the RVCR0. This is separate from RV credits. Watermark defaults to Full-1 (count when full).	Y		
FX1 Instr Issued (S)	Instruction is issued from the RV queue.	Y		
FX1 Ordered Hold (S)	An ordered instruction is in the RV queue waiting on another ordered instruction. This can be due to a shared resource.	Y		
FX1 COrdered Hold (S)	An instruction is waiting to be issued in completion order	Y		
FX1 Dep Hold (S)	Instruction(s) are valid but none are issued this cycle due to source dependencies or any other holds	Y		
LQ RV Empty (S)	No valid entries in the RV queue	Y		
LQ RV Issued OoO (S)	An instruction issued Out of Order.	Y		
LQ RV Above Watermark (S)	The number of instruction in the RV queue exceeds the watermark as set by the RVCR0. This is separate from RV credits. Watermark defaults to Full-1 (count when full).	Y		
LQ Instr Issued (S)	Instruction is issued from the RV queue.	Y		
LQ Ordered Hold (S)	An ordered instruction is in the RV queue waiting on another ordered instruction. This can be due to a shared resource.	Y		
LQ COrdered Hold (S)	An instruction is waiting to be issued in completion order	Y		
LQ Dep Hold (S)	Instruction(s) are valid but none are issued this cycle due to source dependencies or any other holds	Y		

Table 9. RV Performance Events Table (Use RESR1 and RESR2 for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Speculative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
AXU0 RV Empty (S)	No valid entries in the RV queue	Y		
AXU0 RV Issued OoO (S)	An instruction issued Out of Order	Y		
AXU0 RV Above Watermark (S)	The number of instruction in the RV queue exceeds the watermark as set by the RVCR0. This is separate from RV credits. Watermark defaults to Full-1 (count when full).	Y		
AXU0 Instr Issued (S)	Instruction is issued from the RV queue.	Y		
AXU0 Ordered Hold (S)	An ordered instruction is in the RV queue waiting on another ordered instruction. This can be due to a shared resource.	Y		
AXU0 COrdered Hold (S)	An instruction is waiting to be issued in completion order	Y		
AXU0 Dep Hold (S)	Instruction(s) are valid but none are issued this cycle due to source dependencies or any other holds	Y		
AXU1 Reserved	Perf events implementation dependent from AXU1	Y		
AXU1 Reserved	Perf events implementation dependent from AXU1	Y		
AXU1 Reserved	Perf events implementation dependent from AXU1	Y		
AXU1 Reserved	Perf events implementation dependent from AXU1	Y		
AXU1 Reserved	Perf events implementation dependent from AXU1	Y		
AXU1 Reserved	Perf events implementation dependent from AXU1	Y		
AXU1 Reserved	Perf events implementation dependent from AXU1	Y		
AXU1 Reserved	Perf events implementation dependent from AXU1	Y		
AXU1 Reserved	Perf events implementation dependent from AXU1	Y		

11.3.3.6 XU performance events tables

Table 10. XU Performance Events Table (Use XESR1 for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Speculative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
Processor Busy Tag: C	Cycles that any thread is running			
Branch Commit Tag: C	Number of Branches committed			
Branch Mispredict Commit Tag: S	Number of mispredicted Branches committed (does not include target address mispredicted)			
Branch Target Address Mispredict Commit Tag: S	Number of Branch Target addresses mispredicted committed			

Table 10. XU Performance Events Table (Use XESR1 for corresponding mux selects)

Note: Refer to the unit performance events table column descriptions in *Section 11.3.2* on page 446.

Event Name Tag: B/C/E/S/V	Description	Speculative? Y,N	T0 Mux Event Decode	T1 Mux Event Decode
Thread Running Tag: C	Number of cycles that thread is in run state			
Timebase Tick Tag: C	Number of times the timebase has incremented			
SPR Read Commit Tag: C	Number of mfspr, mftb, mfmsr or mfcrr instructions committed			
SPR Write Commit Tag: C	Number of mtspr, mtmsr, mtcrf, wrtee, wrteei instructions committed			
Cycles stalled on waitrsv Tag: B	Number of cycles between commit of waitrsv and wakeup by lost reservation.		Move to CP	
External Interrupt Asserted Tag: C	Number of cycles the external interrupt signal is asserted			
Critical External Interrupt Asserted Tag: C	Number of cycles the critical external interrupt signal is asserted			
Performance Monitor Interrupt Asserted Tag: C	Number of cycles the performance monitor interrupt signal is asserted			
XU Commit Tag: C	Number of XU instructions committed. Every instruction of uCode sequence is counted.			
XU Flush Tag: C	Number of cycles flush is asserted to the XU			
Branch Commit Tag: C	Number of Branches committed			
Branch Mispredict Commit Tag: S	Number of mispredicted Branches committed (does not include target address mispredicted)			
Branch Taken Commit Tag: C	Number of taken branches committed			
Branch Target Address Mispredict Commit Tag: S	Number of Branch Target addresses mispredicted committed			
Mult/Div Collision Tag: C	Number of Multiply/Divide resource collisions		Move to RV?	
Mult/Div Busy Tag: C	Number of cycles the multiplier or divider is in use.			

Bit(s):	Field Name:	Init	Description
40:43	MUXSELEB2	0b0000	<u>Mux Event_Bits(2) Mux Select</u> Event mux select for performance event bit 2
44:47	MUXSELEB3	0b0000	<u>Mux Event_Bits(3) Mux Select</u> Event mux select for performance event bit 3
48:51	MUXSELEB4	0b0000	<u>Mux Event_Bits(4) Mux Select</u> Event mux select for performance event bit 4
52:55	MUXSELEB5	0b0000	<u>Mux Event_Bits(5) Mux Select</u> Event mux select for performance event bit 5
56:59	MUXSELEB6	0b0000	<u>Mux Event_Bits(6) Mux Select</u> Event mux select for performance event bit 6
60:63	MUXSELEB7	0b0000	<u>Mux Event_Bits(7) Mux Select</u> Event mux select for performance event bit 7

11.4.2 IU event select registers

Register Short Name:	IESR1	Read Access:	Priv
Decimal SPR Number:	914	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Slow	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:37	MUXSELEB0	0x0	<u>Mux Event_Bits(0) Mux Select</u> Event mux select for performance event bit 0
38:43	MUXSELEB1	0x0	<u>Mux Event_Bits(1) Mux Select</u> Event mux select for performance event bit 1
44:49	MUXSELEB2	0x0	<u>Mux Event_Bits(2) Mux Select</u> Event mux select for performance event bit 2
50:55	MUXSELEB3	0x0	<u>Mux Event_Bits(3) Mux Select</u> Event mux select for performance event bit 3
56:63	///	0x0	<u>Reserved</u>

Register Short Name:	IESR2	Read Access:	Priv
Decimal SPR Number:	915	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Slow	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:37	MUXSELEB4	0x0	<u>Mux Event_Bits(4) Mux Select</u> Event mux select for performance event bit 4
38:43	MUXSELEB5	0x0	<u>Mux Event_Bits(5) Mux Select</u> Event mux select for performance event bit 5

Bit(s):	Field Name:	Init	Description
44:49	MUXSELEB6	0x0	<u>Mux Event_Bits(6) Mux Select</u> Event mux select for performance event bit 6
50:55	MUXSELEB7	0x0	<u>Mux Event_Bits(7) Mux Select</u> Event mux select for performance event bit 7
56:63	///	0x0	<u>Reserved</u>

Register Short Name:	IESR3	Read Access:	Priv
Decimal SPR Number:	924	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Slow	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:35	MUXSELEB0	0b0000	<u>Mux Event_Bits(0) Mux Select</u> Event mux select for performance event bit 0
36:39	MUXSELEB1	0b0000	<u>Mux Event_Bits(1) Mux Select</u> Event mux select for performance event bit 1
40:43	MUXSELEB2	0b0000	<u>Mux Event_Bits(2) Mux Select</u> Event mux select for performance event bit 2
44:47	MUXSELEB3	0b0000	<u>Mux Event_Bits(3) Mux Select</u> Event mux select for performance event bit 3
48:51	MUXSELEB4	0b0000	<u>Mux Event_Bits(4) Mux Select</u> Event mux select for performance event bit 4
52:55	MUXSELEB5	0b0000	<u>Mux Event_Bits(5) Mux Select</u> Event mux select for performance event bit 5
56:59	MUXSELEB6	0b0000	<u>Mux Event_Bits(6) Mux Select</u> Event mux select for performance event bit 6
60:63	MUXSELEB7	0b0000	<u>Mux Event_Bits(7) Mux Select</u> Event mux select for performance event bit 7

Note: Due to an increase in the number of completion unit events, the completion event muxes will be configured for 32 events. An IESR3 change is required; IESR4 will be added.

11.4.3 LQ event select registers

Register Short Name:	LESR1	Read Access:	Priv
Decimal SPR Number:	920	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Slow	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:37	MUXSELEB0	0x0	<u>Mux Event_Bits(0) Mux Select</u> Event mux select for performance event bit 0
38:43	MUXSELEB1	0x0	<u>Mux Event_Bits(1) Mux Select</u> Event mux select for performance event bit 1
44:49	MUXSELEB2	0x0	<u>Mux Event_Bits(2) Mux Select</u> Event mux select for performance event bit 2
50:55	MUXSELEB3	0x0	<u>Mux Event_Bits(3) Mux Select</u> Event mux select for performance event bit 3
56:63	///	0x0	<u>Reserved</u>

Register Short Name:	LESR2	Read Access:	Priv
Decimal SPR Number:	921	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Slow	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:37	MUXSELEB4	0x0	<u>Mux Event_Bits(4) Mux Select</u> Event mux select for performance event bit 4
38:43	MUXSELEB5	0x0	<u>Mux Event_Bits(5) Mux Select</u> Event mux select for performance event bit 5
44:49	MUXSELEB6	0x0	<u>Mux Event_Bits(6) Mux Select</u> Event mux select for performance event bit 6
50:55	MUXSELEB7	0x0	<u>Mux Event_Bits(7) Mux Select</u> Event mux select for performance event bit 7
56:63	///	0x0	<u>Reserved</u>

Register Short Name:	PESR	Read Access:	Priv
Decimal SPR Number:	893	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Slow	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:35	MUXSELEB0	0b0000	<u>Mux Event_Bits(0) Mux Select</u> Event mux select for performance event bit 0
36:39	MUXSELEB1	0b0000	<u>Mux Event_Bits(1) Mux Select</u> Event mux select for performance event bit 1
40:43	MUXSELEB2	0b0000	<u>Mux Event_Bits(2) Mux Select</u> Event mux select for performance event bit 2
44:47	MUXSELEB3	0b0000	<u>Mux Event_Bits(3) Mux Select</u> Event mux select for performance event bit 3

Bit(s):	Field Name:	Init	Description
48:51	MUXSELEB4	0b0000	<u>Mux Event_Bits(4) Mux Select</u> Event mux select for performance event bit 4
52:55	MUXSELEB5	0b0000	<u>Mux Event_Bits(5) Mux Select</u> Event mux select for performance event bit 5
56:59	MUXSELEB6	0b0000	<u>Mux Event_Bits(6) Mux Select</u> Event mux select for performance event bit 6
60:63	MUXSELEB7	0b0000	<u>Mux Event_Bits(7) Mux Select</u> Event mux select for performance event bit 7

11.4.4 MMU event select registers

Register Short Name:	MESR1	Read Access:	Priv
Decimal SPR Number:	916	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Slow	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:36	MUXSELEB0	0x0	<u>Mux Event_Bits(0) Mux Select</u> Event mux select for performance event bit 0
37:41	MUXSELEB1	0x0	<u>Mux Event_Bits(1) Mux Select</u> Event mux select for performance event bit 1
42:46	MUXSELEB2	0x0	<u>Mux Event_Bits(2) Mux Select</u> Event mux select for performance event bit 2
47:51	MUXSELEB3	0x0	<u>Mux Event_Bits(3) Mux Select</u> Event mux select for performance event bit 3
52:63	///	0x0	<u>Reserved</u>

Register Short Name:	MESR2	Read Access:	Priv
Decimal SPR Number:	917	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Slow	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:36	MUXSELEB4	0x0	<u>Mux Event_Bits(4) Mux Select</u> Event mux select for performance event bit 4
37:41	MUXSELEB5	0x0	<u>Mux Event_Bits(5) Mux Select</u> Event mux select for performance event bit 5
42:46	MUXSELEB6	0x0	<u>Mux Event_Bits(6) Mux Select</u> Event mux select for performance event bit 6

Bit(s):	Field Name:	Init	Description
47:51	MUXSELEB7	0x0	<u>Mux Event_Bits(7) Mux Select</u> Event mux select for performance event bit 7
52:63	///	0x0	<u>Reserved</u>

11.4.5 RV event select registers

Register Short Name:	RESR1	Read Access:	Priv
Decimal SPR Number:	922	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Slow	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:36	MUXSELEB0	0x0	<u>Mux Event_Bits(0) Mux Select</u> Event mux select for performance event bit 0
37:41	MUXSELEB1	0x0	<u>Mux Event_Bits(1) Mux Select</u> Event mux select for performance event bit 1
42:46	MUXSELEB2	0x0	<u>Mux Event_Bits(2) Mux Select</u> Event mux select for performance event bit 2
47:51	MUXSELEB3	0x0	<u>Mux Event_Bits(3) Mux Select</u> Event mux select for performance event bit 3
52:63	///	0x0	<u>Reserved</u>

Register Short Name:	RESR2	Read Access:	Priv
Decimal SPR Number:	923	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Slow	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:36	MUXSELEB4	0x0	<u>Mux Event_Bits(4) Mux Select</u> Event mux select for performance event bit 4
37:41	MUXSELEB5	0x0	<u>Mux Event_Bits(5) Mux Select</u> Event mux select for performance event bit 5
42:46	MUXSELEB6	0x0	<u>Mux Event_Bits(6) Mux Select</u> Event mux select for performance event bit 6
47:51	MUXSELEB7	0x0	<u>Mux Event_Bits(7) Mux Select</u> Event mux select for performance event bit 7
52:63	///	0x0	<u>Reserved</u>

11.4.6 XU event select registers

Register Short Name:	XESR1	Read Access:	Priv
Decimal SPR Number:	918	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Normal	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:35	MUXSELEB0	0b0000	<u>Mux Event_Bits(0) Mux Select</u> Event mux select for performance event bit 0
36:39	MUXSELEB1	0b0000	<u>Mux Event_Bits(1) Mux Select</u> Event mux select for performance event bit 1
40:43	MUXSELEB2	0b0000	<u>Mux Event_Bits(2) Mux Select</u> Event mux select for performance event bit 2
44:47	MUXSELEB3	0b0000	<u>Mux Event_Bits(3) Mux Select</u> Event mux select for performance event bit 3
48:51	MUXSELEB4	0b0000	<u>Mux Event_Bits(4) Mux Select</u> Event mux select for performance event bit 4
52:55	MUXSELEB5	0b0000	<u>Mux Event_Bits(5) Mux Select</u> Event mux select for performance event bit 5
56:59	MUXSELEB6	0b0000	<u>Mux Event_Bits(6) Mux Select</u> Event mux select for performance event bit 6
60:63	MUXSELEB7	0b0000	<u>Mux Event_Bits(7) Mux Select</u> Event mux select for performance event bit 7

Register Short Name:	XESR2	Read Access:	Priv
Decimal SPR Number:	919	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Class:	Normal	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:35	MUXSELEB0	0b0000	<u>Mux Event_Bits(0) Mux Select</u> Selects thread 0 non-speculative branch events for performance event bit 0
36:39	MUXSELEB1	0b0000	<u>Mux Event_Bits(1) Mux Select</u> Selects thread 0 non-speculative branch events for performance event bit 1
40:43	MUXSELEB2	0b0000	<u>Mux Event_Bits(2) Mux Select</u> Selects thread 0 non-speculative branch events for performance event bit 2
44:47	MUXSELEB3	0b0000	<u>Mux Event_Bits(3) Mux Select</u> Selects thread 0 non-speculative branch events for performance event bit 3
48:51	MUXSELEB4	0b0000	<u>Mux Event_Bits(4) Mux Select</u> Selects thread 1 non-speculative branch events for performance event bit 4
52:55	MUXSELEB5	0b0000	<u>Mux Event_Bits(5) Mux Select</u> Selects thread 1 non-speculative branch events for performance event bit 5

Bit(s):	Field Name:	Init	Description
56:59	MUXSELEB6	0b0000	<u>Mux_Event_Bits(6) Mux Select</u> Selects thread 1 non-speculative branch events for performance event bit 6
60:63	MUXSELEB7	0b0000	<u>Mux_Event_Bits(7) Mux Select</u> Selects thread 1 non-speculative branch events for performance event bit 7

11.5 A2O support for core instruction trace

A2 support for the instruction trace function is **TBD**. Any details of the planned instruction trace implementation are backlevel due to the trace bus size being reduced to 32 bits.

Core instruction tracing allows chip level facilities to collect instruction address information over extended periods of time, and store it out in system memory for subsequent performance analysis. This can involve multiple Hardware Trace Macros (HTM) and additional support logic from various units outside the core. A2 supports core trace operations through the following functions:

- ~~Instruction Tracing—each instruction and its relevant address information is driven out onto the debug bus. This mode requires single thread execution in slow mode.~~
- ~~Trace SPR—A Trace SPR will be supported. Software can issue a MTSPR Trace instruction, which enables placement of instruction mark data into the trace record.~~

~~When in *Instruction Trace Mode*, the A2 core will continuously place instruction trace data onto the external debug bus. Additional control signals enable the HTM to interpret the trace data, format it, and write it to memory. This section describes instruction trace mode setup, the A2 core instruction trace data, and how instruction trace mode is used to control placement of this data onto the debug bus.~~

11.5.1 Instruction trace mode setup

The following core facilities need to be configured in order to enable instruction tracing:

- ~~Enable execution on a single thread (TENC or THRCTL[Tx_STOP])~~
- ~~Configure core for single instruction execution (CGR3[SI]).~~
This is an optional step; refer to note below on CESR1 settings.
- ~~Enable writes to TRACE SPR (CGR2[EN_TRACE])~~
- ~~If desired, enable user mode writes to TRACE SPR (XUCR0[TRACE_UM])~~
- ~~Set all unit debug mux control registers to the pass thru mode so that they do not interfere with instruction trace data written to the trace bus. The IU, XU and LSU will drive out on the trace bus as required.~~

Note: ~~The normal power on reset state of the core will initialize all of the debug mux control registers (ARDSR, IDSR, MPDSR, XDSR and LDSR) to 0, which will put them in the pass thru state by default.~~

- ~~Enable *Instruction Trace Mode* (CESR1[INSTTRACE]) and select the thread of operation (CESR1[INSTTRACETID]). This configures the core to drive instruction trace data onto the debug bus when the trace run bit is set. The CESR1[ENABPERF] and CESR1[ENABTRACEBUS] bits must also be set in order to enable clocking to the debug bus and performance related latches involved in instruction tracing.~~

Note: Upon entering instruction trace mode, the selected thread is automatically configured for single-instruction execution by the XU. The state of the GCR3[SI] bit is not affected by this action however, and will provide no status or indication that single instruction execution is active.

- The ITRUN[TRACE_RUN] bit is used to start or stop instruction tracing. When the ITRUN[TRACE_RUN] bit is set, trace data associated with each instruction is placed on the debug bus. When cleared, instruction execution continues but the instruction trace data is not driven out on the debug bus.

11.5.2 Instruction trace record data and control signals

The IU, XU and LSU participate in core trace by driving instruction trace data and control signals outside of the core. Some control signals are placed on the debug bus and written to memory along with the instruction trace data. Other control signals are driven to the HTM logic directly through separate core output pins. Table 11-12 describes the data and controls signals, which unit supplies it, and its size.

Table 11-12 describes the data and controls signals, which unit supplies it, and its size.

Trace Data Type	Unit Driving	Size (bits)	Comments
Instruction Opcode	IU	32	32-bit opcode field
Instruction Effective Address (IEA)	IU	62	
MTSPR Data	XU	64	Data written to a SPR from MTSPR, MTMSR, etc.
Data Effective Address (DEA)	LSU	64	
Data Real Address (DRA)	LSU	40	
xABCDE data pattern	IU	20	Specific data pattern. Part of the information used by software to identify the first instruction trace record.
First Instruction Trace Record Valid	IU	4	This signal is driven out of the core through 2 output signals: ▲ debug-bus(56) ▲ ac_an_coretrace_first_valid
Encoded Trace Record Type	IU	2	A 2-bit encoded field included in the 1st instruction trace record which indicates how many data packets will follow. The <i>Trace Record Type</i> decode is shown below: 00—Opcode only 01—Opcode and IEA 10—Opcode and IEA and MTSPR data 11—Opcode and IEA and DEA and DRA This signal is driven out of the core through 2 output signals: ▲ debug-bus(57-58) ▲ ac_an_coretrace_type(0:1)
Trace Record Valid	IU,XU, LSU	4	This signal is driven out of the core as ac_an_coretrace_valid

11.5.3 Instruction trace record formats and ordering

The *First Instruction Trace Record* includes the opcode field, a unique data pattern, and other control signals which enable the HTM and post-processing software to identify it as the first trace record, and determine what additional trace records will follow. The number and type of additional trace records are determined by the value of the encoded *Trace Record Type* field.

The tables in this section describe how the instruction trace records are driven onto the debug bus, and the specific ordering of the trace records for each trace record type.

Table 13. First Instruction Trace Record format

Core Outputs	Function
debug_bus(0:31)	Opcode
debug_bus(32:35)	Reserved
debug_bus(36:55)	Unique pattern for software identification (xABCDE)
debug_bus(56)	First instruction trace record valid bit
debug_bus(57:58)	Encoded <i>Trace Record Type</i> bits (as described in Table 11-12)
debug_bus(59:63)	Reserved
ae_an_coretrace_valid	Trace record valid signal (used by HTM logic)
ae_an_coretrace_first_valid	First instruction trace record valid bit (used by HTM logic)
ae_an_coretrace_type(0:1)	Encoded <i>Trace Record Type</i> bits (used by HTM logic)

Table 14. Format of Subsequent Instruction Trace Records

Core Outputs	Function
debug_bus(0:63)	Trace record data (either IEA, DEA, DRA or MTSPR data). The number of debug bus bits used, and placement, depends on the particular data type being driven.
ae_an_coretrace_valid	Trace record valid indicator (used by HTM logic). Note: Subsequent trace records may not occur on consecutive cycles. Data placed on the debug bus is a valid instruction trace record only when this signal is active.

Table 15. Trace Record Type decode and Instruction Trace Record ordering

Trace Record Number	Encoded Trace Record Type Bits			
	00	04	40	44
1	1st Instruction Trace Record	1st Instruction Trace Record	1st Instruction Trace Record	1st Instruction Trace Record
2		Instruction Effective Address	MTSPR Data	Data Effective Address
3			Instruction Effective Address	Data Real Address
4				Instruction Effective Address

11.5.4 Additional instruction tracing actions

11.5.4.1 Instruction trace actions when an instruction is flushed

If a *1st Instruction Trace Record* is driven out on the debug bus and that instruction does not complete, the IU will not activate `ac_an_coretrace_valid` when the IEA is placed on the bus. The resulting trace records stored to memory will contain less data than indicated by the trace record type field. Software can use this information to discard the incomplete trace record.

11.5.4.2 Debug bus control when in instruction trace mode

With each unit's debug select registers in the pass through state, the IU, XU and LSU can drive debug bus data and control signals as required while in instruction trace mode.

~~Methods used to coordinate these actions among the three units and avoid conflicts are TBD.~~

11.5.5 Trace SPR

Register Short Name:	TRACE	Read Access:	None
Decimal SPR Number:	1006	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description										
32:63	DATA	0x0	<p><u>Trace Control Data</u></p> <p>This register may be used to control trace features on some L2 implementations. Writing to this register causes a store like transaction on the L2 interface, with a TTYPE of <code>MTSPR_TRACE</code> if the expression <code>(CCR2[EN_TRACE] and (XUCR0[TRACE_UM] or not MSR[PR]))=1</code> for the executing thread. If the expression is false, the operation is treated as a nop. The data written to this field is placed in the address on the L2 interface according to the table below. See L2 specification for details on this feature.</p> <p>Trace L2 req_ra</p> <p>-----</p> <table> <tr> <td>50:59</td> <td>34:43</td> </tr> <tr> <td>60</td> <td>45</td> </tr> <tr> <td>61</td> <td>48</td> </tr> <tr> <td>62</td> <td>47</td> </tr> <tr> <td>63</td> <td>46</td> </tr> </table>	50:59	34:43	60	45	61	48	62	47	63	46
50:59	34:43												
60	45												
61	48												
62	47												
63	46												

11.6 A2O support for instruction sampling

The A2O core implements per-thread Sampled Instruction Address Registers (SIAR) that capture the instruction address upon activation of a performance monitor alert. Additional bits in CESR1 are used to enable performance monitor alerts, and indicate when one has occurred.

11.6.1 Sampled Instruction Address Register (SIAR)

Each completion unit has an associated SIAR which, upon activation of the thread's performance monitor alert, is updated with sampled address and MSR bits. The format for data written to the SIAR is shown below:

Register Short Name:	SIAR	Read Access:	Priv
Decimal SPR Number:	796	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SIAR	0x0	<p><u>Sampled Instruction Address Register</u></p> <p>Upon activation of a threads performance monitor alert, sampled instruction address and MSR values are loaded into the SIAR as follows:</p> <p>0:61 Sampled Instruction Address (corresponding Next Instruction Address register bits)</p> <p>62 Sampled MSR[GS]</p> <p>63 Sampled MSR[PR]</p>

11.6.2 Instruction sampling control bits

For each thread, CESR1 implements a pair of control bits that enable performance monitor alerts, and indicate when sampled address information has been loaded into the SIAR.

11.6.2.1 Performance Monitor Alert Enable (PMAE)

This bit indicates that performance monitor alerts are enabled when set, and disabled when cleared. Software can set or clear this bit. This bit is cleared by hardware when a performance monitor alert is activated.

11.6.2.2 Performance Monitor Alert Occurred (PMAO)

This bit indicates that a performance monitor alert has occurred. Software can set or clear this bit. It is set by hardware when an enabled performance monitor alert occurs.

11.6.2.3 Combined PMAO and PMAE function

PMAO	PMAE	Description
0	0	Performance monitor alerts (and interrupts) are disabled.
0	1	Performance monitor alerts are enabled, but none have occurred since the last time software cleared PMAO.
1	0	A performance monitor alert has occurred. In response, hardware has taken the following actions: <ul style="list-style-type: none"> • set PMAO • cleared PMAE • loaded SIAR with the sampled address and MSR values
1	1	These values are possible only if set by software (i.e. to simulate activation of a performance monitor alert).



12. Implementation Dependent Instructions

This chapter describes all the A2O Core instructions implemented that are not part of Power ISA or that are implementation dependent.

12.1 Miscellaneous

12.1.1 Attention (attn)



For purposes of hardware debugging, the processor supports a special, implementation-dependent instruction for signaling an “attention” signal to system-level hardware, which is beyond the scope of this document. This instruction s per thread and causes the following sequence:

1. A normal CSI event is generated for the thread issuing the instruction.
2. The instructions following the attention instruction are flushed for the thread issuing the instruction.
3. A dispatch stall is enabled so that no further instructions can be dispatched for the thread issuing the instruction.
4. An attention signal, if enabled by the corresponding Special Attention Register (SPATTN) mask bit, is sent to system-level hardware.

The **attn** instruction has the following definition:

- The immediate field (I) has no effect on the operation of this instruction.
- If CCR2[en_attn] = 1 (support processor attention enable bit is set), this instruction causes all preceding instructions to run to completion, the machine to quiesce, and a bit in the Special Attention Register (SPATTN) to be set. If enabled by the corresponding SPATTN register mask bit, a support processor attention signal is asserted.
- If CCR2[en_attn] = 0 (support processor attention enable bit is not set), this instruction causes an illegal instruction type of program interrupt.

12.2 TLB Management Instructions

12.2.1 TLB Read Entry (tlbre)

Software must use the **tlbre** instruction to read entries from the TLB or LRAT. This instruction is embedded hypervisor privileged. Execution of this instruction in guest state (GS = 1) results in an embedded hypervisor privilege exception.

Because this instruction relies on the MMU Assist (MAS) Registers, execution of this instruction in ERAT-only mode (CCR2[NOTLB] = 1) results in an illegal instruction exception. The instruction format and details follow.

tlbre

31	///	///	///	946	/
0	6	11	16	21	31

if $MAS0_{ATSEL} = 0$ then

entry \leftarrow SelectTLB($MAS2_{EPN(27:51)}$, $MAS1_{TID, TSIZE}$, $MAS0_{ESEL(1:2)}$)

rpnl \leftarrow entry_{RPN}

$MAS1_{V IPROT TID TS TSIZE IND} \leftarrow$ entry_{V IPROT TID TS SIZE IND}

if $MSR_{CM} = 0$ then

$MAS2_{EPN[0:31]} \leftarrow 0$

$MAS2_{EPN[32:51] W I M G E} \leftarrow$ entry_{EPN[32:51] W I M G E}

else

$MAS2_{EPN W I M G E} \leftarrow$ entry_{EPN W I M G E}

$MAS3_{RPNL} \leftarrow$ rpnl_{32:51}

$MAS3_{U0:U3} \leftarrow$ entry_{U0:U3}

entry_{SPSIZE0 SPSIZE1 SPSIZE2 SPSIZE3} || 0 0

if entry_{IND} = 1

$MAS3_{SPSIZE0 SPSIZE1 SPSIZE2 SPSIZE3 SPSIZE4 UND} \leftarrow$

$MAS3_{RPNL[52]} \leftarrow$ entry_{RPN[52]}

else

$MAS3_{UX SX UW SW UR SR} \leftarrow$ entry_{UX SX UW SW UR SR}

$MAS3_{RPNL[52]} \leftarrow 0$

```

ExtClass TID_NZ Class WLC ResvAttr ThdID
else
entry ← SelectLRAT(MAS0_ESEL)

MAS1_IPROT TID IND TS ← 0 0 0 0

MAS2_EPN[0:21] ← 0
MAS2_EPN[22:43] ← entry_LPN[22:43]
MAS2_EPN[44:51] ← 0
MAS2_W I M G E ← 0 0 0 0 0
MAS3_RPNL[22:43] ← rpn_22:43
MAS3_RPNL[44:51] ← 0
MAS3_U0:U3 UX SX UW SW UR SR ← 0 0 0 0 0 0 0
MAS7_RPNU ← rpn_22:31

MAS1_V TSIZE ← entry_V SIZE

MAS1_IND ← 0

MAS8_TGS VF ← 0 0
MAS8_TLPID ← entry_LPID
MMUCR3_X ← entry_X
MMUCR3_R C ECL TID_NZ Class WLC ResvAttr ThdID ← 0 0 0 0 0 0 0

0b1111

```

For reading TLB entries, MAS0.ATSEL must be set to 0. The congruence class of the set-associative TLB array is selected by a hardware hash based on MAS0.EPN bits [27:51], MAS1.TID, and MAS1.TSIZE. The TLB way is selected by MAS0.ESEL bits [1:2]. The MSBs of the MAS2.EPN and MAS0.ESEL fields are ignored when reading from the TLB (that is, only certain bits of these fields are used in the hash).

For reading LRAT entries, MAS0.ATSEL must be set to 1. The entry number of the fully-associative LRAT array is selected by MAS0.ESEL bits [0:2]. The MAS2.EPN, MAS1.TID, and TSIZE fields are not used.

This implementation requires the page size to be specified by MAS1.TSIZE to calculate the congruence class of the set-associative TLB array. If the page size specified by MAS1.TSIZE is not supported by this implementation, an illegal instruction exception is generated.

The MAS Registers shown above are updated with the associated fields from the selected TLB or LRAT entry at the completion of the **tlbre** instruction. The MAS registers can be subsequently read via one or more **mfspr** instructions. See *Section 6.17.28 MAS Register Update Summary* on page 259 for a description of values loaded into the MAS registers for this instruction.

Note: The architecturally defined fields of MAS0.TLBSEL, MAS0.NV, MAS2.VLE, and MAS2.ACM are not included in the explanation of this instruction because they are reserved in this implementation.

12.2.2 TLB Write Entry (tlbwe)

Software must use the **tlbwe** instruction to write entries into either the TLB or LRAT. This instruction is supervisor privileged.

Because this instruction relies on the MAS Registers, execution of this instruction in ERAT-only mode (CCR2[NOTLB] = 1) results in an illegal instruction exception. The instruction format and details follow.

tlbwe

31	///	///	///	978	/
0	6	11	16	21	31

if MAS0_{WQ} = 0b00 | MAS0_{WQ} = 0b01 | MAS0_{WQ} = 0b11 then

if MAS0_{ATSEL} = 0 or MSR_{GS} = 1 then
 if MAS0_{HES} = 0 then

entry ← SelectTLB(MAS1_{TID} TSIZE, MAS2_{EPN}, MAS0_{ESEL})

else

entry ← SelectTLB(MAS1_{TID} TSIZE, MAS2_{EPN}, hardware_replacement_algorithm)

if (MAS0_{WQ} = 0b00) | (MAS0_{WQ} = 0b01 & TLB reservation) | (MAS0_{WQ} = 0b11) then

if (MSR_{GS} = 1) & (MAS1_V = 1) then

rpn ← translate_logical_to_real(MAS7_{RPNU} || MAS3_{RPNL},

MAS8_{TLPID})

else

rpn ← MAS7_{RPNU} || MAS3_{RPNL}

entry_V IPROT TID TS SIZE ← MAS1_V IPROT TID TS SIZE

entry_{WIMGE} ← MAS2_{WIMGE}

entry_{U0:U3} ← MAS3_{U0:U3}

if MAS1_{IND} = 1 and TLB0CFG_{IND} = 1 then

entry_{SPSIZE0 SPSIZE1 SPSIZE2 SPSIZE3 SPSIZE4 RPN[52]} ← MAS3_{SP-}
 SIZE0 SPSIZE1 SPSIZE2 SPSIZE3 || 0 || rpn₅₂

entry_{IND} ← MAS1_{IND}

else

entry_{UX SX UW SW UR SR} ← MAS3_{UX SX UW SW UR SR}

entry_{IND} ← 0

```

if MSRCM = 0 then
  entryEPN[0:31] ← 0
  entryEPN[32:51] ← MAS2EPN[32:51]
else
  entryEPN ← MAS2EPN

```

```

entryRPN[22:51] ← rpn22:51

entryTGS VF TLPID ← MAS8TGS VF TLPID
entryX R C Class WLC ResvAttr ThdID ← MMUCR3X R C Class WLC

```

ResvAttr ThdID

```

entryExtClass ← MAS1IPROT & MMUCR3ECL

entryTID_NZ ← or_reduce(MAS1TID)

```

```

else

entry ← SelectLRAT(MAS0ESEL)

if (MAS0WQ = 0b00) & (MAS0HES = 0b0) then

  entryV SIZE ← MAS1V TSIZE
  entryLPN ← MAS2EPN[22:43]

  entryRPN ← MAS7RPNU[22:31] || MAS3RPNL[32:43]

  entryLPID ← MAS8TLPID

  entryX ← MMUCR3X

```

```

TLB reservationV ← 0

```

else

```

TLB reservationV ← 0

```

For writing TLB entries, MAS0.ATSEL must be set to 0. The congruence class of the set-associative TLB array is selected by a hardware hash based on MAS2.EPN bits [27:51], MAS1.TID, and MAS1.TSIZE. When MAS0.HES = 0, the TLB way is selected by MAS0.ESEL bits [1:2]. When MAS0.HES = 1, the TLB way is selected by a hardware pseudo-LRU replacement algorithm. The MSBs of the MAS2.EPN and MAS0.ESEL fields are ignored when writing to the TLB (that is, only certain bits of these fields are used in the hash).

For writing LRAT entries, MAS0.ATSEL must be set to 1, MAS0.HES must be set 0, and MAS0.WQ must be set to 0 or 3. The entry number of the fully-associative LRAT array is selected by MAS0.ESEL bits [0:2]. The MAS2.EPN field is not used to select an entry. If a **tlbwe** instruction with MAS0.ATSEL = 1 is attempted and either MAS0.HES = 1 or the MAS0.WQ = 1 or 2, an illegal instruction exception is generated.

The MAS register contents shown above are used as the source data for the associated fields to be updated in the selected TLB or LRAT entry at the completion of the **tlbwe** instruction. The MAS registers are assumed to have been previously written via one or more **mtspr** instructions.

This implementation requires the page size to be specified by MAS1_{TSIZE} to calculate the congruence class of the set-associative TLB array. If the page size specified by MAS1_{TSIZE} is not supported by this implementation, an illegal instruction exception is generated. If MAS0_{ATSEL} = 0, and MAS1_{IND} = 1, and the page size and sub-page size combination contained in MAS1_{TSIZE} and MAS3_{SPSIZE} is not supported by this implementation, an illegal instruction exception is generated.

If MAS1_{IND} = 1 and the memory attributes specified by MAS2_{WIMAGE} are not consistent with those specified in *Section 6.16.6 Hardware Page Table Storage Control Attributes*, an illegal instruction exception is generated.

Note: The architecturally defined fields of MAS0.TLBSEL, MAS0.NV, MAS2.VLE, and MAS2.ACM are not included in the explanation of this instruction because they are reserved in this implementation.

12.2.3 TLB Search Indexed (tlbsx[.])

Software must use the **tlbsx[.]** instruction to search entries in the TLB (searching the LRAT is not supported in this implementation). This instruction is embedded hypervisor privileged. Execution of this instruction in guest state (GS = 1) results in an embedded hypervisor privilege exception.

Because this instruction relies on the MAS Registers, execution of this instruction in ERAT-only mode (CCR2[NOTLB] = 1) results in an illegal instruction exception. The instruction format and details follow.

tlbsx **RA, RB** **Rc = 0**
tlbsx. **RA, RB** **Rc = 1**

31	///	RA	RB	914	Rc
0	6	11	16	21	31

if RA = 0 then b ← 0 else b ← (RA)

EA ← b + (RB)

EPN ← EA(0:51)

pid ← MAS6_{SPID}

as ← MAS6_{SAS}

gs ← MAS5_{SGS}

lpid ← MAS5_{SLPID}

vpn ← gs || lpid || as || pid || EPN

thread_num ← number of executing thread (0 to 3)

if Rc = 1 then

CR_{CR0(0)} ← 0

CR_{CR0(1)} ← 0

CR_{CR0(3)} ← 0

Valid_matching_entry_exists ← 0

for each TLB entry

m ← ¬((1 << (2 X (entry_{SIZE} - 1))) - 1)

n ← 64 - log₂(page size in bytes)

if ((EA_{0:51} & m) = (entry_{EPN} & m)) &

(entry_{TLPID} = MAS5_{SLPID} | entry_{TLPID} = 0) & (entry_{TGS} =

MAS5_{SGS}) &

```

(entryIND = MAS1IND) &
num) = 1)
    (entryTID = MAS6SPID | entryTID = 0) & (entryTS = MAS6SAS) &
    (entryX = 0 | EPNn:51 > entryEPN[n:51]) & (entryTHDID(thread_
then
    Valid_matching_entry_exists ← 1
    exit for loop

if Valid_matching_entry_exists = 1 then
    entry ← matching entry found
    index ← index of TLB entry found (TLB way)
    rpn ← entryRPN
    MAS0ATSEL ← 0
    MAS0ESEL ← index
    MAS0HES ← TLB0CFGHES
    MAS0WQ ← 0b01
    MAS1v ← 1
    MAS1IPROT TID TS TSIZE ← entryIPROT TID TS SIZE
    MAS1IND ← entryIND
    MAS2EPN W I M G E ← entryEPN W I M G E

    if entryIND = 1
        MAS3SPSIZE0 SPSIZE1 SPSIZE2 SPSIZE3 SPSIZE4 UND ←
entrySPSIZE0 SPSIZE1 SPSIZE2 SPSIZE3 || 0 0
        MAS3RPNL[52] ← entryRPN[52]
    else
        MAS3UX SX UW SW UR SR ← entryUX SX UW SW UR SR
        MAS3RPNL[52] ← 0

        MAS3RPNL[32:51] ← rpn32:51
        MAS3U0:U3 ← entryU0:U3

```

```

MAS7RPNU ← rpn0:31

MAS8TGS VF TLPID ← entryTGS VF TLPID

MMUCR3X R C ECL TID_NZ Class WLC ResvAttr ThdID ← entryX R C
ExtClass TID_NZ Class WLC ResvAttr ThdID

if Rc = 1 then
    CRCR0(2) ← 1
else
    MAS0ATSEL ← 0
    MAS0ESEL ← 0
    MAS0HES ← TLB0CFGHES
    MAS0WQ ← 0b01
    MAS1V IPROT ← 0
    MAS1TID TS ← MAS6SPID SAS
    MAS1TSIZE ← MAS4TSIZED
    MAS1IND ← MAS4INDD
    MAS2W I M G E ← MAS4WD ID MD GD ED
    MAS2EPN ← unchanged
    MAS3RPNL ← 0
    MAS3U0:U3 UX SX UW SW UR SR ← 0
    MAS7RPNU ← 0
    if Rc = 1 then
        CRCR0(2) ← 0

```

An effective address (EA) is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0 and is the contents of register RA_{0:63} otherwise.

An effective page number (EPN) is determined from EA bits 0 to 51. The effective page number bits used for page matching for a given TLB entry is EPN[0:63-p], where p = log₂(entry page size in bytes). If the TLB array contains an entry corresponding to the virtual page number formed by MAS5_{SGS}, MAS5_{SLPID}, MAS6_{SAS}, MAS6_{SPID}, and EPN[0:63-p], and the entry's indirect bit (IND) matches that value in MAS6_{SIND}, that entry's contents and the index (the matching TLB way in this case) are read into the MAS and MMUCR3 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 (MMUCR3 is unchanged). See *Section 6.17.28 MAS Register Update Summary* on page 259 for a description of default values loaded into the MAS registers for this instruction. If more than one entry matches the search parameters, a machine check exception is generated.



The record bit (Rc) specifies whether the results of the search will affect CR[CR0] as shown above, such that CR[CR0]₂ can be tested if there is a possibility that the search might fail.

12.2.4 TLB Search and Reserve Indexed (tlbsrx.)

Software can use the **tlbsrx.** instruction to search for entries in the local TLB and, as a side-affect, sets a local TLB reservation for the associated virtual address.

Because the Embedded.Hypervisor category is supported, if guest execution of TLB management instructions is disabled ($EPCR_{DGTMI} = 1$), this instruction is embedded hypervisor privileged. Otherwise, this instruction is supervisor privileged. Because this instruction relies on the MAS Registers, execution of this instruction in ERAT-only mode ($CCR2[NOTLB] = 1$) results in an illegal instruction exception. The instruction format and details follow.

tlbsrx. **RA,RB** **Rc = 1**

31	///	RA	RB	850	1
0	6	11	16	21	31

if RA = 0 then b ← 0 else b ← (RA)

EA ← b + (RB)

EPN ← EA_{0:51}

pid ← MAS1_{TID}

as ← MAS1_{TS}

ind ← MAS1_{IND}

gs ← MAS5_{SGS}

lpid ← MAS5_{SLPID}

thread_num ← number of executing thread (0 to 3)

vpn ← gs || lpid || as || pid || EPN

TLB-RESERVATION_V ← 1

TLB-RESERVATION_{IND} ← ind

TLB-RESERVATION_{GS} ← gs

TLB-RESERVATION_{LPID} ← lpid

TLB-RESERVATION_{AS} ← as

TLB-RESERVATION_{PID} ← pid

TLB-RESERVATION_{EPN} ← EPN

Valid_matching_entry_exists ← 0

for each TLB entry

```

m ← ¬((1 << (2 X (entrySIZE - 1))) - 1)
n ← 64 - log2(page size in bytes)
if ((EA0:51 & m) = (entryEPN & m)) &
(MAS5SGS) &
(entryTLPID = MAS5SLPID | entryTLPID = 0) & (entryTGS =
MAS5SGS) &
(entryIND = MAS1IND) &
(entryTID = MAS1TID | entryTID = 0) & (entryTS = MAS1TS) &
(entryX = 0 | EPNn:51 > entryEPN[n:51]) & (entryTHDID(thread_ -
num) = 1)
then
Valid_matching_entry_exists ← 1
exit for loop
if Valid_matching_entry_exists = 1 then
CR0 ← 0b0010
else
CR0 ← 0b0000

```

Let the EA be the sum (RA|0) + (RB).

If the TLB array contains a valid entry matching the MAS1_{IND} and virtual address formed by MAS5_{SGS}, MAS5_{SLPID}, MAS1_{TS} TID, and EA, the search is considered successful. A TLB entry matches if all the following conditions are met:

- The valid bit of the TLB entry is 1.
- The IND value of the TLB entry is equal to MAS1_{IND}.
- The logical AND of EA_{0:53} and m is equal to the logical AND of the EPN value of the TLB entry and m, where m is equal to the logical NOT of ((1 << (2 × (entry_{SIZE}-1))) - 1).
- The X value of the TLB entry is 0, or EPN_{n:51} is greater than the value of the entry EPN_{n:51}, where n equals 64 - log₂(entry page size in bytes).
- The TID value of the TLB entry is equal to MAS1_{TID} or is zero.
- The TS value of the TLB entry is equal to MAS1_{TS}.
- The TGS value of the TLB entry is equal to MAS5_{SGS}
- The TLPID value of the TLB entry is equal to MAS5_{SLPID} or is zero.
- CR field 0 is set as follows: CR0_{LT GT EQ SO} = 0b00 || n || 0, where n is a 1-bit value that indicates whether the search was successful.

This instruction creates a TLB reservation for use by a **tlbwe** instruction. The virtual address described above is associated with the TLB reservation, and replaces any address previously associated with the TLB reservation.

If there are multiple matching TLB entries, a machine check exception occurs.

12.2.5 TLB Invalidate Virtual Address Indexed (tlbivax)

Software can use the **tlbivax** instruction to invalidate entries in the TLB (and associated copies in the ERATs). The **tlbivax** instruction pertains to all processors in the same logical partition (that is, this a “global” instruction), as opposed to the **tlbilx** instruction (which is a “local” instruction to this processor only). The global **tlbivax** instruction is broadcast to all processors in the system when the A2 is connected to an L2 memory subsystem with invalidation snoop capability. See *Section 6.9.4 TLB Invalidate Virtual Address (Indexed) Instruction (tlbivax)* for implementation-specific system requirements and parameters associated with the broadcast aspect of this instruction.

This instruction is embedded hypervisor privileged. Execution of this instruction in guest state (GS = 1) results in an embedded hypervisor privilege exception. Because this instruction relies on the MAS Registers, execution of this instruction in ERAT-only mode (CCR2[NOTLB] = 1) results in an illegal instruction exception.

tlbivax **RA,RB**

31	///	RA	RB	786	/
0	6	11	16	21	31

$EA \leftarrow (RA|0) + (RB)$

$EPN \leftarrow EA[0:51]$

$lpid \leftarrow MAS5[SLPID]$

$gs \leftarrow MAS5[SGS]$

$ts \leftarrow MAS6[SAS]$

$tid \leftarrow MAS6[SPID]$

$size \leftarrow MAS6[ISIZE]$

$ind \leftarrow MAS6[SIND]$

if size= ‘0001’ and ind=0 then pg_size ← 4 KB

else if size = ‘0011’ and ind = 0 then pg_size ← 64 KB

else if size = ‘0101’ then pg_size ← 1 MB

else if size = ‘0111’ and ind = 0 then pg_size ← 16 MB

else if size = ‘1001’ and ind = 1 then pg_size ← 256 MB

else if size = ‘1010’ and ind = 0 then pg_size ← 1 GB

else illegal instruction exception

$p \leftarrow \log_2(\text{pg_size})$

if $\text{pg_size} = 4 \text{ KB}$ then $L \leftarrow 0$ else $L \leftarrow 1$

$w \leftarrow$ Most significant bit position supported by this processor's physical system address bus (see *Section 6.9.4 TLB Invalidate Virtual Address (Indexed) Instruction (tlbivax)* for a description of w values for this implementation)

for each processor in the system

for each TLB entry

$n \leftarrow 64 - \log_2(\text{entry page size in bytes})$

if (entry[EPN_{w:63-p}] = EPN_{w:63-p}) AND

(entry[X] = 0 OR EPN_{n:51} > entry[EPN_{n:51}]) AND

(entry[TGS] = gs) AND

(entry[TLPID] = lpid) AND

(entry[TS] = ts) AND

(entry[TID] = tid) AND

(entry[SIZE] = size) AND

(entry[IND] = ind) AND

(entry[IPROT] = 0)

then entry[V] \leftarrow 0

for each ERAT entry

$n \leftarrow 64 - \log_2(\text{entry page size in bytes})$

if (entry[EPN_{31:63-p}] = EPN_{31:63-p}) AND

(entry[X] = 0 OR EPN_{n:51} > entry[EPN_{n:51}]) AND

(entry[TGS] = gs) AND

(entry[TS] = ts) AND

(entry[TID] = tid_{6:13}) AND

(entry[THDID] = tid_{2:5} OR MMUCR1[I/DTTID] = 0) AND

(entry[CLASS] = tid_{0:1} OR MMUCR1[I/DCTID] = 0) AND

(entry[SIZE] = convert_to_3bit(size)) AND

(entry[TID_NZ] = or_reduce(tid_{0:13})) AND

```

(ind = 0) AND
(entry[EXTCLASS] = 0)
then entry[V] ← 0
  
```

An EA is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0, and is the contents of register RA_{0:63} otherwise.

This implementation requires the target page size to be specified by MAS6_{ISIZE} (MMUCFG_{TWC} = 1 and TLB0CFG_{HES} = 1). For the T = 3 form, the target page size is used by the set-associative TLB structure in this implementation to calculate the one and only congruence class in which the targeted entry can be stored. The target page size is used by the fully associative ERAT structures to minimize generous invalidations that would otherwise occur when the full EPN is not transferred. If the page size specified by MAS6_{ISIZE} is not supported by this implementation, an illegal instruction exception is generated.

All TLB entries on all processors that have all of the following properties are made invalid. The MAS registers listed are those in the processor executing the **tlbivax**.

- The EPN_{w:63-p} value of the TLB entry is equal to EPN_{w:63-p}.
- The X value of the TLB entry is 0, or EPN_{n:51} is greater than the value of the entry EPN_{n:51}, where n equals 64 - log₂(entry page size in bytes).
- The TGS value of the TLB entry is equal to MAS5_{SGS}.
- The TLPID value of the TLB entry is equal to MAS5_{SLPID}.
- The TS value of the TLB entry is equal to MAS6_{SAS}.
- The TID value of the TLB entry is equal to MAS6_{SPID}.
- The SIZE value of the TLB entry is equal to MAS6_{ISIZE}.
- The IND value of the TLB entry is equal to MAS6_{SIND}.
- The IPROT value of the TLB entry is 0.

All shadow ERAT entries on all processors that have all of the following properties are made invalid. The MAS registers listed are those in the processor executing the **tlbivax**.

- The EPN_{31:63-p} value of the ERAT entry is equal to EPN_{31:63-p}.
- The X value of the ERAT entry is 0, or EPN_{n:51} is greater than the value of the entry EPN_{n:51}, where n equals 64 - log₂(entry page size in bytes).
- The TGS value of the ERAT entry is equal to MAS5_{SGS}.
- The TS value of the ERAT entry is equal to MAS6_{SAS}.
- The 8-bit TID value of the ERAT entry is equal to MAS6_{SPID}[6:13].
- Either the appropriate MMUCR1[I/DTTID] bit (for I-ERAT or D-ERAT) is 0, or the 4-bit ThdID value of the ERAT entry is equal to MAS6_{SPID}[2:5].
- Either the appropriate MMUCR1[I/DCTID] bit (for I-ERAT or D-ERAT) is 0, or the 2-bit Class value of the ERAT entry is equal to MAS6_{SPID}[0:1].
- The 3-bit SIZE value of the ERAT entry is equal to the 3-bit interpretation of the 4-bit MAS6_{ISIZE}.
- The TID_NZ bit of the ERAT entry is equal to the logical OR of all the bits of MAS6_{SPID}[0:13].
- The MAS6_{SIND} value is 0 (that is, ERATs contain only direct entries).

- The ExtClass value of the ERAT entry is 0.

12.2.6 TLB Invalidate Local Indexed (tlbilx)

Software can use the **tlbilx** instruction to invalidate entries in the local TLB (and associated copies in the local ERAT structures).

The “c” parameter (which architecturally can depend on $MMUCFG_{TWC}$) is defined always as $MAS6_{ISIZE}$ because $MMUCFG_{TWC} = 1$ for this processor.

Because the Embedded.Hypervisor category is supported, if guest execution of TLB management instructions is disabled ($EPCR_{DGTMI} = 1$), this instruction is embedded hypervisor privileged. Otherwise, this instruction is supervisor privileged. Because this instruction relies on the MAS Registers, execution of this instruction in ERAT-only mode ($CCR2[NOTLB] = 1$) results in an illegal instruction exception. The instruction format and details follow.

tlbilx **T,RA,RB**

31	///	T	RA	RB	18	/
0	6	8	11	16	21	31

if RA = 0 then b ← 0 else b ← (RA)

EA ← b + (RB)

for each TLB entry

```

c ← MAS6ISIZE
m ← ¬((1 << (2×(c-1))) - 1)
n ← 64 - log2(entry page size in bytes)
if (entryIPROT = 0) & (entryTLPID = MAS5SLPID) then
  if T = 0 then entryV ← 0
  if T = 1 & entryTID = MAS6SPID & (MAS6SIND = 0 | (MAS6SIND
= 1 & entryIND = 0))
    then entryV ← 0
    if T = 3 & entryTGS = MAS5SGS &
((EA0:51 & m) = (entryEPN & m)) &
(entryX = 0 | EPNn:51 > entryEPN[n:51]) & entrySIZE = MAS6ISIZE
&
entryTID = MAS6SPID & entryTS = MAS6SAS & entryIND =
MAS6SIND
    then entryV ← 0
    if T = 4 & entryCLASS = 0 & MMUCR1ICTID DCTID = 0 0 then

```

```

entryV ← 0

if T = 5 & entryCLASS = 1 & MMUCR1ICTID DCTID = 0 0 then
entryV ← 0

if T = 6 & entryCLASS = 2 & MMUCR1ICTID DCTID = 0 0 then
entryV ← 0

if T = 7 & entryCLASS = 3 & MMUCR1ICTID DCTID = 0 0 then
entryV ← 0

for each ERAT entry

c ← entrySIZE
m ← ¬((1 << (2×(c-1))) - 1)
n ← 64- $\log_2$ (entry page size in bytes)
if (entryEXTCLASS = 0) then
if T = 0 then entryV ← 0
if T = 1 & entryTID = MAS6SPID[6:13] & entryTID_NZ =
or_reduce(MAS6SPID[0:13]) &
(entryTHDID = MAS6SPID[2:5] | MMUCR1I/DTTID = 0) &
(entryCLASS = MAS6SPID[0:1] | MMUCR1I/DCTID = 0)
then entryV ← 0
if T = 3 & entryTGS = MAS5SGS &
((EA0:51 & m) = (entryEPN & m)) &
(entryX = 0 | EPNn:51 > entryEPN[n:51]) &
entryTID = MAS6SPID[6:13] & entryTS = MAS6SAS & MAS6IND =
0 & entryTID_NZ = or_reduce(MAS6SPID[0:13]) &
(entryTHDID = MAS6SPID[2:5] | MMUCR1I/DTTID = 0) &
(entryCLASS = MAS6SPID[0:1] | MMUCR1I/DCTID = 0)
then entryV ← 0
if T = 4 & entryCLASS = 0 & MMUCR1I/DCTID = 0 then entryV
← 0
if T = 5 & entryCLASS = 1 & MMUCR1I/DCTID = 0 then entryV ←
0
if T = 6 & entryCLASS = 2 & MMUCR1I/DCTID = 0 then entryV ←
0

```

if $T = 7$ & $\text{entry}_{\text{CLASS}} = 3$ & $\text{MMUCR1}_{\text{I/DCTID}} = 0$ then $\text{entry}_V \leftarrow$

0

Let the EA be the sum $(\text{RA}|0) + (\text{RB})$.

The **tlbilx** instruction invalidates TLB and ERAT entries in the processor (core) that executes the **tlbilx** instruction. TLB entries that are protected by the IPROT attribute ($\text{entry}_{\text{IPROT}} = 1$) are not invalidated. ERAT entries that are protected by the ExtClass attribute ($\text{entry}_{\text{EXTCLASS}} = 1$) are not invalidated.

If $T = 0$, all TLB (and ERAT) entries that have all of the following properties are made invalid on the processor (core) executing the **tlbilx** instruction:

- The TLPID of the entry matches $\text{MAS5}_{\text{SLPID}}$ (ERAT entries ignore $\text{MAS5}_{\text{SLPID}}$).
- The IPROT (or ExtClass) of the entry is 0.

If $T = 1$, all TLB (and ERAT) entries that have all of the following properties are made invalid on the processor executing the **tlbilx** instruction:

- The TLPID of the entry matches $\text{MAS5}_{\text{SLPID}}$ (ERAT entries ignore $\text{MAS5}_{\text{SLPID}}$).
- The TID of the entry (and perhaps the ThdID and/or Class of the ERAT entries, depending on $\text{MMUCR1}_{\text{ITTID DTTID ICTID DCTID}}$ bits) matches $\text{MAS6}_{\text{SPID}}$.
- The TID_NZ bit value of the ERAT entry (does not apply to TLB entries) matches the logical OR of all bits of $\text{MAS6}_{\text{SPID}(0:13)}$.
- The IPROT (or ExtClass) of the entry is 0.

If $T = 3$, all TLB entries (and except where noted, all ERAT entries) in the processor executing the **tlbilx** instruction that have all of the following properties are made invalid:

- The TLPID value of the entry is equal to $\text{MAS5}_{\text{SLPID}}$ (ERAT entries ignore $\text{MAS5}_{\text{SLPID}}$)
- The TGS value of the entry is equal to MAS5_{SGS} .
- The logical AND of $\text{EA}_{0:53}$ and m is equal to the logical AND of the EPN value of the entry and m , where m is based on the following:
 - c is equal $\text{MAS6}_{\text{ISIZE}}$.
 - m is equal to the logical NOT of $((1 \ll (c - 1)) - 1)$. Note this might seem in conflict with the architecture for MAV 2.0, but this implementation supports only the 4 MSbs of the SIZE fields (that is, this processor supports only power of 4, 1 KB page sizes).
- The X value of the entry is 0, or $\text{EPN}_{n:51}$ is greater than the value of the entry $\text{EPN}_{n:51}$, where n equals $64 - \log_2(\text{entry page size in bytes})$.
- The TID value of the entry (and perhaps the ThdID and/or Class values of the ERAT entries, depending on $\text{MMUCR1}_{\text{ITTID DTTID ICTID DCTID}}$ bits) is equal to $\text{MAS6}_{\text{SPID}}$.
- The TID_NZ bit value of the ERAT entry (does not apply to TLB entries) matches the logical OR of all bits of $\text{MAS6}_{\text{SPID}(0:13)}$.
- The TS value of the entry is equal to MAS6_{SAS} .
- For TLB entries, the SIZE value of the entry is equal to $\text{MAS6}_{\text{ISIZE}}$ (this does not apply to ERAT entries).
- The IND value of the TLB entry is equal to $\text{MAS6}_{\text{SIND}}$ (or $\text{MAS6}_{\text{SIND}} = 0$ for ERAT entries).
- The IPROT (or ExtClass) of the entry is 0.

If $T = 4, 5, 6,$ or 7 , all TLB (and ERAT) entries that have all of the following properties are made invalid on the processor executing the **tlbilx** instruction:

- The TLPID value of the entry is equal to $MAS5_{SLPID}$ (ERAT entries ignore $MAS5_{SLPID}$).
- The Class value of the entry equals $T - 4$.
- $MMUCR1_{ICTID} = 0$ (for I-ERAT entries) or $MMUCR1_{DCTID} = 0$ (for D-ERAT entries).
- The IPROT (or ExtClass) of the entry is 0.

The effects of the invalidation are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation.

If $T = 2$, an illegal instruction exception is generated.

If $T = 4, 5, 6,$ or 7 , and $MMUCR1_{ICTID} = 1$ and $MMUCR1_{DCTID} = 1$, an illegal instruction exception is generated.

This implementation requires the target page size to be specified by $MAS6_{ISIZE}$ (because $MMUCFG_{TWC} = 1$ and $TLB0CFG_{HES} = 1$). For the $T = 3$ form, the target page size is used by the set-associative TLB structure in this implementation to calculate the one and only congruence class in which the targeted entry can be stored. The target page size is not required by the fully associative ERAT structures because the full EPN is specified and generous invalidates are therefore inherently minimized. If $T = 3$ and the page size specified by $MAS6_{ISIZE}$ is not supported by this implementation, an illegal instruction exception is generated.

12.3 ERAT Management Instructions

12.3.1 ERAT Read Entry (eratre)

Software must use the **eratre** instruction to read entries from either ERAT. The **eratre** instruction relies on the MMUCR0[TLBSEL] to determine on which hardware ERAT structure (I-ERAT or D-ERAT) the instruction operates.

This instruction is embedded hypervisor privileged. This instruction can be executed in either MMU mode or ERAT-only mode (CCR2[NOTLB] = don't care). The instruction format and details follow.

eratre **RT,RA,WS**

0	31 6	RT 6	RA 11	WS 16	179 21	/ 31
---	---------	---------	----------	----------	-----------	---------

IF MMUCR0[TLBSEL] = 0 or 1 (reserved settings) THEN

illegal instruction exception

ELSE IF MMUCR0[TLBSEL] = 2 THEN

If WS = 0 then

$(RT) \leftarrow I-ERAT_{WS0}[(RA_{60:63})]$

$(MMUCR0[TGS]) \leftarrow I-ERAT_{WS0}[(RA_{60:63})].TGS$

$(MMUCR0[TS]) \leftarrow I-ERAT_{WS0}[(RA_{60:63})].TS$

$(MMUCR0[TID_{50:51}]) \leftarrow I-ERAT_{WS0}[(RA_{60:63})].CLASS$ when

MMUCR1[ICTID] = 1 else unchanged

$(MMUCR0[TID_{52:55}]) \leftarrow I-ERAT_{WS0}[(RA_{60:63})].THDID$ when

MMUCR1[ITTID] = 1 else unchanged

$(MMUCR0[TID_{56:63}]) \leftarrow I-ERAT_{WS0}[(RA_{60:63})].TID$

$(MMUCR0[TID_NZ]) \leftarrow I-ERAT_{WS0}[(RA_{60:63})].TID_NZ$

$(MMUCR0[ECL]) \leftarrow I-ERAT_{WS0}[(RA_{60:63})].EXTCLASS$

else If WS = 1 then

$(RT) \leftarrow I-ERAT_{WS1}[(RA_{60:63})]$

else If WS = 3 then

$(RT) \leftarrow I-ERAT$ round-robin pointer when MMUCR1[IRRE] = 1

else least recently used (LRU) index

ELSE IF MMUCR0[TLBSEL] = 3 THEN

if WS = 0

(RT) ← D-ERAT_{WS0}[(RA_{59:63})]

(MMUCR0[TGS]) ← D-ERAT_{WS0}[(RA_{59:63})].TGS

(MMUCR0[TS]) ← D-ERAT_{WS0}[(RA_{59:63})].TS

(MMUCR0[TID_{50:51}]) ← D-ERAT_{WS0}[(RA_{59:63})].CLASS when

MMUCR1[DCTID] = 1 else unchanged

(MMUCR0[TID_{52:55}]) ← D-ERAT_{WS0}[(RA_{59:63})].THDID when

MMUCR1[DTTID] = 1 else unchanged

(MMUCR0[TID_{56:63}]) ← D-ERAT_{WS0}[(RA_{59:63})].TID

(MMUCR0[TID_NZ]) ← D-ERAT_{WS0}[(RA_{59:63})].TID_NZ

(MMUCR0[ECL]) ← D-ERAT_{WS0}[(RA_{59:63})].EXTCLASS

else If WS = 1 then

(RT) ← D-ERAT_{WS1}[(RA_{59:63})]

else If WS = 3 then

(RT) ← D-ERAT round-robin pointer when MMUCR1[DRRE] =

1 else least recently used (LRU) index

The contents of the selected ERAT entry is placed into register RT (and possibly into MMUCR0[TGS, TS, TID, and ECL]).

MMUCR0[TLBSEL] is used as the source structure selection for this instruction: I-ERAT or D-ERAT (MMUCR0[TLBSEL] = 2 or 3 respectively; settings 0 and 1 are reserved).

Bits 52:63 of RA are used as an index into the I-ERAT or D-ERAT (up to 4096 entries). Bits 60:63 are used to select the I-ERAT entries, bits 59:63 select the D-ERAT entries, and bits 52:58 are reserved.

The WS field specifies which portion of the selected the entry (WS0 or WS1) is loaded into RT.

If WS is 0 (the RPN portion is being accessed), then the MMUCR0[TGS, TS, TID_{56:63}, TID_NZ, and ECL] fields are set to the values of the TGS, TS, TID, TID_NZ, and EXTCLASS fields from the entry. Otherwise, the MMUCR0 SPR is not affected. Also, the MMUCR1[ITTID] and [ICTID] bits for I-ERAT, and MMUCR1[DTTID] and [DCTID] bits for D-ERAT, play a role in updating the MMUCR0[TID_{50:55}] field. When MMUCR1[I/DTTID] = 1, then MMUCR0[TID_{52:55}] is set to the value of the THDID field of the chosen I-ERAT or D-ERAT entry. When MMUCR1[I/DCTID] = 1, then MMUCR0[TID_{50:51}] is set to the value of the CLASS field of the chosen I/D-ERAT entry.

If WS is 1, the RPN portion is returned, and if WS is 3, the ERAT least recently used (LRU) entry index is returned. Setting WS = 2 is reserved for the 32-bit subset.

The contents of RT after completion of this instruction are defined below.

If WS = 0 (EPN portion) and MMUCR0[TLBSEL] = 2 or 3 (I-ERAT or D-ERAT selected):

RT[0:51] ← EPN[0:51]

RT[52:53] ← Class[0:1] when MMUCR[I/DCTID] = 0 else “00”

RT[54] ← V

RT[55] ← X (exclusion enable)

supported else “0000”

RT[56:59] ← convert_to_4bits(SIZE[0:2]) when entry page size

“1111”

RT[60:63] ← ThdID[0:3] when MMUCR[I/DTTID] = 0 else

MMUCR0[TGS] ← TGS

MMUCR0[TS] ← TS

else unchanged

MMUCR0[TID_{50:51}] ← Class[0:1] when MMUCR[I/DCTID] = 1

1 else unchanged

MMUCR0[TID_{52:55}] ← ThdID[0:3] when MMUCR[I/DTTID] =

MMUCR0[TID_{56:63}] ← TID[0:7]

MMUCR0[TID_NZ] ← TID_NZ

MMUCR0[ECL] ← EXTCLASS

If WS = 1 (RPN portion) and MMUCR0[TLBSEL] = 2 or 3 (I-ERAT or D-ERAT selected):

RT[0:7] ← “00000000”

RT[8:9] ← WLC[0:1]

RT[10] ← ResvAttr

RT[11] ← ‘0’

RT[12:15] ← U[0:3]

RT[16] ← R

RT[17] ← C

RT[18:21] ← “0000”

RT[22:51] ← RPN[22:51]

RT[52:56] ← WIMGE

RT[57] ← VF

RT[58:59] ← UX,SX

$$RT[60:61] \leftarrow UW, SW$$

$$RT[62:63] \leftarrow UR, SR$$

If WS = 3 (LRU portion), MMUCR0[TLBSEL] = 2 or 3 (I-ERAT or D-ERAT selected), and MMUCR1[IRRE] = 0 for I-ERAT or MMUCR1[DRRE] = 0 for D-ERAT:

$$RT[0:51] \leftarrow \text{"00...0"}$$

$$RT[52:63] \leftarrow \text{Least Recently Used (LRU) entry index}^1$$

If WS = 3 (LRU portion), MMUCR0[TLBSEL] = 2 or 3 (I-ERAT or D-ERAT selected), and MMUCR1[IRRE] = 1 for I-ERAT or MMUCR1[DRRE] = 1 for D-ERAT:

$$RT[0:51] \leftarrow \text{"00...0"}$$

$$RT[52:63] \leftarrow \text{Round-robin entry pointer value}^{1, 2}$$
Notes:

1. For ERAT structures containing less than 4096 entries, unused MSBs return '0'.
2. See *Section 6.7.4 ERAT LRU Round-Robin Replacement Mode* for a description of the round-robin entry pointer operation.

12.3.2 ERAT Write Entry (eratwe)

Software must use the **eratwe** instruction to write entries into either ERAT. The **eratwe** instruction relies on the MMUCR0[TLBSEL] to determine on which hardware ERAT structure (I-ERAT or D-ERAT) the instruction operates.

This instruction is embedded hypervisor privileged. This instruction can be executed in either MMU mode or ERAT-only mode (CCR2[NOTLB] = don't care). The instruction format and details follow.

eratwe **RS,RA,WS**

31	RS	RA	WS	211	/
0	6	11	16	21	31

IF MMUCR0[TLBSEL] = 0 or 1 (reserved settings) THEN

illegal instruction exception

ELSE IF MMUCR0[TLBSEL] = 2 THEN

If MMUCR1.IRRE = 1 then

entry ← I-ERAT LRU round-robin pointer

I-ERAT LRU round-robin pointer ← [(I-ERAT LRU round-robin pointer + 1) mod (watermark + 1)]

else

entry ← RA_{60:63}

If WS = 0 then

I-ERAT_{WS0}[(entry)].EPN ← (RS_{0:51})

I-ERAT_{WS0}[(entry)].CLASS ← (MMUCR0[TID_{50:51}]) when

MMUCR1[ICTID] = 1 else (RS_{52:53})

I-ERAT_{WS0}[(entry)].V, X ← (RS_{54:55})

I-ERAT_{WS0}[(entry)].TSIZE ← convert_to_3bits(RS_{56:59})

I-ERAT_{WS0}[(entry)].THDID ← (MMUCR0[TID_{52:55}]) when

MMUCR1[ITTID] = 1 else (RS_{60:63})

I-ERAT_{WS0}[(entry)].TGS ← (MMUCR0[TGS])

I-ERAT_{WS0}[(entry)].TS ← (MMUCR0[TS])

I-ERAT_{WS0}[(entry)].TID ← (MMUCR0[TID_{56:63}])

I-ERAT_{WS0}[(entry)].TID_NZ ← (MMUCR0[TID_NZ])

I-ERAT_{WS0}[(entry)].EXTCLASS ← (MMUCR0[ECL])

```

I-ERATWS1[(entry)] ← (I-ERAT.RPNREG)
else if WS = 1 then
    I-ERAT.RPNREG ← (RS)
else if WS = 3 then
    I-ERAT.LRU-Watermark- ← (RS60:63)

ELSE IF MMUCR0[TLBSEL] = 3 THEN

    If MMUCR1[DRRE] = 1 then
        entry ← D-ERAT LRU round-robin pointer
        D-ERAT LRU round-robin pointer ← [(D-ERAT LRU round-
robin pointer + 1) mod (watermark + 1)]
    else
        entry ← RA59:63
    If WS = 0 then
        D-ERATWS0[(entry)].EPN ← (RS0:51)
        D-ERATWS0[(entry)].CLASS ← (MMUCR0[TID50:51]) when
MMUCR1[DCTID] = 1 else (RS52:53)

        D-ERATWS0[(entry)].V, X ← (RS54:55)
        D-ERATWS0[(entry)].TSIZE ← convert_to_3bits(RS56:59)
        D-ERATWS0[(entry)].THDID ← (MMUCR0[TID52:55]) when
MMUCR1[DTTID] = 1 else (RS60:63)

        D-ERATWS0[(entry)].TGS ← (MMUCR0[TGS])
        D-ERATWS0[(entry)].TS ← (MMUCR0[TS])
        D-ERATWS0[(entry)].TID ← (MMUCR0[TID56:63])
        D-ERATWS0[(entry)].TID_NZ ← (MMUCR0[TID_NZ])
        D-ERATWS0[(entry)].EXTCLASS ← (MMUCR0[ECL])
        D-ERATWS1[(entry)] ← (D-ERAT.RPNREG)
    else if WS = 1 then
        D-ERAT.RPNREG ← (RS)
    else if WS = 3 then
        D-ERAT.LRU-Watermark ← (RS59:63)

```

The contents of register RS (and possibly the contents of MMUCR0[TGS, TS, TID, TID_NZ, and ECL]) are placed into the selected ERAT entry.

Bits MMUCR0[TLBSEL] are used as the target structure selection for this instruction: I-ERAT or D-ERAT (MMUCR0[TLBSEL] = 2 or 3 respectively; settings 0 and 1 are reserved).

When MMUCR1[I/DRRE] = 0, bits 52:63 of RA are used as an index into the I-ERAT or D-ERAT (up to 4096 entries). Bits 60:63 are used to select the I-ERAT entries, bits 59:63 select the D-ERAT entries, and bits 52:58 are reserved. When MMUCR1[I/DRRE] = 1, the appropriate LRU round-robin pointer is used as the entry pointer, and subsequently incremented (when less than the current watermark value; otherwise, it rolls over to zero).

The WS field specifies which portion of the selected the entry (WS0 or WS1) is loaded into from RS. The ERAT structures in this implementation contain an intermediate holding register (RPNREG) for the WS = 1 (RPN) portion. This is done to provide for an atomic update of the entire entry at one time. When this instruction is executed with WS = 1, the contents of RS are actually written into the holding register. When this instruction is executed with WS = 0, the contents of RS and MMUCR0 are written into the WS0 portion, and the contents of the holding register are written into the WS1 portion of the entry.

If WS is 0 (the WS0 portion is being written), then the MMUCR0[TGS, TS, TID_{56:63}, TID_NZ, and ECL] fields are used to set the values of the TGS, TS, TID, TID_NZ, and EXTCLASS fields in the entry. Also, the MMUCR1[ITTID] and [ICTID] bits for I-ERAT, and MMUCR1[DTTID] and [DCTID] bits for D-ERAT, play a role in updating the entry THDID and CLASS fields. When MMUCR1[I/DTTID] = 1, then MMUCR0[TID_{52:55}] is used to update the value of the THDID field of the chosen I/D-ERAT entry. When MMUCR1[I/DCTID] = 1, then MMUCR0[TID_{50:51}] is used to update the value of the CLASS field of the chosen I/D-ERAT entry.

If WS is 3 and MMUCR0[TLBSEL] = 2 or 3, the ERAT LRU watermark value is updated with the contents of RS. The setting of WS = 2 is reserved for the 32-bit subset.

The contents of the entry after completion of this instruction are defined below.

If WS = 0 (WS0 portion) and MMUCR0[TLBSEL] = 2 or 3 (I-ERAT or D-ERAT selected):

```

EPN[0:51] ← RS[0:51]
Class[0:1] ← MMUCR0[TID50:51] when MMUCR1[I/DCTID] = 1 else RS[52:53]
V ← RS[54]
X (exclusion enable) ← RS[55]
SIZE[0:2] ← convert_to_3bits(RS[56:59])
ThdID[0:3] ← MMUCR0[TID52:55] when MMUCR1[I/DTTID] = 1 else RS[60:63]
TGS ← MMUCR0[TGS]
TS ← MMUCR0[TS]
TID[0:7] ← MMUCR0[TID56:63]
TID_NZ ← MMUCR0[TID_NZ]
EXTCLASS ← MMUCR0[ECL]

```

```

unused ← RPNREG[0:7]
WLC[0:1] ← RPNREG[8:9]
ResvAttr ← RPNREG[10]
unused ← RPNREG[11]
U[0:3] ← RPNREG[12:15]
R ← RPNREG[16]
C ← RPNREG[17]

```

unused ← RPNREG[18:21]
RPN[22:51] ← RPNREG[22:51]
WIMGE ← RPNREG[52:56]
VF ← RPNREG[57]
UX,SX ← RPNREG[58:59]
UW,SW ← RPNREG[60:61]
UR,SR ← RPNREG[62:63]

If WS = 1 (WS1 portion) and MMUCR0[TLBSEL] = 2 or 3 (I-ERAT or D-ERAT selected):
RPNREG[0:63] ← RS[0:63]

12.3.3 ERAT Search Indexed (eratsx[.])

Software must use the **eratsx[.]** instruction to search the entries in either ERAT. The **eratsx[.]** instruction relies on the MMUCR0[TLBSEL] field to determine on which hardware ERAT structure (I-ERAT or D-ERAT) the instruction operates.

This instruction is embedded hypervisor privileged. This instruction can be executed in either MMU mode or ERAT-only mode (CCR2[NOTLB] = don't care). The instruction format and details are described below.

eratsx **RT,RA,RB** **Rc = 0**
eratsx. **RT,RA,RB** **Rc = 1**

31	RT	RA	RB	147	Rc
0	6	11	16	21	31

$$EA \leftarrow (RA|0)^{1+} (RB)$$

$$EPN \leftarrow EA[0:63-p], \text{ where } p = \log_2(4^{\text{Entry}[\text{SIZE}]} \times 1 \text{ K})$$

$$\text{thread_num} \leftarrow \text{number of executing thread (0 to 3)}$$

If Rc = 1

$$CR[CR0]_0 \leftarrow 0$$

$$CR[CR0]_1 \leftarrow 0$$

$$CR[CR0]_3 \leftarrow 0$$

IF MMUCR0[TLBSEL] = 0 or 1 (reserved settings) THEN

Illegal Instruction exception

ELSE IF MMUCR0[TLBSEL] = 2 THEN

if exactly one valid, matching entry with all of the following prop-

erties:

1. entry[TGS] = MMUCR0[TGS]
2. entry[TS] = MMUCR0[TS]
3. entry[TID] = MMUCR0[TID56:63], or entry[TID_NZ] = 0
4. MMUCR1[ICTID] = 0, or entry[CLASS] = MMUCR0[TID50:51], or entry[TID_NZ] = 0
5. MMUCR1[ITTID] = 0, or entry[THDID] = MMUCR0[TID52:55], or entry[TID_NZ] = 0
6. MMUCR1[ITTID] = 1, or entry[THDID(thread_num)] = 1
7. entry[EPN0:63-p] = EPN0:63-p
8. entry[V] = 1

is in the I-ERAT then

$$(RT[52:63]) \leftarrow \text{index of matching I-ERAT entry}^2$$

$$(RT[50:51]) \leftarrow \text{“01”}$$

if Rc = 1 then CR[CR0]₂ ← 1

else
 if more than one valid, matching entry is in the I-ERAT then
 (RT[52:63]) ← index of first matching I-ERAT entry²
 (RT[50:51]) ← “11”
 if Rc = 1 then CR[CR0]₂ ← 1

else
 (RT[52:63]) ← undefined
 (RT[50:51]) ← “00”
 if Rc = 1 then CR[CR0]₂ ← 0

ELSE IF MMUCR0[TLBSEL] = 3 THEN

if exactly one valid, matching entry with all of the following properties:

1. entry[TGS] = MMUCR0[TGS]
2. entry[TS] = MMUCR0[TS]
3. entry[TID] = MMUCR0[TID56:63], or entry[TID_NZ] = 0
4. MMUCR1[DCTID] = 0, or entry[CLASS] = MMUCR0[TID50:51], or entry[TID_NZ] = 0
5. MMUCR1[DTTID] = 0, or entry[THDID] = MMUCR0[TID52:55], or entry[TID_NZ] = 0
6. MMUCR1[DTTID] = 1, or entry[THDID(thread_num)] = 1
7. entry[EPN0:63-p] = EPN0:63-p
8. entry[V] = 1

is in the D-ERAT then

(RT[52:63]) ← index of matching D-ERAT entry²
 (RT[50:51]) ← “01”
 if Rc = 1 then CR[CR0]₂ ← 1

else
 if more than one valid, matching entry is in the D-ERAT then
 (RT[52:63]) ← index of first matching D-ERAT entry²
 (RT[50:51]) ← “11”
 if Rc = 1 then CR[CR0]₂ ← 1

else
 (RT[52:63]) ← undefined
 (RT[50:51]) ← “00”
 if Rc = 1 then CR[CR0]₂ ← 0

Notes:

1. An EA is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0, and is the contents of register RA_{0:63} otherwise. Because of the pipelined nature of searches of the ERAT arrays, and because the effective address adder resides in the execution unit (not the instruction unit), only the RA = 0 variation of this instruction is supported when searching the I-ERAT array (that is, EA is always calculated as EA = 0 + (RB) when MMUCR0[TLBSEL] = 2).
2. For ERAT structures containing less than 4096 entries, unused MSBs return '0'.

An EPN is determined from EA bits 0 to 51. The effective page number bits used for page matching for a given ERAT entry are EPN[0:63-p], where $p = \log_2(4^{\text{Entry[SIZE]}} \times 1 \text{ K})$.

The MMUCR0[TLBSEL] bits are used to select a particular source structure (I-ERAT or D-ERAT).

The chosen ERAT is searched for a valid entry, which translates MMUCR0[TGS], MMUCR0[TS], MMUCR0[TID_{56:63}], and EPN (the ERATs do not contain the TLPID or IND values). Depending on the value of the MMUCR1[I/DCTID] and [I/DTTID] bits, the entry Class and/or ThdID fields can participate in the search as part of the TID value. If one or more matching entries are found, the index number of the first matching entry is returned in register RT. If no matching entries are found, the index number returned is undefined.

The RT bits 50:51 can be tested after an **eratsx[.]** instruction to determine if the search found exactly one matching entry (01), found more than one matching entry (11), or failed to find a matching entry (00).

The record bit (Rc) specifies whether the results of the search will affect CR[CR0] as shown above, such that CR[CR0]₂ can be tested if there is a possibility that the search might fail.

12.3.4 ERAT Invalidate Virtual Address Indexed (erativax)

Software must use the **erativax** instruction to globally invalidate entries in the ERATs while operating in ERAT-only mode (CCR2[NOTLB] = 1). The global **erativax** instruction is broadcast to all processors in the same logical partition when the A2 is connected to an L2 memory subsystem with invalidation snoop capability. See *Section 6.10.3 ERAT Invalidate Virtual Address (Indexed) Instruction (erativax)* for implementation-specific system requirements and parameters associated with the broadcast aspect of this instruction.

This instruction is embedded hypervisor privileged. Execution of the instruction while MSR[GS] = 1 results in an embedded hypervisor privilege exception.

This instruction can be executed in ERAT-only mode (CCR2[NOTLB] = 1). Execution of this instruction in MMU mode (CCR2[NOTLB] = 0) results in an illegal instruction exception. When specific, architected global invalidations are required in MMU mode, the **tlbivax** instruction is recommended.

erativax **RS,RA,RB**

31	RS	RA	RB	819	/
0	6	11	16	21	31

EA ← (RA[0] + (RB

EPN ← EA[0:51]

IS ← RS[56:57]

class ← RS[58:59]

size ← RS[60:63]

tgs ← MMUCR0[TGS]

ts ← MMUCR0[TS]

tid ← MMUCR0[TID_{50:63}]

lpid ← LPIDR[LPID]

if size = '0001' then pg_size ← 4 KB

else if size = '0011' then pg_size ← 64 KB

else if size = '0101' then pg_size ← 1 MB

else if size = '0111' then pg_size ← 16 MB

else if size = '1010' then pg_size ← 1 GB

else illegal instruction exception

$p \leftarrow \log_2(\text{pg_size})$

$L \leftarrow 0$ when $\text{pg_size} = 4 \text{ KB}$

else $L \leftarrow 1$

$w \leftarrow$ most significant bit position supported by this processor's physical system address bus (see *Section 6.10.3 ERAT Invalidate Virtual Address (Indexed) Instruction (eratvax)* on page 205 for a description of w values for this implementation)

for each processor in the logical partition
for each ERAT entry

$n \leftarrow 64 - \log_2(\text{entry page size in bytes})$

if $\{(IS = "11") \text{ AND}$

$(\text{entry}[EPN_{w:63-p}] = EPN_{w:63-p}) \text{ AND}$

$(\text{entry}[X] = 0 \text{ OR } EPN_{n:51} > \text{entry}[EPN_{n:51}]) \text{ AND}$

$(\text{entry}[TGS] = \text{gs}) \text{ AND}$

$(\text{entry}[TS] = \text{ts}) \text{ AND}$

$(\text{entry}[TID] = \text{tid}_{6:13}) \text{ AND}$

$(\text{entry}[THDID] = \text{tid}_{2:5} \text{ OR } \text{MMUCR1}[I/DTTID] = 0) \text{ AND}$

$(\text{entry}[CLASS] = \text{tid}_{0:1} \text{ OR } \text{MMUCR1}[I/DCTID] = 0) \text{ AND}$

$(\text{entry}[TID_NZ] = \text{or_reduce}(\text{tid}_{0:13})) \text{ AND}$

$(\text{entry}[SIZE] = \text{convert_to_3bit}(\text{size})) \text{ AND}$

$(\text{entry}[EXTCLASS] = 0)\} \text{ OR}$

$\{(IS = "10") \text{ AND}$

$(\text{entry}[CLASS] = \text{class} \text{ AND } \text{MMUCR1}[I/DCTID]_{\text{TARGET}} = 0)$

AND

$(\text{entry}[EXTCLASS] = 0)\} \text{ OR}$

$\{(IS = "01") \text{ AND}$

$(\text{entry}[TID] = \text{tid}_{6:13}) \text{ AND}$

$(\text{entry}[THDID] = \text{tid}_{2:5} \text{ OR } \text{MMUCR1}[I/DTTID]_{\text{TARGET}} = 0)$

AND

```

AND
    (entry[CLASS] = tid0:1 OR MMUCR1[I/DCTID]TARGET = 0)
    (entry[TID_NZ] = or_reduce(tid0:13)) AND
    (entry[EXTCLASS] = 0)} OR
    {(IS = "00") AND
    (entry[EXTCLASS] = 0)}
    then entry[V] ← 0

```

An effective address EA is formed by adding an index to a base address. The index is the contents of register RB. The base address is 0 if the RA field is 0, and is the contents of register RA_{0:63} otherwise.

This implementation requires a valid direct page size to be specified by RS_{60:63}. If the page size specified by RS_{60:63} is not supported by this implementation for direct pages, an illegal instruction exception is generated.

When IS = '11', all ERAT entries on all processors in the same logical partition that have all of the following properties are made invalid. The RS and MMUCR0 registers listed are those in the processor executing the **erativax** instruction. The MMUCR1 register listed is that in the processor receiving the **erativax** snoop.

- The EPN_{w:63-p} value of the ERAT entry is equal to EPN_{w:63-p}.
- The X value of the ERAT entry is 0, or EPN_{n:51} is greater than the value of the entry EPN_{n:51}, where n equals 64 - log₂(entry page size in bytes).
- The TGS value of the ERAT entry is equal to MMUCR0_{TGS}.
- The TS value of the ERAT entry is equal to MMUCR0_{TS}.
- The 8-bit TID value of the ERAT entry is equal to MMUCR0_{TID[56:63]}.
- Either the appropriate MMUCR1[I/DTTID] bit (for target I-ERAT or D-ERAT) is 0, or the 4-bit ThdID value of the ERAT entry is equal to MMUCR0_{TID[52:55]}.
- Either the appropriate MMUCR1[I/DCTID] bit (for target I-ERAT or D-ERAT) is 0, or the 2-bit Class value of the ERAT entry is equal to MMUCR0_{TID[50:51]}.
- The TID_NZ value of the ERAT entry is equal to or_reduce(MMUCR0_{TID[50:63]}).
- The 3-bit SIZE value of the ERAT entry is equal to the 3-bit interpretation of the 4-bit RS_{60:63}.
- The ExtClass value of the ERAT entry is 0.

This implementation requires the direct target page size to be specified by RS_{60:63}. For the IS = '11' form, the target page size is used by the fully associative ERAT structures to minimize generous invalidations that would otherwise occur when the full EPN is not transferred. If the direct page size specified by RS_{60:63} is not supported by this implementation, an illegal instruction exception is generated.

When IS = '10', all ERAT entries on all processors in the same logical partition that have all of the following properties are made invalid:

- The appropriate MMUCR1[I/DCTID] bit (for target I-ERAT or D-ERAT) is 0, and the 2-bit Class value of the ERAT entry is equal to RS_{58:59}.
- The ExtClass value of the ERAT entry is 0.

When IS = '01', all ERAT entries on all processors in the same logical partition that have all of the following properties are made invalid:

- The 8-bit TID value of the ERAT entry is equal to $MMUCR0_{TID[56:63]}$.
- Either the appropriate $MMUCR1[I/DTTID]$ bit (for target I-ERAT or D-ERAT) is 0, or the 4-bit ThdID value of the ERAT entry is equal to $MMUCR0_{TID[52:55]}$.
- Either the appropriate $MMUCR1[I/DCTID]$ bit (for target I-ERAT or D-ERAT) is 0, or the 2-bit Class value of the ERAT entry is equal to $MMUCR0_{TID[50:51]}$.
- The TID_NZ value of the ERAT entry is equal to $or_reduce(MMUCR0_{TID[50:63]})$.
- The ExtClass value of the ERAT entry is 0.

When IS = '00', all ERAT entries on all processors in the same logical partition that have an ExtClass value of 0 are made invalid.

12.3.5 ERAT Invalidate Local Indexed (eratilx)

Software can use the **eratilx** instruction to invalidate entries in the local processor's ERAT structures. The **eratilx** invalidations are not broadcast to other processors.

This instruction is embedded hypervisor privileged. This instruction can be executed in either MMU mode or ERAT-only mode (CCR2[NOTLB] = don't care). The instruction format and details follow.

eratilx **T,RA,RB**

31	///	T	RA	RB	51	/
0	6	8	11	16	21	31

if RA = 0 then b ← 0 else b ← (RA)

EA ← b + (RB)

for each ERAT entry

c ← entry_{SIZE}

m ← ¬((1 << (2×(c-1))) - 1)

n ← 64 - log₂(entry page size in bytes)

if (entry_{EXTCLASS} = 0) then

if T = 0 then entry_V ← 0

if T = 1 & entry_{TID} = MMUCR0_{TID[56:63]} & entry_{TID_NZ} =

or_reduce(MMUCR0_{TID[50:63]}) &

(entry_{THDID} = MMUCR0_{TID[52:55]} | MMUCR1_{IDTTID} = 0) &

(entry_{CLASS} = MMUCR0_{TID[50:51]} | MMUCR1_{IDCTID} = 0)

then entry_V ← 0

if T = 2 & entry_{TGS} = MMUCR0_{TGS} then entry_V ← 0

if T = 3 & entry_{TGS} = MMUCR0_{TGS} &

((EA_{0:51} & m) = (entry_{EPN} & m)) &

(entry_X = 0 | EPN_{n:51} > entry_{EPN[n:51]}) &

entry_{TID} = MMUCR0_{TID[56:63]} & entry_{TID_NZ} =

or_reduce(MMUCR0_{TID[50:63]}) & entry_{TS} = MMUCR0_{TS} &

(entry_{THDID} = MMUCR0_{TID[52:55]} | MMUCR1_{IDTTID} = 0) &

(entry_{CLASS} = MMUCR0_{TID[50:51]} | MMUCR1_{IDCTID} = 0)

```

then entryV ← 0
if T = 4 & entryCLASS = 0 & MMUCR1I/DCTID = 0 then entryV ←
← 0
0
if T = 5 & entryCLASS = 1 & MMUCR1I/DCTID = 0 then entryV ←
0
if T = 6 & entryCLASS = 2 & MMUCR1I/DCTID = 0 then entryV ←
0
if T = 7 & entryCLASS = 3 & MMUCR1I/DCTID = 0 then entryV ←
0

```

Let the EA be the sum (RA|0) + (RB).

The **eratilx** instruction invalidates ERAT entries in the processor (core) that executes the **eratilx** instruction. ERAT entries that are protected by the ExtClass attribute (entry_{EXTCLASS} = 1) are not invalidated.

If T = 0, all ERAT entries that have an ExtClass value of 0 are made invalid on the processor (core) executing the **eratilx** instruction:

If T = 1, all ERAT entries that have all of the following properties are made invalid on the processor executing the **eratilx** instruction:

- The TID of the entry (and perhaps the ThdID and/or Class of the ERAT entry, depending on MMUCR1_{ITTID DTTID ICTID DCTID} bits) matches MMUCR0_{TID}.
- The TID_NZ value of the entry matches the logical OR of all bits of MMUCR0_{TID(0:13)}.
- The ExtClass of the entry is 0.

If T = 2, all ERAT entries that have all of the following properties are made invalid on the processor executing the **eratilx** instruction:

- The TGS of the entry matches MMUCR0_{TGS}.
- The ExtClass of the entry is 0.

If T = 3, all ERAT entries in the processor executing the **eratilx** instruction that have all of the following properties are made invalid:

- The TGS value of the entry is equal to MMUCR0_{TGS}.
- The logical AND of EA_{0:53} and m is equal to the logical AND of the EPN value of the entry and m, where m is based on the following:
 - c is equal entry_{SIZE}.
 - m is equal to the logical NOT of ((1 << (2 × (c-1))) - 1). Note that this might seem in conflict with the architecture for MAV 2.0, but this implementation supports only the 4 MSBs of the SIZE fields (that is, this processor supports only power of 4 × 1 KB page sizes).
- The X value of the entry is 0, or EPN_{n:51} is greater than the value of the entry EPN_{n:51}, where n equals 64 - log₂(entry page size in bytes).
- The TID value of the entry (and perhaps the ThdID and/or Class values of the ERAT entries, depending on MMUCR1_{ITTID DTTID ICTID DCTID} bits) is equal to MMUCR0_{TID}.
- The TID_NZ value of the entry matches the logical OR of all bits of MMUCR0_{TID(0:13)}.

- The TS value of the entry is equal to $MMUCR0_{TS}$.
- The ExtClass of the entry is 0.

If $T = 4, 5, 6,$ or 7 , all ERAT entries that have all of the following properties are made invalid on the processor executing the **eratilx** instruction:

- The Class value of the entry equals $T - 4$.
- $MMUCR1_{ICTID} = 0$ (for I-ERAT entries) or $MMUCR1_{DCTID} = 0$ (for D-ERAT entries).
- The ExtClass of the entry is 0.

The effects of the invalidation are not guaranteed to be visible to the programming model until the completion of a context synchronizing operation.

If $T = 4, 5, 6,$ or 7 , and $MMUCR1_{ICTID} = 1$ and $MMUCR1_{DCTID} = 1$, an illegal instruction exception is generated.

12.4 Software Transactional Memory Instructions

Support is provided for three problem-state instructions. The following notation assumes the existence of an extra bit per hardware thread per memory location X denoted watchbit(X), which is initialized to 0. An additional sticky bit per thread is maintained denoted watchlost, which reflects whether any watches have been lost since that thread last reset its watchlost bit. In the following descriptions, references to watchbit(X) and watchlost refer to the bits associated with the thread executing the instruction. The watch bits and watchlost bit associated with other threads in the system are unaffected.

The watch granule is 64 bytes, the same as the cache line size.

The watch  is supported for coherent, cacheable, nonguarded, nonwrite through memory (that is, for pages with `WIMG = 0b0010`). Its behavior for other types of storage is boundedly undefined. In the case that a cache-inhibited **ldawx.** hit in the L1 data cache, this might result in an invalid CR update. The watchlost bit associated for a thread will not be set for any cache-inhibited **ldawx.** executions.

For verification purposes, the A2 core treats the WIMG bits as follows.

M and W bits are completely ignored.

I and G bits:



- 00 - Good behavior as defined by the [STM](#) instruction.
- 01 - Same as 00.
- 10 - Load executes as I = 1, no watch bit is set, CR is updated.
- 11 - Same as 10, but this instruction waits for any previous G = 1 load to complete.

12.4.1 Load Doubleword and Watch Indexed X-Form (ldawx.)

ldawx. RT,RA,RB

31	RT	RA	RB	212	1
0	6	11	16	21	31

if RA = 0 then b ← 0

else b ← (RA)

EA ← b + (RB)

if EA.watchbit = 0

then

CR0 ← 0b00 || 0b0 || XERSO

EA.watchbit ← 1

else

CR0 ← 0b00 || 0b1 || XERSO

RT ← MEM(EA, 8)

Let the EA be the sum (RA|0) + (RB).

If the watch bit associated with address EA is not set, the bit is set to 1. CR field 0 is set to reflect whether the watch bit was already set for the referenced block as follows.

CR0LT GT EQ SO = 0b00 || watch bit already set || XERSO

Hardware guarantees that the reading of the watchbit(EA) and MEM(EA, 8) and setting of watchbit(EA) are performed as a single atomic operation with respect to operations performed by other threads.

EA must be a multiple of 8. If it is not, an alignment interrupt occurs.

This instruction is treated as a load.

Special Registers Altered

CR0

12.4.2 Watch Check All X-Form (wchkall)

wchkall **BF**

0	31	BF	//	///	///	902	/
		6	9	11	16	21	31

This instruction probes the watch monitoring facility, which maintains a watchlost sticky bit, to check whether any watches have been lost, due to invalidation or capacity reasons, since the watchlost bit was previously set to 0 via **wclr**. CR is updated as follows:

$$CR4xBF + 32: 4xBF + 35 = 0b00 \parallel \text{watchlost sticky bit} \parallel XERSO$$

watchlost stickybit = 0 indicates no watches lost.

watchlost stickybit = 1 indicates at least one watch might have been lost.

Special Registers Altered

CR field BF

Programming Note:

wchkall serves as both a basic and an extended mnemonic. The assembler recognizes a **wchkall** mnemonic with one operand as the basic form and a **wchkall** mnemonic with no operand as the extended form. In the extended form, the BF operand is omitted and assumed to be 0.

Programming Note:

Because a single watch can be checked using the **ldawx** instruction, a variant of **wchkall** that checks the watch status of a single block is not provided.

Programming Note:

When unmapping a physical page, the operating system must ensure that all pre-existing watches on the page are cleared by performing a sequence of watch clearing operations to the blocks within that page (for example, via **dcbi** or **dcbz** instruction) and a **wclr** (to clear those watches for the local hardware thread).

A **ldawx** by a processor P1 is performed with respect to any processor or mechanism P2 when the value and watchbit to be returned by the **ldawx** can no longer be changed by an operation by P2. A **wchkall** instruction by P1 is performed with respect to P2 when an operation by P2 can no longer affect the state of any watches summarized by the **wchkall** condition value.

Programming Note:

A **wclrral** L operation is not performed with respect to the processor executing the **wclrral** instruction until a subsequent **isync** instruction has been executed by that processor.

12.5 Coprocessor Instructions

A coprocessor is not a standard processor, but instead is a specialized processor that is capable of one or more particular tasks with the intent to provide acceleration of each task that might have otherwise been done by the program. For example, cryptographic functions can be performed by a coprocessor and are commonly known as clear-key functions. Depending upon configuration controls, a coprocessor is generally available to be initiated from any standard processor.

Each standard processor does not necessarily have its own suite of coprocessors. The system design is to provide overall platform acceleration by allowing a coprocessor to be invoked from where the application need arises.

Initiation of a coprocessor begins an asynchronous processing of the requested function. Upon completion, the completion and any exception status is signaled to the initiating program. This latter topic is covered in coprocessor architecture, which is outside the scope of this document.

A coprocessor is located outside of any standard processor, but is within the same coherence domain of an invoking, standard processor.

Part of the mechanism for a standard processor to initiate a function performed by a coprocessor involves a storage interface where a 64-byte control block is effectively pushed to a selected coprocessor. The particular coprocessor is identified using a coprocessor instance (CI) and can be explicitly specified with a CI value or is otherwise determined by an implementation-dependent default selection means.

The set of all coprocessors is first subdivided into categories where a category is identified as a coprocessor type (CT) and represents a related set of functions that a coprocessor of the CT recognizes. Then, within each CT that is provided, one or more CI values identify the coprocessor instances within the category, any of which can perform the same set of functions. A CT is an integer value in the range 0:63. The actual assignment of each CT value is implementation dependent and is configurable. A representative set of CTs that might or might not be provided in a given implementation include the following:

- Symmetric cryptographic functions
- Asymmetric cryptographic functions
- Asynchronous data copy
- Random-number generation
- Compression/decompression

- Regular-expression search
- Others

The instruction that initiates a coprocessor is normally a problem-state instruction. However, the definition also provides a higher-privileged instruction to assist a privileged-state or hypervisor-state program with the ability to logically re-issue the same instruction that was issued by the lower-privileged program, on behalf of that program. This avoids the higher-privileged program having to enter the context of the lower-privileged program.



Access to each CT is controlled by a privileged program. The control used is the ACOP special-purpose register, which has bit positions 0:63 defined. If $ACOP_{CT}$ is 1 when a problem-state program attempts to initiate a coprocessor of type CT, permission has been granted by the supervisor-state program. See *Available-Coprocessor* on page 516. Neither a hypervisor-state nor a privileged-state program is subject to ACOP control.



Similarly, a hypervisor state control is provided by the HACOP special-purpose register. If $HACOP_{CT}$ is 1 when a problem-or supervisor-state program attempts to initiate a coprocessor of type CT, permission has been granted by the hypervisor. A hypervisor-state program is not subject to HACOP control.

When an initiated function request is made to a coprocessor, the initiating instruction can include the ability to set CR0 to indicate status of the initiation. This is not indicative of the completion of the function, but only whether or not it has been accepted.

12.5.1 Initiate Coprocessor Store Word Indexed (icswx[.])

Initiation of a coprocessor is requested by issuing the *Initiate Coprocessor Store Word Indexed (icswx)* instruction.

Initiate Coprocessor Store Word Indexed X-form

icswx **RS,RA,RB** **(Rc = 0)**
icswx. **RS,RA,RB** **(Rc = 1)**

	31	RS	RA	RB	406	Rc
0	6	11	16	21		31

; Determine Processor State, PID, and LPID

hv ← ¬MSR_{GS} MSR_{GS}
pr ← MSR_{PR} MSR_{PR}
dr ← MSR_{DS} MSR_{DS}

ps ← hv||pr||dr||00000

pid ← PID_{32:63} My PID
lpid ← LPIDR_{32:63} My LPAR ID

; Determine Address of CRB

if (RA == 0) then Fulfill (RA|0)
 b ← 0 Value of 0 Since R0
else RA is R1:R31
 b ← (RA) Effective Addr in b
EA ← b + (RB) Calc EA of CRB

; Determine CT and CCW per Endian State

if (TLB_E(EA) == 0) then If Big Endian (BE)
 ct ← RS_{42:47} Get CT
 cdm ← RS_{48:55} Get MSB CD
 cdl ← RS_{56:63} Get LSB CD

$ccw_{0:31} \leftarrow ps 00 ct cdm cdl$	Form BE CCW
else	Is Little Endian (LE)
$ct \leftarrow RS_{50:55}$	Get CT
$cdm \leftarrow RS_{40:47}$	Get MSB CD
$cdl \leftarrow RS_{32:39}$	Get LSB CD
$ccw_{0:31} \leftarrow cdl cdm 00 ct ps$	Form LE CCW

; Check if HACOP & ACOP Permit CT

if ($ps_{0:1} \neq 0b10$) then	If Not Hypervisor
if ($HACOP_{ct} == 0$)	If Hyp Precludes, Or
($ps_1 == 1$) &	If Problem State and
($ACOP_{ct} == 0$) then	If OS Precludes
$ESR_{49} \leftarrow 1$	UCT
Data-Storage Interrupt	

; Store CCW into CRB Bytes 0:3

$MEM(EA,4) \leftarrow ccw_{0:31}$

; Signal Coprocessor

signal (

$MEM(b,64), CRB 0:63$

pid,

lpid)

; Set CR0 If Necessary

if ($Rc == 1$) then	If Setting CR0
if (available) then	
$CR0 \leftarrow 0b1000$	Initiated or Negative
elseif (busy) then	

CR0 ← 0b0100 Busy or Positive

elseif (no match on CT & FRC) then

CT  0b0  Reject

The PID and LPID values are normalized to 32-bit values. The *signal* internal function is a coarse abstraction of the relationship between the issuing processor and the coprocessor.

The coprocessor-command word (CCW) consists of the machine-state byte appended with the three least-significant bytes of general register RS (see *Figure 12-2 Coprocessor Command Word (CCW)* on page 512). The CCW is volatile in that, when used, a fresh instance of the CCW is determined during execution of **icswx** and then forgotten.

Unless stated otherwise, the CCW is stored in word 0 of the CRB (see *Figure 12-3 Generic Coprocessor-Request Block* on page 514).

A side effect of **icswx** execution is that an attempt is made to initiate a coprocessor by sending the CCW and the 64-byte CRB block to a coprocessor whose coprocessor type is CT and that provides support for execution of the CD specified.

Regardless of the condition set at the completion of **icswx** execution, the store of CCW into the CRB will have been performed.

The **icswx** instruction is not subject to exceptions related to storage-access WIMG bits.

The **icswx** instruction is a problem-state instruction.

12.5.1.1 General Registers

RS

General register RS contains multiple fields and bits that are illustrated in *Figure 12-1* and defined as follows. When applicable, the correspondence to the coprocessor-command word (CCW) is also indicated.

When little endian is in effect for the storage containing the CRB, and therefore the CCW, the format of RS is byte-reversed.

Figure 3. ICSWX (RS_{32:63}) Coprocessor-Command Word

Reserved	/	/	CT	CD
32	40	41	42	48 63

RSBits

Definition

0:31 RS bits 0:31 (not illustrated) are reserved.

32:39 RS bits 32:39 are reserved as a placeholder for the processor state (PS). CCW_{0:7} are based upon the required processor state, not RS.

40:41 Reserved.

- 42:47 RS bits 42:47 ($CCW_{10:15}$) are a 6-bit unsigned integer whose value specifies a coprocessor type (CT). A full-broadcast CT value specifies that all coprocessor types are designated. A specific coprocessor CT specifies that a particular coprocessor type is designated.
- 48:63 RS bits 48:63 ($CCW_{16:31}$) are a 16-bit unsigned integer whose value specifies the coprocessor directive (CD) that the coprocessor is requested to perform. See the specific chip or project hardware reference manual for details on supported coprocessor directives for that project.

RA and RB

The address of a coprocessor-request block (CRB) is located on a 128-byte boundary; otherwise, an alignment interrupt is recognized.

Before signaling a coprocessor, the **icswx** instruction performs storage write-access checks on the location specified by $(RA|0) + (RB)$. A data-storage interrupt can be recognized.

12.5.1.2 Initial Execution

Let the EA of the coprocessor-request block (CRB) be the sum $(RA|0) + (RB)$. A data-storage interrupt is recognized for a translation exception.

12.5.2 Initiate Coprocessor Store Word External Process ID Indexed (icswepx[.])

Initiation of a coprocessor is requested by issuing the *Initiate Coprocessor Store Word External Process ID Indexed (icswepx)* instruction. The **icswepx** instruction is identical to the **icswx** instruction except that it obtains identification of the issuing process from the EPSC SPR.

Initiate Coprocessor Store Word External Process ID Indexed X-form

icswepx **RS,RA,RB** **(Rc = 0)**
icswepx. **RS,RA,RB** **(Rc = 1)**

31	RS	RA	RB	950	Rc
0	6	11	16	21	31

This instruction behaves identically to an **icswx** instruction except for using the EPSC register substitutions. See *Initiate Coprocessor Store Word Indexed (icswx[.])* on page 509.

The **icswepx** instruction is a privileged-state instruction.

For **icswepx**, the following substitutions are made:

- EPSC_{EPR} is used in place of MSR_{PR}.
- EPSC_{EGS} is used in place of MSR_{GS}.
- EPSC_{EAS} is used in place of MSR_{DS}.
- EPSC_{EPIID} is used in place of PID.
- EPSC_{ELPID} is used in place of LPIDR.

12.5.3 Execution

The coprocessor type (CT) specified must be enabled. To be enabled, ACOP_{CT} must be 1 and, when implemented, HACOP_{CT} must be 1. When not enabled, an unavailable-coprocessor-type (UCT) exception is recognized, and:

- No access of storage occurs.
- A data-storage interrupt is recognized.
- Storage protection remains effective.

Execution of **icswx** forms a coprocessor-command word (CCW) that is stored into bytes 0:3 of the CRB. The CCW format is illustrated in *Figure 12-2*.

Figure 4. Coprocessor Command Word (CCW)

PS	CT	Coprocessor Directive
0	8	16 31

Bits 0:7 (PS) contains MSR[^]_{GS PR AS} || 0000. Bits 8:15 (CT) and 16:31 (CD) are obtained from the defined general-register fields.

Execution of **icswx** attempts to initiate a coprocessor by presenting the 64-byte CRB to a coprocessor designated by the coprocessor type (CT) and coprocessor directive (CD).

If the type of coprocessor specified by CT is enabled, when **icswx** completes, the condition is set in CR0 (see *Condition Register 0* on page 513).

After successfully initiated (CR0 bit 0 is 1), execution of a function completes asynchronously. See the coprocessor architecture for details.

Programming Note: The nature of the execution of the **icswx** instructions is such that the additional, processor-state information acquired and associated with a CRB is made available only to a coprocessor. It is not possible for any processor to alter the values sent to a coprocessor that were current when the **icswx** instruction was issued.

Programming Note: During execution of an **icswx** instruction, additional processor state information is acquired and communicated to the coprocessor. The additional processor state information consists of the following:

- PID Process identification is used to help identify the issuing program.
- LPID When operating in the logically-partitioned mode, the logical-partition identification is also used to help identify the issuing program.
- PS $MSR_{AGS_PR_AS}$ are used and obtained during the formation of a CCW to inform a coprocessor about address-translation and issuer-privilege matters.

12.5.3.1 Condition Register 0

Condition-register field 0 (CR0) is set to reflect the result of the **icswx**. instruction execution as follows:

Bit	Description
0	Initiated
1	Busy
2	Reject
3	Undefined

Exactly one bit of CR0_{0:2} is set to 1.

When bit 0 of CR0 is 1, the request has been initiated but not necessarily completed. After successfully initiated, execution of a function completes asynchronously. See the coprocessor architecture for details.

When bit 1 of CR0 is 1, the request is not initiated because no coprocessor of the specified CT is idle.

When bit 2 of CR0 is 1, the request is not initiated because no coprocessor of the specified coprocessor type is available.

Bit 3 of CR0 is set to 0.

12.5.4 Coprocessor-Request Block

A coprocessor-request block (CRB) must be located on a 128-byte boundary; otherwise, the **icswx** instruction specifying such an unaligned CRB recognizes an alignment interrupt.

A CRB is, at most, 64 bytes long. The definition of the contents of a CRB depends upon the coprocessor type and coprocessor directive that is specified by the **icswx** instruction. See the coprocessor architecture for details.

If the implementation forms a CCW, storing it into bytes 0:3 of the CRB, execution of an **icswx** instruction is subject to a storage-protection data storage interrupt.

A CRB is not subject to exceptions related to storage-access WIMG bits.

Figure 5. Generic Coprocessor-Request Block



Byte Meaning

- 0:3 Implementation-dependent CCW.
- 4:63 Bytes 0:63, the data field, are defined by each coprocessor type.

Programming Note: When little endian is in effect for the storage containing a CRB, byte-reversal stores must be performed in setting up a CRB data field such that the big-endian definition in the CRB contents is maintained. This includes setup by the program of any byte-reversed integer value whose length is a power of 2 and is normally subject to reversal.

12.5.4.1 Available Coprocessor Register (ACOP)

The ACOP is a 64-bit register. Available Coprocessor Register bits are numbered 0 (most-significant) to 63 (least-significant). The Available Coprocessor Register provides a 64-bit mask where a bit position corresponds to a coprocessor type. When a bit position is one, at least one coprocessor of that coprocessor type might be available. When a bit position is zero, no coprocessor of that coprocessor type is available.

The Available Coprocessor Register can be read using **mfspr** and can be written using **mtspr**. Only the least-significant 32 bits of the Available Coprocessor Register are implemented. The most-significant 32 bits of the Available-Coprocessor Register are treated as reserved. Each thread has an ACOP register.

When **icswx** is issued by a program in hypervisor or privileged state, ACOP checking does not apply. When **icswx** is issued by a program in problem state, ACOP checking applies. When ACOP checking applies and fails, an unavailable coprocessor type (UCT) exception is recognized and a DSI occurs.

Register Short Name:	ACOP	Read Access:	Priv
Decimal SPR Number:	31	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>

Bit(s):	Field Name:	Init	Description
32:63	CT	0x0	<p><u>Coprocessor Type</u></p> <p>Indicates available coprocessor types for the icswx instruction. Bit n of the register indicates availability of coprocessor type n.</p> <p>0 Coprocessor unavailable. Accesses will generate an unavailable coprocessor type data storage interrupt.</p> <p>1 Coprocessor available.</p>

1. When any bit for a specific coprocessor type (CT) is set to 1, the bit position for the full-broadcast CT must also be set to 1 to also enable the broadcast coprocessor type; otherwise, any request that specifies the broadcast CT fails.
2. Before **mtspr** to ACOP, the program must issue **sync** with L = 0 (also known as a heavyweight) or otherwise ensure that there is no **icswx** instruction that has not yet completed.
3. A problem-state process causes a DSI on issue of the **icswx** instruction when bit position CT in the ACOP is zero. Notwithstanding that DSI, because each thread has an ACOP SPR, the same process can cause another DSI for a subsequent issue of the **icswx** instruction on a different hardware thread.

12.5.4.2 Hypervisor Available Coprocessor Register (HACOP)

The HACOP is a 64-bit register. Hypervisor Available Coprocessor Register bits are numbered 0 (most-significant) to 63 (least-significant). The Hypervisor Available Coprocessor Register provides a 64-bit mask where a bit position corresponds to a coprocessor type. When a bit position is 1, at least one coprocessor of that coprocessor type might be available. When a bit position is zero, no coprocessor of that coprocessor type is available.

The Hypervisor Available Coprocessor Register can be read using **mfspr** and can be written using **mtspr**. Only the least-significant 32 bits of the Hypervisor Available Coprocessor Register are implemented. The most-significant 32 bits of the Hypervisor Available Coprocessor Register are treated as reserved. Each logical partition has an HACOP register. When the **icswx** instruction is executed, a fresh result of the effective mask is generated by a bitwise AND of ACOP with HACOP. The resulting mask is used for the current execution of the **icswx** instruction and not remembered.

When **icswx** is issued by a program in hypervisor state, HACOP checking does not apply. When **icswx** is issued by a program in privileged state, HACOP checking applies. When **icswx** is issued by a program in problem state, HACOP checking applies. When HACOP checking applies and fails, an unavailable coprocessor type (UCT) exception is recognized and a DSI occurs.

The reset state of HACOP is ⁶⁴0.

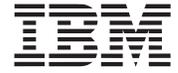
Register Short Name:	HACOP	Read Access:	Priv
Decimal SPR Number:	351	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>

Bit(s):	Field Name:	Init	Description
32:63	CT	0x0	<p><u>Coprocessor Type</u></p> <p>Indicates available coprocessor types for the icswx instruction. Bit n of the register indicates availability of coprocessor type n.</p> <p>0 Coprocessor unavailable. Accesses will generate an unavailable coprocessor type data storage interrupt.</p> <p>1 Coprocessor available.</p>

Programming Notes:

1. When any bit for a regular coprocessor type (CT) is set to 1, the bit position for the broadcast CT must also be set to 1 to enable the broadcast coprocessor type; otherwise, any request that specifies the broadcast CT fails.
2. Before an **mtspr** to HACOP, the program must issue **sync** with L = 0 (also known as a heavyweight sync) or otherwise ensure that there are no **icswx** instructions that have not yet completed.
3. A process without hypervisor privilege causes a DSI on issue of the **icswx** instruction when bit position CT in HACOP is zero.



13. Power Management Methods

13.1 Power-savings methods

The A2O core supports different levels of power-savings. Some clock gating actions occur automatically, while others involve varying amounts of hardware and software interaction. Power management logic and clock controls external to the core can force a thread to a stopped state, and along with software support bring about deeper levels of power savings. This section summarizes A2O power-saving methods:

- **local clock gating** - clocking to unused logic can be blocked through the ACT pin of their local clock buffers. This can be done statically through mode controls (i.e. debug mode), as well as dynamically by holding ACT inactive until the latch is being written.
- **external input controls** - two input signals, *an_ac_pm_thread_stop* and *an_ac_pm_fetch_halt*, are available for power-management control over individual threads. These per-thread inputs can be used to reduce core power in response to a thermal fault, or for some other power savings function. The level of power savings from these signals would be the same as using the ACT control to gate off unused logic.

an_ac_pm_thread_stop forces instruction execution to stop for the designated thread; any instructions in the pipeline will be flushed. Instructions at the completion point, as well as commands consisting of microcoded instruction sequences, will continue to completion before the thread is stopped. Thread stop status is available through the *ac_an_pm_thread_running* output signal, as well as by monitoring the THRCTL[Tx_Run] bits. Execution continues at the next instruction address upon deactivation of this signal.

an_ac_pm_fetch_halt blocks new instructions from being fetched; any active instructions will continue to completion. This signal does not provide thread stop status, either through the THRCTL[Tx_Run] bits or the *ac_an_pm_thread_running* output signal.

When using either of these inputs to throttle the core, enough time must be allowed for instruction execution to quiesce before any actual power savings will occur. While *an_ac_pm_thread_stop* should empty the pipeline faster due to flushing instructions, microcoded and long latency instructions could still require a significant number of cycles to complete. Since the flushed instructions will be reissued once *an_ac_pm_thread_stop* is released, extra cycles will be required to repeat these instructions. Also, if the throttling cycle is too fast a long latency instruction could end up being repeatedly flushed, thereby blocking forward progress on that thread. The *an_ac_pm_fetch_halt* input, while potentially requiring more time to quiesce the core, has the advantage of completing active instructions instead of having to reissue flushed instructions each time it is activated.

- **software control** - power-management controls in CCR0 and CCR1 allow software to configure two power-savings options (PM_Sleep and PM_RVW), along with wake up conditions that will enable the core to continue running. While the core takes the same power-savings actions for both options, PM_RVW along with chip level actions can be used to completely stop core clocking and power. See *Section 13.2 Power-savings instructions*.
- **chip level clocking and power** - after software has put all cores in the same clock region into the PM_RVW power-savings state, chip level controls can stop all clock inputs and turn off power to the core. A power-on reset is required if power is dropped or clock synchronization lost.

13.2 Power-savings instructions

Software can induce power-saving actions through the use of core level power management (CCR0[PME]) controls in combination with CCR0[WE]. The CCR0[WE] bits can be set through the **wait** instruction, or by a **mtspr** CCR0.

The CCR0[WE] bits can be used to disable a thread. Once a thread is stopped the *ac_an_pm_thread_running* output signal for that thread is driven to 0, indicating the thread is stopped. When stopped the thread will shut off unneeded logic using the ACT pin. This is the most basic level of power savings and is independent of CCR0[PME] settings. Requests from the A2/L2 interface (Snoop invalidate and TLB invalidate) are still handled as normal, and the state of all processor resources is maintained.

In order to enable core level power management options, the Power Management Enable field of the CCR0 register must be initialized. CCR0[PME] is a 2 bit field that allows software to select one of the three power states available in the A2 core: Running, PM_Sleep and PM_RVW.

1. Running State
 - CCR0[PME] = Disabled
 - This state is independent of the setting of CCR0[WE] bits.
2. PM_Sleep
 - CCR0[PME] = PM_Sleep_enable, and CCR0[WE] bits for all threads are set.
 - A2 activates run tholds to stop clocks for power savings. Requests from the A2/L2 interface (Snoop invalidate and TLB invalidate) are still handled as normal.
 - A2 signals chip power management logic that A2 is in PM_Sleep state by activating the *ac_an_power_managed* signal.
 - The state of all processor resources is maintained.
 - Wake up options are determined by CCR1[WC0,WC1] values.
3. PM_RVW
 - CCR0[PME] = PM_RVW_enable, and CCR0[WE] bits for all threads are set.
 - A2 activates run tholds to stop clocks for power savings. Requests from the A2/L2 interface (Snoop invalidate and TLB invalidate) are still handled as normal.
 - A2 signals chip power management logic that A2 is in PM_RVW state by activating the *ac_an_rvwinkle_mode* signal (**note:** *ac_an_power_managed* is also activated).
 - Optionally, chip level controls may shut off all clocking and power to the core. All processor state may be lost. Software must save any processor state prior to going into PM_RVW state.
 - Wake up options are determined by CCR1[WC0,WC1] values. A power-on reset is required when chip level power-saving actions drop power to the core, or disrupt clock synchronization.

Note that from the A2 Core standpoint, actual power-saving actions are the same whether the PM_Sleep or PM_RVW state is entered. In either case the core activates run tholds to stop some clocking; responds to snoop invalidate and TLB invalidate requests from the L2; and accepts wake up conditions enabled by CCR1. The difference in power-savings attained between PM_Sleep and PM_RVW states, as well as the method required to restart the core, depend on chip level hardware and software actions.

13.2.1 Power-saving instruction sequence

1. Software sets CCR1[WC0-WC1] to enable required wake up options for each thread. The CCR1 Wake Control (WC) field enables specific exceptions to interrupt the power-managed thread and resume execution. For each thread, the CCR1[WC] bits select from the following wake up conditions:
 - Wake on Critical Interrupt
 - Wake on External Interrupt

- Wake on Decrementer
 - Wake on Fixed Interval Timer
 - Wake on waitrsv (**wait** instruction with WC field set to 0b01 and *an_ac_reservation_vld* active)
 - Wake on waitimpl (**wait** instruction with WC field set to 0b10 and *an_ac_sleep_en* active)
2. Software sets CCR0[PME] to the desired power management enable control for the core.
 - **Disabled** - No additional power-savings actions when all CCR0[WE] bits are set.
 - **PM_Sleep_enable** - PM_Sleep power-savings state when all CCR0[WE] bits are set.
 - **PM_RVW_enable** - PM_RVW power-savings state when all CCR0[WE] bits are set.
 3. The thread is disabled. This stops the thread once outstanding operations are performed.
 - The thread is stopped; unused logic is clock-gated with the ACT control.
 - The *ac_an_pm_thread_running* signal to chip logic will be de-activated.
 - The thread continues to respond to Snoop and TLB invalidate requests from the L2.
 4. If all threads are stopped and *either* power-management enable (PM_Sleep_enable or PM_RVW_enable) is active, the A2 Core will:
 - Activate the run tholds to gate off additional core logic.
 - Activate the *ac_an_power_managed* signal to indicate that the core is in a power-savings state.
 - If PM_RVW_enable is active, the *ac_an_rvwinkle_mode* signal is also activated.
 5. Once the *ac_an_rvwinkle_mode* signal has been asserted, the L2 may take additional actions in preparation for chip power down. Further power-savings actions can be taken by stopping all core clocks and shutting off power to the core.
 6. Upon wake up from a power-savings state, the corresponding CCR0[WE] bit will be cleared, indicating that the corresponding thread has been restarted. The *ac_an_power_managed* and, if PM_RVW was enabled, the *ac_an_rvwinkle_mode* signals are deactivated. The run tholds are shut off to enable clocking to all core latches again. If in the PM_RVW state chip level power-saving actions remove power from the core or disrupt clock synchronization, then a power-on reset is required.



14. Register Summary

This chapter provides an alphabetical listing of and bit definitions for the registers contained in the A2O Core.

The five types of registers are grouped into several functional categories according to the processor functions with which they are associated. More information about the registers and register categories is provided in *Section 2.4 Registers* on page 79 and in the chapters describing the processor functions with which each register category is associated.

14.1 Register Categories

Table 14-1 lists the Special Purpose Registers (SPRs) in alphabetical order. The table gives register names, mnemonics, SPR numbers, access levels necessary for read/write, and mapping behavior, as well as listing if an SPR is multithreaded or slow.

Note: All SPR numbers not listed are reserved, and should be neither read nor written.

The “Access Mapped” column defines the mapping behavior when in guest mode. An “N” in this column means that the register is not mapped to any other register while in guest mode, nor does any register map to it. A “Y” means that mapping exists in guest mode. A register name in this column means that the register named in the field is the one used in guest mode. For more information about how register mapping in guest mode works, see *Section 7.5.1* on page 285.

The “Multithreaded” column indicates if the register is replicated or shared across threads. An “N” in this column means that the register is shared across threads. A “Y” means that the register is replicated for each thread.

The “Slow SPR” column indicates if the register is a “Slow” SPR. A “Y” in this column means that the register accesses go around the slow SPR bus, and will be lower performance than other register access. An “N” indicates that the register access does not use the slow SPR bus.

Table 1. Register Summary (Sheet 1 of 5)

Register Mnemonic	SPR Number	Minimum mtspr Access	Minimum mfspr Access	Access Mapped to Register	Multithreaded	Slow SPR	Scan Ring	Full Name	Before Write (Instr)	After Write (Instr)	Before Write (Data)	After Write (Data)	Note
ACOP	31	Priv	Priv	N	Y	N	func	Available Coprocessor	None	CSI	None	None	AM
AESR	913	Priv	Priv	N	N	Y	func	AXU Event Select Register	SR	SR	SR	SR	
CCR0	1008	Hypv	Hypv	N	N	N	bcfg	Core Configuration Register 0	None	None	None	None	
CCR1	1009	Hypv	Hypv	N	N	N	func	Core Configuration Register 1	None	CSI	None	None	
CCR2	1010	Hypv	Hypv	N	N	N	ccfg	Core Configuration Register 2	sync, CSI	sync, CSI	sync, CSI	sync, CSI	
CCR3	1013	Hypv	Hypv	N	Y	N	ccfg	Core Configuration Register 3	sync, CSI	sync, CSI	None	None	
CESR	912	Priv	Priv	N	N	Y	func	Core Event Select Register	SR	SR	SR	SR	
CR	N/A	Any	Any	N	Y	N	func	Condition Register	None	None	None	None	
CSRR0	58	Hypv	Hypv	N	Y	N	func	Critical Save/Restore Register 0	None	None	None	None	AM
CSRR1	59	Hypv	Hypv	N	Y	N	func	Critical Save/Restore Register 1	None	None	None	None	AM
CTR	9	Any	Any	N	Y	N	func	Count Register	None	None	None	None	
DAC1	316	Hypv	Hypv	N	N	N	func	Data Address Compare 1	N/A	N/A	None	CSI	AM
DAC2	317	Hypv	Hypv	N	N	N	func	Data Address Compare 2	N/A	N/A	None	CSI	AM
DAC3	849	Hypv	Hypv	N	N	N	func	Data Address Compare 3	N/A	N/A	None	CSI	
DAC4	850	Hypv	Hypv	N	N	N	func	Data Address Compare 4	N/A	N/A	None	CSI	
DBCR0	308	Hypv	Hypv	N	Y	N	dcfg	Debug Control Register 0	None	CSI	None	CSI	
DBCR1	309	Hypv	Hypv	N	Y	N	func	Debug Control Register 1	None	CSI	None	CSI	
DBCR2	310	Hypv	Hypv	N	Y	N	func	Debug Control Register 2	None	CSI	None	CSI	AM
DBCR3	848	Hypv	Hypv	N	Y	N	func	Debug Control Register 3	None	CSI	None	CSI	
DBSR ¹	304	Hypv	Hypv	N	Y	N	func	Debug Status Register	None	None	None	None	WC
DBSRWR	306	Hypv	None	N	Y	N	func	Debug Status Register Write Register	None	CSI	None	None	HM
DEAR	61	Priv	Priv	GDEAR	Y	N	func	Data Exception Address Register	None	None	None	None	
DEC	22	Hypv	Hypv	N	Y	N	func	Decrementer	None	None	None	None	
DECAR	54	Hypv	Hypv	N	Y	N	func	Decrementer Auto-Reload	None	None	None	None	AM
DVC1	318	Hypv	Hypv	N	N	Y	func	Data Value Compare 1	N/A	N/A	sync, CSI	CSI	AM
DVC2	319	Hypv	Hypv	N	N	Y	func	Data Value Compare 2	N/A	N/A	sync, CSI	CSI	AM

RO: This register or field is read only.

IO: This register or field is controlled by setting I/Os on the A2 core.

HO: This register or field is only writable in hypervisor state.

WS: Write to Set. Writing 1s to this field or register sets 1s. Writing 0s to this field or register has no effect.

WC: Write to Clear. Writing 1s to this field or register sets 0s. Writing 0s to this field or register has no effect.

HM: Only available when compiled with hvmode generic set to 1.

AM: Only available when compiled with a 2mode generic set to 1.

NP: This register or field is nonpersistent; reads always return zero.

Table 1. Register Summary (Sheet 2 of 5)

Register Mnemonic	SPR Number	Minimum mtspr Access	Minimum mfspr Access	Access Mapped to Register	Multithreaded	Slow SPR	Scan Ring	Full Name	Before Write (Instr)	After Write (Instr)	Before Write (Data)	After Write (Data)	Note
EPCR	307	Hypv	Hypv	N	Y	N	func	Embedded Processor Control Register	None	CSI	None	None	HM
EPLC	947	Priv	Priv	N	Y	Y	func	External Process ID Load Context	None	CSI	None	None	HM
EPSC	948	Priv	Priv	N	Y	Y	func	External Process ID Store Context	None	CSI	None	None	HM
EPTCFG	350	None	Hypv	N	N	Y	func	Embedded Page Table Configuration Register	None	None	None	None	HM
ESR	62	Priv	Priv	GESR	Y	N	func	Exception Syndrome Register	None	None	None	None	
GDEAR	381	Priv	Priv	Y	Y	N	func	Guest Data Exception Address Register	None	None	None	None	HM
GESR	383	Priv	Priv	Y	Y	N	func	Guest Exception Syndrome Register	None	None	None	None	HM
GIVPR	447	Hypv	Priv	N	N	N	func	Guest Interrupt Vector Prefix Register	None	None	None	None	HM
GPIR	382	Hypv	Priv	Y	Y	N	func	Guest Processor ID Register	None	None	None	None	HM
GSPRG0	368	Priv	Priv	Y	Y	N	func	Guest Software Special Purpose Register 0	None	None	None	None	HM
GSPRG1	369	Priv	Priv	Y	Y	N	func	Guest Software Special Purpose Register 1	None	None	None	None	HM
GSPRG2	370	Priv	Priv	Y	Y	N	func	Guest Software Special Purpose Register 2	None	None	None	None	HM
GSPRG3	371	Priv	Priv	Y	Y	N	func	Guest Software Special Purpose Register 3	None	None	None	None	HM
GSRR0	378	Priv	Priv	Y	Y	N	func	Guest Save/Restore Register 0	None	None	None	None	HM
GSRR1	379	Priv	Priv	Y	Y	N	func	Guest Save/Restore Register 1	None	None	None	None	HM
HACOP	351	Hypv	Priv	N	Y	N	func	Hypervisor Available Coprocessor	None	CSI	None	None	HM
IAC1	312	Hypv	Hypv	N	N	N	func	Instruction Address Compare 1	None	CSI	N/A	N/A	
IAC2	313	Hypv	Hypv	N	N	N	func	Instruction Address Compare 2	None	CSI	N/A	N/A	
IAC3	314	Hypv	Hypv	N	N	N	func	Instruction Address Compare 3	None	CSI	N/A	N/A	AM
IAC4	315	Hypv	Hypv	N	N	N	func	Instruction Address Compare 4	None	CSI	N/A	N/A	AM
IAR	882	Hypv	Hypv	N	Y	N	bcfg	Instruction Address Register	CSI	None	None	None	
IESR1	914	Priv	Priv	N	N	Y	func	IU Event Select Register 1	SR	SR	SR	SR	
IESR2	915	Priv	Priv	N	N	Y	func	IU Event Select Register 2	SR	SR	SR	SR	
IMMR	881	Hypv	Hypv	N	N	Y	func	Instruction Match Mask Register	None	CSI	None	None	AM
IMPDEP0	976 - 991	Hypv	Hypv	N/A	N/A	Y	N/A	Implementation Dependant Region 0	N/A	N/A	N/A	N/A	
IMPDEP1	912 - 927	Priv	Priv	N/A	N/A	Y	N/A	Implementation Dependant Region 1	N/A	N/A	N/A	N/A	
IMR	880	Hypv	Hypv	N	N	Y	func	Instruction Match Register	None	CSI	None	None	AM
IUCR0	1011	Hypv	Hypv	N	N	Y	ccfg	Instruction Unit Configuration Register 0	None	CSI	None	None	
IUCR1	883	Hypv	Hypv	N	Y	Y	ccfg	Instruction Unit Configuration Register 1	None	CSI	None	None	

RO: This register or field is read only.

IO: This register or field is controlled by setting I/Os on the A2 core.

HO: This register or field is only writable in hypervisor state.

WS: Write to Set. Writing 1s to this field or register sets 1s. Writing 0s to this field or register has no effect.

WC: Write to Clear. Writing 1s to this field or register sets 0s. Writing 0s to this field or register has no effect.

HM: Only available when compiled with hvmode generic set to 1.

AM: Only available when compiled with a2mode generic set to 1.

NP: This register or field is nonpersistent; reads always return zero.



Table 1. Register Summary (Sheet 3 of 5)

Register Mnemonic	SPR Number	Minimum mtspr Access	Minimum mfspr Access	Access Mapped to Register	Multithreaded	Slow SPR	Scan Ring	Full Name	Before Write (Instr)	After Write (Instr)	Before Write (Data)	After Write (Data)	Note
IUCR2	884	Hypv	Hypv	N	Y	Y	ccfg	Instruction Unit Configuration Register 2	None	CSI	None	None	
IUDBG0	888	Hypv	Hypv	N	N	Y	func	Instruction Unit Debug Register 0	Quiesce	None	None	None	
IUDBG1	889	None	Hypv	N	N	Y	func	Instruction Unit Debug Register 1	None	None	None	None	
IUDBG2	890	None	Hypv	N	N	Y	func	Instruction Unit Debug Register 2	None	None	None	None	
IULFSR	891	Hypv	Hypv	N	N	Y	func	Instruction Unit LFSR	None	CSI	None	None	
IULLCR	892	Hypv	Hypv	N	N	Y	ccfg	Instruction Unit Live Lock Control Register	None	None	None	None	
IVPR	63	Hypv	Hypv	N	N	N	func	Interrupt Vector Prefix Register	None	None	None	None	
LPER	56	Hypv	Hypv	N	Y	Y	func	Logical Page Exception Register	None	None	None	None	HM
LPERU	57	Hypv	Hypv	N	Y	Y	func	Logical Page Exception Register (Upper)	None	None	None	None	HM
LPIDR	338	Hypv	Hypv	N	N	Y	func	Logical Partition ID Register	None	CSI	CSI	CSI	
LR	8	Any	Any	N	Y	N	func	Link Register	None	None	None	None	
LRATCFG	342	None	Hypv	N	N	Y	func	LRAT Configuration Register	None	None	None	None	HM
LRATPS	343	None	Hypv	N	N	Y	func	LRAT Page Size Register	None	None	None	None	HM
MAS0	624	Priv	Priv	N	Y	Y	func	MMU Assist Register 0	None	None	None	None	HM
MAS0_MAS1	373	Priv	Priv	N	Y	Y	func	MMU Assist Registers 0 and 1	None	None	None	None	HM
MAS1	625	Priv	Priv	N	Y	Y	func	MMU Assist Register 1	None	None	None	None	HM
MAS2	626	Priv	Priv	N	Y	Y	func	MMU Assist Register 2	None	None	None	None	HM
MAS2U	631	Priv	Priv	N	Y	Y	func	MMU Assist Register 2 (Upper)	None	None	None	None	HM
MAS3	627	Priv	Priv	N	Y	Y	func	MMU Assist Register 3	None	None	None	None	HM
MAS4	628	Priv	Priv	N	Y	Y	ccfg	MMU Assist Register 4	None	None	None	None	HM
MAS5	339	Hypv	Hypv	N	Y	Y	func	MMU Assist Register 5	None	None	None	None	HM
MAS5_MAS6	348	Hypv	Hypv	N	Y	Y	func	MMU Assist Registers 5 and 6	None	None	None	None	HM
MAS6	630	Priv	Priv	N	Y	Y	func	MMU Assist Register 6	None	None	None	None	HM
MAS7	944	Priv	Priv	N	Y	Y	func	MMU Assist Register 7	None	None	None	None	HM
MAS7_MAS3	372	Priv	Priv	N	Y	Y	func	MMU Assist Registers 7 and 3	None	None	None	None	HM
MAS8	341	Hypv	Hypv	N	Y	Y	func	MMU Assist Register 8	None	None	None	None	HM
MAS8_MAS1	349	Hypv	Hypv	N	Y	Y	func	MMU Assist Registers 8 and 1	None	None	None	None	HM
MCSR ¹	572	Hypv	Hypv	N	Y	N	func	Machine Check Syndrome Register	None	None	None	None	AM
MCSRR0	570	Hypv	Hypv	N	Y	N	func	Machine Check Save/Restore Register 0	None	None	None	None	AM

- RO:** This register or field is read only.
- IO:** This register or field is controlled by setting I/Os on the A2 core.
- HO:** This register or field is only writable in hypervisor state.
- WS:** Write to Set. Writing 1s to this field or register sets 1s. Writing 0s to this field or register has no effect.
- WC:** Write to Clear. Writing 1s to this field or register sets 0s. Writing 0s to this field or register has no effect.
- HM:** Only available when compiled with hvmode generic set to 1.
- AM:** Only available when compiled with a 2mode generic set to 1.
- NP:** This register or field is nonpersistent; reads always return zero.

Table 1. Register Summary (Sheet 4 of 5)

Register Mnemonic	SPR Number	Minimum mtspr Access	Minimum mfspr Access	Access Mapped to Register	Multithreaded	Slow SPR	Scan Ring	Full Name	Before Write (Instr)	After Write (Instr)	Before Write (Data)	After Write (Data)	Note
MCSRR1	571	Hypv	Hypv	N	Y	N	func	Machine Check Save/Restore Register 1	None	None	None	None	AM
MESR1	916	Priv	Priv	N	N	Y	func	MMU Event Select Register 1	SR	SR	SR	SR	
MESR2	917	Priv	Priv	N	N	Y	func	MMU Event Select Register 2	SR	SR	SR	SR	
MMUCFG	1015	None	Hypv	N	N	Y	ccfg	MMU Configuration Register	None	None	None	None	HM
MMUCR0	1020	Hypv	Hypv	N	Y	Y	func	Memory Management Unit Control Register 0	None	None	None	None	AM
MMUCR1	1021	Hypv	Hypv	N	N	Y	ccfg	Memory Management Unit Control Register 1	None	None	None	None	AM
MMUCR2	1022	Hypv	Hypv	N	N	Y	ccfg	Memory Management Unit Control Register 2	None	CSI	None	None	AM
MMUCR3	1023	Priv	Priv	N	Y	Y	ccfg	Memory Management Unit Control Register 3	None	None	None	None	HM
MMUCSR0	1012	Hypv	Hypv	N	N	Y	func	MMU Control and Status Register 0	None	CSI, sync	None	CSI, sync	HM
MSR	N/A	Priv	Priv	N	Y	N	ccfg	Machine State Register	None	None	None	None	
MSRP	311	Hypv	Hypv	N	Y	N	func	Machine State Register Protect	None	None	None	None	HM
PID	48	Priv	Priv	N	Y	Y	func	Process ID	None	CSI	None	None	
PIR	286	None	Priv	GPIR	N	N	func	Processor ID Register	None	None	None	None	
PPR32	898	Any	Any	N	Y	Y	ccfg	Program Priority Register	None	CSI	None	None	
PVR	287	None	Priv	N	N	N	func	Processor Version Register	None	None	None	None	
SPRG0	272	Priv	Priv	GSPRG0	Y	N	func	Software Special Purpose Register 0	None	None	None	None	
SPRG1	273	Priv	Priv	GSPRG1	Y	N	func	Software Special Purpose Register 1	None	None	None	None	
SPRG2	274	Priv	Priv	GSPRG2	Y	N	func	Software Special Purpose Register 2	None	None	None	None	
SPRG3	275/259	Priv/None	Priv/Any	GSPRG3	Y	N	func	Software Special Purpose Register 3	None	None	None	None	
SPRG4	276/260	Priv/None	Priv/Any	N	Y	N	func	Software Special Purpose Register 4	None	None	None	None	
SPRG5	277/261	Priv/None	Priv/Any	N	Y	N	func	Software Special Purpose Register 5	None	None	None	None	
SPRG6	278/262	Priv/None	Priv/Any	N	Y	N	func	Software Special Purpose Register 6	None	None	None	None	
SPRG7	279/263	Priv/None	Priv/Any	N	Y	N	func	Software Special Purpose Register 7	None	None	None	None	
SPRG8	604	Hypv	Hypv	N	Y	N	func	Software Special Purpose Register 8	None	None	None	None	
SRR0	26	Priv	Priv	GSRR0	Y	N	func	Save/Restore Register 0	None	None	None	None	
SRR1	27	Priv	Priv	GSRR1	Y	N	func	Save/Restore Register 1	None	None	None	None	
TB	268	None	Any	N	N	N	func	Timebase	None	None	None	None	

- RO:** This register or field is read only.
- IO:** This register or field is controlled by setting I/Os on the A2 core.
- HO:** This register or field is only writable in hypervisor state.
- WS:** Write to Set. Writing 1s to this field or register sets 1s. Writing 0s to this field or register has no effect.
- WC:** Write to Clear. Writing 1s to this field or register sets 0s. Writing 0s to this field or register has no effect.
- HM:** Only available when compiled with hvmode generic set to 1.
- AM:** Only available when compiled with a2mode generic set to 1.
- NP:** This register or field is nonpersistent; reads always return zero.

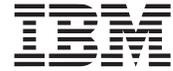


Table 1. Register Summary (Sheet 5 of 5)

Register Mnemonic	SPR Number	Minimum mtspr Access	Minimum mfspr Access	Access Mapped to Register	Multithreaded	Slow SPR	Scan Ring	Full Name	Before Write (Instr)	After Write (Instr)	Before Write (Data)	After Write (Data)	Note
TBL	284	Hypv	None	N	N	N	func	Timebase Lower	None	None	None	None	
TBU	285/269	Hypv/None	None/Any	N	N	N	func	Timebase Upper	None	None	None	None	
TCR	340	Hypv	Hypv	N	Y	N	func	Timer Control Register	None	None	None	None	AM
TENC	439	Hypv	Hypv	N	N	N	bcfg	Thread Enable Clear Register	None	None	None	None	WC
TENS	438	Hypv	Hypv	N	N	N	bcfg	Thread Enable Set Register	None	None	None	None	WS
TENSR	437	None	Hypv	N	N	N	func	Thread Enable Status Register	None	None	None	None	
TIR	446	None	Hypv	N	N	N	func	Thread Identification Register	None	None	None	None	
TLB0CFG	688	None	Hypv	N	N	Y	ccfg	TLB 0 Configuration Register	None	None	None	None	HM
TLB0PS	344	None	Hypv	N	N	Y	func	TLB 0 Page Size Register	None	None	None	None	HM
TRACE	1006	Any	None	N	N	N	func	Hardware Trace Macro Control Register	None	None	None	None	
TSR ¹	336	Hypv	Hypv	N	Y	N	func	Timer Status Register	None	None	None	None	WC, AM
UDEC	550	Any	Any	N	Y	N	func	User Decrementer	None	None	None	None	AM
VRSAVE	256	Any	Any	N	Y	N	func	Vector Register Save	None	None	None	None	
XER	1	Any	Any	N	Y	N	func	Fixed Point Exception Register	None	None	None	None	
XESR1	918	Priv	Priv	N	N	Y	func	XU Event Select Register 1	SR	SR	SR	SR	
XESR2	919	Priv	Priv	N	N	Y	func	XU Event Select Register 2	SR	SR	SR	SR	
XESR3	920	Priv	Priv	N	N	Y	func	XU Event Select Register 3	SR	SR	SR	SR	
XESR4	921	Priv	Priv	N	N	Y	func	XU Event Select Register 4	SR	SR	SR	SR	
XUCR0	1014	Hypv	Hypv	N	N	N	ccfg	Execution Unit Configuration Register 0	sync, CSI	sync, CSI	sync, CSI	sync, CSI	
XUCR1	851	Hypv	Hypv	N	Y	N	ccfg	Execution Unit Configuration Register 1	sync, CSI	sync, CSI	sync, CSI	sync, CSI	
XUCR2	1016	Hypv	Hypv	N	N	Y	func	Execution Unit Configuration Register 2	sync, CSI	sync, CSI	sync, CSI	sync, CSI	
XUCR3	852	Hypv	Hypv	N	N	N	dcfg	Execution Unit Configuration Register 3	None	None	None	None	
XUCR4	853	Hypv	Hypv	N	N	N	dcfg	Execution Unit Configuration Register 4	None	None	None	None	
XUDBG0	885	Hypv	Hypv	N	N	Y	func	Execution Unit Debug Register 0	None	None	Quiesce	None	
XUDBG1	886	None	Hypv	N	N	Y	func	Execution Unit Debug Register 1	None	None	None	None	
XUDBG2	887	None	Hypv	N	N	Y	func	Execution Unit Debug Register 2	None	None	None	None	

RO: This register or field is read only.
IO: This register or field is controlled by setting I/Os on the A2 core.
HO: This register or field is only writable in hypervisor state.
WS: Write to Set. Writing 1s to this field or register sets 1s. Writing 0s to this field or register has no effect.
WC: Write to Clear. Writing 1s to this field or register sets 0s. Writing 0s to this field or register has no effect.
HM: Only available when compiled with hvmode generic set to 1.
AM: Only available when compiled with a 2mode generic set to 1.
NP: This register or field is nonpersistent; reads always return zero.

1. DBSR, MCSR, and TSR have read/clear access. These three registers are *status* registers, and as such behave differently than other SPRs when written. The term “read/clear” does not mean that these registers are automatically cleared upon being read. Rather, the “clear” refers to their behavior when being written. Instead of simply overwriting the SPR with the data in the source GPR, the status SPR is updated by zeroing those bit positions corresponding to 1 values in the source GPR; those bit positions corresponding to 0 values in the source GPR are left unchanged. In this fashion, it is possible for software to read one of these status SPRs, and then write to it using the same data that was read. Any bits that were read as 1 are then cleared, and any bits that were not yet set at the time the SPR was read are left unchanged. If any of these previously clear bits happen to be set between the time the SPR is read and when it is written, then when the SPR is later read again, software observes any newly set bits. If it were not for this behavior, software could erroneously clear bits that it had not yet observed as having been set, and overlook the occurrence of certain exceptions.

14.2 Reserved Fields

For all registers with fields marked as reserved, the reserved fields should be written as *zero* and read as *undefined*. That is, when writing to a reserved field, write a zero to that field. When reading from a reserved field, ignore that field.

The recommended coding practice is to perform the initial write to a register with reserved fields as described in the preceding paragraph, and to perform all subsequent writes to the register using a read-modify-write strategy: read the register, alter desired fields with logical instructions, and then write the register.

Note: Software must not set any field of a system register to a reserved value.

14.3 Unimplemented SPRs

An unimplemented SPR is defined as any SPR number that is not listed in *Table 14-1*.

14.4 Device Control Registers

Move to and move from DCR instructions when CCR2(EN_DCR) is zero are dropped silently; they are no-ops and do not cause an exception.



14.5 Alphabetical Register Listing

The following pages list the registers available in the A2 Core. For each register, the following information is supplied:

- Register mnemonic and name
- Cross reference to detailed register information
- Register type (if SPR); the types of the other registers are the same as the register names (CR, GPR, MSR)
- Register number (address)
- Register programming model (user or supervisor) and access (read-clear, read-only, read/write (R/W), write-only)
- A diagram illustrating the register fields (all register fields have mnemonics, unless there is only one field)
- A table describing the register fields, giving field mnemonics, field bit locations, field names, and the functions associated with the various field values

14.5.1 A0ESR - AXU0 Event Select Register

Register Short Name:	A0ESR	Read Access:	Priv
Decimal SPR Number:	913	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	INPSELEB0	0b0	<u>Mux Event_Bits(0) Input Select</u> For event mux, bit 0, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:7), 1 = A1_Events(0:7)
33:35	MUXSELEB0	0b000	<u>Mux Event_Bits(0) 2:1 Mux Select</u> Selects which 2:1 mux is gated for driving bit 0 of the event mux (axu0_iu_event_bits(0)). Decoded values select Mux 0 ("000") through Mux 7 ("111").
36	INPSELEB1	0b0	<u>Mux Event_Bits(1) Input Select</u> For event mux, bit 1, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:7), 1 = A1_Events(0:7)
37:39	MUXSELEB1	0b000	<u>Mux Event_Bits(1) 2:1 Mux Select</u> Selects which 2:1 mux is gated for driving bit 1 of the event mux (axu0_iu_event_bits(1)). Decoded values select Mux 0 ("000") through Mux 7 ("111").
40	INPSELEB2	0b0	<u>Mux Event_Bits(2) Input Select</u> For event mux, bit 2, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:7), 1 = A1_Events(0:7)
41:43	MUXSELEB2	0b000	<u>Mux Event_Bits(2) 2:1 Mux Select</u> Selects which 2:1 mux is gated for driving bit 2 of the event mux (axu0_iu_event_bits(2)). Decoded values select Mux 0 ("000") through Mux 7 ("111").
44	INPSELEB3	0b0	<u>Mux Event_Bits(3) Input Select</u> For event mux, bit 3, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:7), 1 = A1_Events(0:7)
45:47	MUXSELEB3	0b000	<u>Mux Event_Bits(3) 2:1 Mux Select</u> Selects which 2:1 mux is gated for driving bit 3 of the event mux (axu0_iu_event_bits(3)). Decoded values select Mux 0 ("000") through Mux 7 ("111").
48	INPSELEB4	0b0	<u>Mux Event_Bits(4) Input Select</u> For event mux, bit 4, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:7), 1 = B1_Events(0:7)
49:51	MUXSELEB4	0b000	<u>Mux Event_Bits(4) 2:1 Mux Select</u> Selects which 2:1 mux is gated for driving bit 4 of the event mux (axu0_iu_event_bits(4)). Decoded values select Mux 0 ("000") through Mux 7 ("111").
52	INPSELEB5	0b0	<u>Mux Event_Bits(5) Input Select</u> For event mux, bit 5, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:7), 1 = B1_Events(0:7)

Bit(s):	Field Name:	Init	Description
53:55	MUXSELEB5	0b000	<u>Mux Event_Bits(5) 2:1 Mux Select</u> Selects which 2:1 mux is gated for driving bit 5 of the event mux (axu0_iu_event_bits(5)). Decoded values select Mux 0 ("000") through Mux 7 ("111").
56	INPSELEB6	0b0	<u>Mux Event_Bits(6) Input Select</u> For event mux, bit 6, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:7), 1 = B1_Events(0:7)
57:59	MUXSELEB6	0b000	<u>Mux Event_Bits(6) 2:1 Mux Select</u> Selects which 2:1 mux is gated for driving bit 6 of the event mux (axu0_iu_event_bits(6)). Decoded values select Mux 0 ("000") through Mux 7 ("111").
60	INPSELEB7	0b0	<u>Mux Event_Bits(7) Input Select</u> For event mux, bit 7, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:7), 1 = B1_Events(0:7)
61:63	MUXSELEB7	0b000	<u>Mux Event_Bits(7) 2:1 Mux Select</u> Selects which 2:1 mux is gated for driving bit 7 of the event mux (axu0_iu_event_bits(7)). Decoded values select Mux 0 ("000") through Mux 7 ("111").

14.5.2 ACOP - Available Coprocessor

Register Short Name:	ACOP	Read Access:	Priv
Decimal SPR Number:	31	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:63	CT	0x0	<p><u>Coprocessor Type</u></p> <p>Indicates available coprocessor types for the icswx instruction. Bit n of the register indicates availability of coprocessor type n.</p> <p>0 Coprocessor unavailable. Accesses will generate an unavailable coprocessor type data storage interrupt.</p> <p>1 Coprocessor available.</p>

14.5.3 CCR0 - Core Configuration Register 0

Register Short Name:	CCR0	Read Access:	Hypv
Decimal SPR Number:	1008	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	bcfg

Bit(s):	Field Name:	Init	Description
32:33	PME	0b00	<p><u>Power Management Enable</u></p> <p>00 Disabled: No power savings mode entered. 01 PM_Sleep_enable: PM_Sleep state entered when all threads are stopped. 10 PM_RVW_enable: PM_RVW state entered when all threads are stopped. 11 Disabled2: No power savings mode entered.</p> <p>NOTE: Refer to the A2 User Manual, Power Management Methods section.</p>
34:51	///	0x0	<u>Reserved</u>
52:55	WEM	0b0000	<p><u>Wait Enable Mask</u></p> <p>0 No effect to CCR0[WE] 1 Allows writing of corresponding bit in CCR0[WE] field. These bits are non-persistent. A read always returns zeros.</p>
56:59	///	0b0000	<u>Reserved</u>
60:63	WE	0b0000	<p><u>Wait Enable</u></p> <p>For $t < 4$, bit 63-t corresponds to thread t: 0 Indicates the thread is enabled 1 Indicates the thread is disabled</p> <p>Note: This field may also be set by a 'wait' instruction</p>

14.5.4 CCR1 - Core Configuration Register 1

Register Short Name:	CCR1	Read Access:	Hypv
Decimal SPR Number:	1009	Write Access:	Hypv
Initial Value:	0x00000000f0f0f0f	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	///	0b00	<u>Reserved</u>
34:39	WC3	0xf	<u>Thread 3 Wake Control</u> (0) 1 Disables sleep on waitrsv (1) 1 Disables sleep on waitimpl (2) 1 Enables wake on Critical Input, Watchdog, Critical Doorbell, Guest Critical Doorbell, or Guest Machine Check Doorbell Interrupts (3) 1 Enables wake on External Input, Performance Monitor, Doorbell, or Guest Doorbell Interrupts (4) 1 Enables wake on Decrementer or User Decrementer Interrupts (5) 1 Enables wake on Fixed Interval Timer Interrupts
40:41	///	0b00	<u>Reserved</u>
42:47	WC2	0xf	<u>Thread 2 Wake Control</u> (0) 1 Disables sleep on waitrsv (1) 1 Disables sleep on waitimpl (2) 1 Enables wake on Critical Input, Watchdog, Critical Doorbell, Guest Critical Doorbell, or Guest Machine Check Doorbell Interrupts (3) 1 Enables wake on External Input, Performance Monitor, Doorbell, or Guest Doorbell Interrupts (4) 1 Enables wake on Decrementer or User Decrementer Interrupts (5) 1 Enables wake on Fixed Interval Timer Interrupts
48:49	///	0b00	<u>Reserved</u>
50:55	WC1	0xf	<u>Thread 1 Wake Control</u> (0) 1 Disables sleep on waitrsv (1) 1 Disables sleep on waitimpl (2) 1 Enables wake on Critical Input, Watchdog, Critical Doorbell, Guest Critical Doorbell, or Guest Machine Check Doorbell Interrupts (3) 1 Enables wake on External Input, Performance Monitor, Doorbell, or Guest Doorbell Interrupts (4) 1 Enables wake on Decrementer or User Decrementer Interrupts (5) 1 Enables wake on Fixed Interval Timer Interrupts
56:57	///	0b00	<u>Reserved</u>
58:63	WC0	0xf	<u>Thread 0 Wake Control</u> (0) 1 Disables sleep on waitrsv (1) 1 Disables sleep on waitimpl (2) 1 Enables wake on Critical Input, Watchdog, Critical Doorbell, Guest Critical Doorbell, or Guest Machine Check Doorbell Interrupts (3) 1 Enables wake on External Input, Performance Monitor, Doorbell, or Guest Doorbell Interrupts (4) 1 Enables wake on Decrementer or User Decrementer Interrupts (5) 1 Enables wake on Fixed Interval Timer Interrupts

14.5.5 CCR2 - Core Configuration Register 2

Register Short Name:	CCR2	Read Access:	Hypv
Decimal SPR Number:	1010	Write Access:	Hypv
Initial Value:	0x0000000000000001	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32	EN_DCR	0b0	<u>Enable DCR instructions</u> 0 Disable Embedded.Device Control category instructions 1 Enable Embedded.Device Control category instructions
33	EN_TRACE	0b0	<u>Enable Hardware Trace Control Register</u> 0 Writes to TRACE SPR behave as a nop 1 Writes to TRACE SPR are enabled
34	EN_PC	0b0	<u>Enable Processor Control Instructions</u> 0 Disable "msgsnd" and "msgclr" Instructions, will cause illegal instruction type program interrupt 1 Enable "msgsnd" and "msgclr" Instructions
35:43	IFRATSC	0x0	<u>IFRAT Storage Control</u> Sets the storage control bits used when CCR2[IFRAT]=1. When set to 1, indicates: 0 (W) Write Through Required 1 (I) Caching Inhibited 2 (M) Memory Coherence Required 3 (G) Guarded 4 (E) Big Endian 5 (U0) User Defined Bit 0 6 (U1) User Defined Bit 1 7 (U2) User Defined Bit 2 8 (U3) User Defined Bit 3
44	IFRAT	0b0	<u>Instruction Force Real Address Translation</u> Debug facility which forces EA=RA address translation for instructions: 0 Access I-ERAT for instruction translation 1 Force EA=RA instruction translation
45:53	DFRATSC	0x0	<u>DFRAT Storage Control</u> Sets the storage control bits used when CCR2[DFRAT]=1. When set to 1, indicates: 0 (W) Write Through Required 1 (I) Caching Inhibited 2 (M) Memory Coherence Required 3 (G) Guarded 4 (E) Big Endian 5 (U0) User Defined Bit 0 6 (U1) User Defined Bit 1 7 (U2) User Defined Bit 2 8 (U3) User Defined Bit 3
54	DFRAT	0b0	<u>Data Force Real Address Translation</u> Debug facility which forces EA=RA address translation for data: 0 Access D-ERAT for data translation 1 Force EA=RA data translation

Bit(s):	Field Name:	Init	Description
55	UCODE_DIS	0b0	<u>Microcode Disable</u> 0 Enable microcode (Normal Operation) 1 Disable microcode (All microcoded instructions cause an unimplemented op type program interrupt)
56:59	AP	0b0000	<u>Auxillary Processor Available</u> Per thread enable for auxillary processor instructions; this field corresponds to threads [0:3]. 0 The auxillary processor cannot execute any instructions 1 The processor can execute instructions
60	EN_ATTN	0b0	<u>Enable Attn Instruction</u> 0 Disable "Attn" Instruction, will cause illegal instruction type program interrupt 1 Enable "Attn" Instruction
61	EN_DITC	0b0	<u>Enable mtdp/mfdp Instructions</u> 0 Disable "mtdp" and "mfdp" Instructions, will cause illegal instruction type program interrupt 1 Enable "mtdp" and "mfdp" Instruction
62	EN_ICSWX	0b0	<u>Enable icswx Instruction</u> 0 Disable "icswx" Instruction, will cause illegal instruction type program interrupt 1 Enable "icswx" Instruction
63	NOTLB	0b1	<u>ERAT Only Mode</u> 0 Backing TLB is present 1 No Backing TLB, use ERATs as micro-TLBs

14.5.6 CCR3 - Core Configuration Register 3

Register Short Name:	CCR3	Read Access:	Hypv
Decimal SPR Number:	1013	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:61	///	0x0	<u>Reserved</u>
62	EN_EEPRI	0b0	<u>Raise Priority when External Interrupts are Disabled</u> 0 Use Priority in PPR32[PRI] 1 If MSR[EE]=0 & PPR32[PRI]=A2LOW, effective priority is raised to A2MEDIUM. PPR32 is left unchanged.
63	SI	0b0	<u>Single Instruction Mode</u> 0 Processor Runs Normally 1 Process executes only one instruction per thread at a time

14.5.7 CCR4 - Core Configuration Register 4

Register Short Name:	CCR4	Read Access:	Hypv
Decimal SPR Number:	854	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:62	///	0x0	<u>Reserved</u>
63	EN_DNH	0b0	<u>Enable dnh Instruction</u> 0 Disable "dnh" Instruction, will cause illegal instruction type program interrupt 1 Enable "dnh" Instruction

14.5.8 CESR1 - Core Event Select Register1

Register Short Name:	CESR1	Read Access:	Priv
Decimal SPR Number:	912	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	ENABPERF	0b0	<u>Enable Performance Event Latches</u> When set, latches used to redrive performance event signals and the event bus are enabled. This bit must be set prior to making performance measurements.
33:35	COUNTMODES	0b000	<u>Performance Event Count Modes</u> This field determines which count modes are valid for the selected performance events. More than one count mode bit at a time may be enabled. 33 = count events when in Problem mode. 34 = count events when in Guest Supervisor mode. 35 = count events when in Hypervisor mode.
36	ENABTRACEBUS	0b0	<u>Enable Trace-Trigger Bus Latches</u> When set, latches used to redrive the Trace-Trigger bus and trace related signals are enabled. This bit is an alternate method of enabling the Trace-Trigger bus, similar to PCCR0[Enable Debug Mode]. Unlike PCCR0[Enable Debug Mode] it does not enable additional debug mode functions in the THRCTL or PCCR0 registers.
37	///	0b0	<u>Reserved</u>
38	SELDBGHI	0b0	<u>Select Trace Bits on Event Bus (0:3)</u> Selects ac_an_debug_bus(0:3) to drive out on ac_an_event_bus(0:3) in place of performance event signals. Any debug signal muxed onto these bits can be counted by the PMU.
39	SELDBGLO	0b0	<u>Select Trace Bits on Event Bus (4:7)</u> Selects ac_an_debug_bus(4:7) to drive out on ac_an_event_bus(4:7) in place of performance event signals. Any debug signal muxed onto these bits can be counted by the PMU.
40	INSTTRACE	0b0	<u>Instruction Trace Mode Enable</u> This bit enables support of the Core Trace function by activating mux selects and controls used to perform instruction tracing. Note: This bit applies to Server Group Instruction Tracing function.
41	INSTTRACETID	0b0	<u>Instruction Trace Mode Thread ID</u> Indicates which thread is selected for Core Trace. T0='0', T1='1' Note: This bit applies to Server Group Instruction Tracing function.
42:43	///	0b00	<u>Reserved</u>
44	PMAE_T0	0b0	<u>Performance Monitor Alert Enable, T0</u> 0 Performance Monitor Alerts on thread 0 are disabled. 1 Performance Monitor Alerts on thread 0 are enabled. Software can set or clear this bit. Hardware will clear this bit upon activation of a performance monitor alert.

Bit(s):	Field Name:	Init	Description
45	PMAO_T0	0b0	<u>Performance Monitor Alert Occurred, T0</u> 0 A Performance Monitor Alert has not occurred on thread 0 since this bit was last cleared. 1 A Performance Monitor Alert has occurred on thread 0 since this bit was last cleared. Software can set or clear this bit. Hardware will set this bit upon activation of a performance monitor alert.
46	PMAE_T1	0b0	<u>Performance Monitor Alert Enable, T1</u> 0 Performance Monitor Alerts on thread 1 are disabled. 1 Performance Monitor Alerts on thread 1 are enabled. Software can set or clear this bit. Hardware will clear this bit upon activation of a performance monitor alert.
47	PMAO_T1	0b0	<u>Performance Monitor Alert Occurred, T1</u> 0 A Performance Monitor Alert has not occurred on thread 1 since this bit was last cleared. 1 A Performance Monitor Alert has occurred on thread 1 since this bit was last cleared. Software can set or clear this bit. Hardware will set this bit upon activation of a performance monitor alert.
48:63	///	0x0	<u>Reserved</u>

14.5.9 CESR2 - Core Event Select Register2

Register Short Name:	CESR2	Read Access:	Priv
Decimal SPR Number:	893	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	SELSPEC0	0b0	<u>Enable speculative events on ac_an_event_bus(0)</u> When active, the performance event signals from each unit are driven out directly. The default state will only output non-speculative events.
33:35	SELEB0	0b000	<u>Select signal driven on ac_an_event_bus(0)</u> Decoded value for 0b000 through 0b111 is TBD.
36	SELSPEC1	0b0	<u>Enable speculative events on ac_an_event_bus(1)</u> When active, the performance event signals from each unit are driven out directly. The default state will only output non-speculative events.
37:39	SELEB1	0b000	<u>Select signal driven on ac_an_event_bus(1)</u> Decoded value for 0b000 through 0b111 is TBD.
40	SELSPEC2	0b0	<u>Enable speculative events on ac_an_event_bus(2)</u> When active, the performance event signals from each unit are driven out directly. The default state will only output non-speculative events.
41:43	SELEB2	0b000	<u>Select signal driven on ac_an_event_bus(2)</u> Decoded value for 0b000 through 0b111 is TBD.
44	SELSPEC3	0b0	<u>Enable speculative events on ac_an_event_bus(3)</u> When active, the performance event signals from each unit are driven out directly. The default state will only output non-speculative events.
45:47	SELEB3	0b000	<u>Select signal driven on ac_an_event_bus(3)</u> Decoded value for 0b000 through 0b111 is TBD.
48	SELSPEC4	0b0	<u>Enable speculative events on ac_an_event_bus(4)</u> When active, the performance event signals from each unit are driven out directly. The default state will only output non-speculative events.
49:51	SELEB4	0b000	<u>Select signal driven on ac_an_event_bus(4)</u> Decoded value for 0b000 through 0b111 is TBD.
52	SELSPEC5	0b0	<u>Enable speculative events on ac_an_event_bus(5)</u> When active, the performance event signals from each unit are driven out directly. The default state will only output non-speculative events.
53:55	SELEB5	0b000	<u>Select signal driven on ac_an_event_bus(5)</u> Decoded value for 0b000 through 0b111 is TBD.
56	SELSPEC6	0b0	<u>Enable speculative events on ac_an_event_bus(6)</u> When active, the performance event signals from each unit are driven out directly. The default state will only output non-speculative events.
57:59	SELEB6	0b000	<u>Select signal driven on ac_an_event_bus(6)</u> Decoded value for 0b000 through 0b111 is TBD.
60	SELSPEC7	0b0	<u>Enable speculative events on ac_an_event_bus(7)</u> When active, the performance event signals from each unit are driven out directly. The default state will only output non-speculative events.

Bit(s):	Field Name:	Init	Description
61:63	SELEB7	0b000	<u>Select signal driven on ac_an_event_bus(7)</u> Decoded value for 0b000 through 0b111 is TBD.

14.5.10 CPCRO - Completion Configuration Register 0

Register Short Name:	CPCRO	Read Access:	Hypv
Decimal SPR Number:	816	Write Access:	Hypv
Initial Value:	0x000000000111100	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:42	///	0x0	<u>Reserved</u>
43:47	FX1_WM	0x11	<u>FX1 Watermark</u>
48:50	///	0b000	<u>Reserved</u>
51:55	FX0_WM	0x11	<u>FX0 Watermark</u>
56:58	///	0b000	<u>Reserved</u>
59:63	LSU_WM	0x0	<u>LSU Watermark</u>

14.5.11 CPR1 - Completion Configuration Register 1

Register Short Name:	CPCR1	Read Access:	Hypv
Decimal SPR Number:	817	Write Access:	Hypv
Initial Value:	0x000000000000c0c0c	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:42	///	0x0	<u>Reserved</u>
43:47	FX1_CNT	0xc	<u>FX1 Credit Count</u>
48:50	///	0b000	<u>Reserved</u>
51:55	FX0_CNT	0xc	<u>FX0 Credit Count</u>
56:58	///	0b000	<u>Reserved</u>
59:63	SQ_CNT	0xc	<u>LSU Credit Count</u>

14.5.12 CPR2 - Completion Configuration Register 2

Register Short Name:	CPCR2	Read Access:	Hypv
Decimal SPR Number:	818	Write Access:	Hypv
Initial Value:	0x0000000000002810	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:42	///	0x0	<u>Reserved</u>
43	LSU_INORDER	0b0	<u>LSU In-Order Mode</u>
44:48	///	0x0	<u>Reserved</u>
49:55	CP_CNT	0x28	<u>Completion Credit Count</u>
56:58	///	0b000	<u>Reserved</u>
59:63	LS_CNT	0x10	<u>Load Store Credit Count</u>

14.5.13 CR - Condition Register

Register Short Name:	CR	Read Access:	Any
Decimal SPR Number:	N/A	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:35	CR0	0b0000	<u>Condition Register Field 0</u>
36:39	CR1	0b0000	<u>Condition Register Field 1</u>
40:43	CR2	0b0000	<u>Condition Register Field 2</u>
44:47	CR3	0b0000	<u>Condition Register Field 3</u>
48:51	CR4	0b0000	<u>Condition Register Field 4</u>
52:55	CR5	0b0000	<u>Condition Register Field 5</u>
56:59	CR6	0b0000	<u>Condition Register Field 6</u>
60:63	CR7	0b0000	<u>Condition Register Field 7</u>

14.5.14 CSRR0 - Critical Save/Restore Register 0

Register Short Name:	CSRR0	Read Access:	Hypv
Decimal SPR Number:	58	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	SRR0	0x0	<p><u>Critical Save/Restore Register 0</u></p> <p>This register is used to save machine state on critical interrupts, and to restore machine state when an rfc_i is executed. When a critical interrupt is taken, the CSRR0 is set to the current or next instruction address. When rfc_i is executed, instruction execution continues at the address in CSRR0. In general, CSRR0 contains the address of the instruction that caused the critical interrupt, or the address of the instruction to return to after a critical interrupt is serviced.</p>
62:63	///	0b00	<u>Reserved</u>

14.5.15 CSRR1 - Critical Save/Restore Register 1

Register Short Name:	CSRR1	Read Access:	Hypv
Decimal SPR Number:	59	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	CM	0b0	<u>Computation Mode</u> 0 The processor runs in 32-bit mode 1 The processor runs in 64-bit mode
33:34	///	0b00	<u>Reserved</u>
35	GS	0b0	<u>Guest State</u> 0 The processor is in hypervisor state if MSR[PR] = 0. 1 The processor is in guest state.
36	///	0b0	<u>Reserved</u>
37	UCLE	0b0	<u>User Cache Locking Enable</u> 0 Cache Locking instructions are privileged 1 Cache Locking instructions can be executed in user mode (MSR[PR]=1)
38	SPV	0b0	<u>Vector Available</u> 0 The processor cannot execute any Vector instruction 1 The processor can execute Vector instructions
39:45	///	0x0	<u>Reserved</u>
46	CE	0b0	<u>Critical Enable</u> 0 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are disabled 1 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are enabled
47	///	0b0	<u>Reserved</u>
48	EE	0b0	<u>External Enable</u> 0 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are disabled. 1 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are enabled. When an interrupt masked by MSR[EE] is directed to the hypervisor state, the interrupt is enabled if MSR[EE]=1 or MSR[GS]=1 except for Guest Processor Doorbell which is enabled if MSR[EE]=1 and MSR[GS]=1. When an interrupt masked by MSR[EE] is directed to the guest state, the interrupt is enabled if MSR[EE]=1 and MSR[GS]=1.
49	PR	0b0	<u>Problem State</u> 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.) 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource
50	FP	0b0	<u>Floating-Point Available</u> 0 The processor cannot execute any floating-point instructions, including floating-point loads, stores and moves. 1 The processor can execute floating-point instructions.

Bit(s):	Field Name:	Init	Description
51	ME	0b0	<u>Machine Check Enable</u> 0 Machine Check interrupts are disabled 1 Machine Check interrupts are enabled
52	FE0	0b0	<u>Floating-Point Exception Mode 0</u> Sets Floating-Point Exception Mode
53	///	0b0	<u>Reserved</u>
54	DE	0b0	<u>Debug Interrupt Enable</u> 0 Debug interrupts are disabled 1 Debug interrupts are enabled if DBCR0[IDM]=1
55	FE1	0b0	<u>Floating-Point Exception Mode 1</u> Sets Floating-Point Exception Mode
56:57	///	0b00	<u>Reserved</u>
58	IS	0b0	<u>Instruction Address Space</u> 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	0b0	<u>Data Address Space</u> 0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry) 1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry)
60:63	///	0b0000	<u>Reserved</u>

14.5.16 CTR - Count Register

Register Short Name:	CTR	Read Access:	Any
Decimal SPR Number:	9	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	CTR	0x0	<p><u>Counter</u></p> <p>The Count Register (CTR) is a 64-bit register. It can be used to hold a loop count that can be decremented during execution of Branch instructions that contain an appropriately coded BO field. If the value in the Count Register is 0 before being decremented, it is -1 afterward. The Count Register can also be used to provide the branch target address for the Branch Conditional to Count Register instruction</p>

14.5.17 DAC1 - Data Address Compare 1

Register Short Name:	DAC1	Read Access:	Hypv
Decimal SPR Number:	316	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DAC1	0x0	<u>Data Address Compare 1</u> A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified, or to blocks of addresses specified by the combination of the DAC1 and DAC2.

14.5.18 DAC2 - Data Address Compare 2

Register Short Name:	DAC2	Read Access:	Hypv
Decimal SPR Number:	317	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DAC2	0x0	<u>Data Address Compare 2</u> A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified, or to blocks of addresses specified by the combination of the DAC1 and DAC2.

14.5.19 DAC3 - Data Address Compare 3

Register Short Name:	DAC3	Read Access:	Hypv
Decimal SPR Number:	849	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DAC3	0x0	<u>Data Address Compare 3</u> A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified, or to blocks of addresses specified by the combination of the DAC3 and DAC4.

14.5.20 DAC4 - Data Address Compare 4

Register Short Name:	DAC4	Read Access:	Hypv
Decimal SPR Number:	850	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DAC4	0x0	<p><u>Data Address Compare 4</u></p> <p>A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified, or to blocks of addresses specified by the combination of the DAC3 and DAC4.</p>

14.5.21 DBCR0 - Debug Control Register 0

Register Short Name:	DBCR0	Read Access:	Hypv
Decimal SPR Number:	308	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	dcfg

Bit(s):	Field Name:	Init	Description
32	EDM ^{RO}	0b0	<u>External Debug Mode</u> ^{RO} Reports the state of external debug mode 0 external debug mode is disabled 1 external debug mode is enabled. External debug mode is set, and the corresponding debug action selected, from the decoded value of PCCR0[DBA] bits.
33	IDM	0b0	<u>Internal Debug Mode</u> Enable internal debug mode. If MSR[DE]=1, then the occurrence of a debug event or the recording of an earlier debug event in the Debug Status Register when MSR[DE]=0 or DBCR0[IDM]=0 will cause a Debug interrupt
34:35	RST	0b00	<u>Reset</u> 00 No Action 01 Reset1 10 Reset2 11 Reset3
36	ICMP	0b0	<u>Instruction Completion Debug Event</u> 0 ICMP debug events are disabled 1 ICMP debug events are enabled when MSR[DE]=1
37	BRT	0b0	<u>Branch Taken Debug Event</u> 0 BRT debug events are disabled 1 BRT debug events are enabled when MSR[DE]=1
38	IRPT	0b0	<u>Interrupt Taken Debug Event Enable</u> 0 IRPT debug events are disabled 1 IRPT debug events are enabled
39	TRAP	0b0	<u>Trap Debug Event Enable</u> 0 TRAP debug events cannot occur 1 TRAP debug events can occur
40	IAC1	0b0	<u>Instruction Address Compare 1 Debug Event Enable</u> 0 IAC1 debug events cannot occur 1 IAC1 debug events can occur
41	IAC2	0b0	<u>Instruction Address Compare 2 Debug Event Enable</u> 0 IAC2 debug events cannot occur 1 IAC2 debug events can occur
42	IAC3	0b0	<u>Instruction Address Compare 3 Debug Event Enable</u> 0 IAC3 debug events cannot occur 1 IAC3 debug events can occur
43	IAC4	0b0	<u>Instruction Address Compare 4 Debug Event Enable</u> 0 IAC4 debug events cannot occur 1 IAC4 debug events can occur

Bit(s):	Field Name:	Init	Description
44:45	DAC1	0b00	<u>Data Address Compare 1 Debug Event Enable</u> 00 Disabled: DAC1 debug events cannot occur 01 Store Only: DAC1 debug events can occur only if a store-type data storage access 10 Load Only: DAC1 debug events can occur only if a load-type data storage access 11 Any: DAC1 debug events can occur on any data storage access
46:47	DAC2	0b00	<u>Data Address Compare 2 Debug Event Enable</u> 00 Disabled: DAC2 debug events cannot occur 01 Store Only: DAC2 debug events can occur only if a store-type data storage access 10 Load Only: DAC2 debug events can occur only if a load-type data storage access 11 Any: DAC2 debug events can occur on any data storage access
48	RET	0b0	<u>Return Debug Event Enable</u> 0 RET debug events cannot occur 1 RET debug events can occur
49:58	///	0x0	<u>Reserved</u>
59:60	DAC3	0b00	<u>Data Address Compare 3 Debug Event Enable</u> 00 Disabled: DAC3 debug events cannot occur 01 Store Only: DAC3 debug events can occur only if a store-type data storage access 10 Load Only: DAC3 debug events can occur only if a load-type data storage access 11 Any: DAC3 debug events can occur on any data storage access
61:62	DAC4	0b00	<u>Data Address Compare 4 Debug Event Enable</u> 00 Disabled: DAC4 debug events cannot occur 01 Store Only: DAC4 debug events can occur only if a store-type data storage access 10 Load Only: DAC4 debug events can occur only if a load-type data storage access 11 Any: DAC4 debug events can occur on any data storage access
63	FT	0b0	<u>Freeze Timers on Debug Event</u> 0 Enable clocking of timers 1 Disable clocking of timers if any DBSR bit is set (except MRR)

14.5.22 DBCR1 - Debug Control Register 1

Register Short Name:	DBCR1	Read Access:	Hypv
Decimal SPR Number:	309	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	IAC1US	0b00	<u>Instruction Address Compare 1 User/Supervisor Mode</u> 00 Enabled: IAC1 debug events can occur 01 Reserved 10 Enabled PR0: IAC1 debug events can occur only if MSR[PR]=0 11 Enabled PR1: IAC1 debug events can occur only if MSR[PR]=1
34:35	IAC1ER	0b00	<u>Instruction Address Compare 1 Effective/Real Mode</u> 00 Effective: IAC1 debug events are based on effective addresses 01 Not Implemented: 10 Effective IS0: IAC1 debug events are based on effective addresses and if MSR[IS]=0 11 Effective IS1: IAC1 debug events are based on effective addresses and if MSR[IS]=1
36:37	IAC2US	0b00	<u>Instruction Address Compare 2 User/Supervisor Mode</u> 00 Enabled: IAC2 debug events can occur 01 Reserved 10 Enabled PR0: IAC2 debug events can occur only if MSR[PR]=0 11 Enabled PR1: IAC2 debug events can occur only if MSR[PR]=1
38:39	IAC2ER	0b00	<u>Instruction Address Compare 2 Effective/Real Mode</u> 00 Effective: IAC2 debug events are based on effective addresses 01 Not Implemented 10 Effective IS0: IAC2 debug events are based on effective addresses and if MSR[IS]=0 11 Effective IS1: IAC2 debug events are based on effective addresses and if MSR[IS]=1
40	///	0b0	<u>Reserved</u>
41	IAC12M	0b0	<u>Instruction Address Compare 1/2 Mode</u> 0 Exact address compare 1 Address bit match
42:47	///	0x0	<u>Reserved</u>
48:49	IAC3US	0b00	<u>Instruction Address Compare 3 User/Supervisor Mode</u> 00 Enabled: IAC3 debug events can occur 01 Reserved 10 Enabled PR0: IAC3 debug events can occur only if MSR[PR]=0 11 Enabled PR1: IAC3 debug events can occur only if MSR[PR]=1
50:51	IAC3ER	0b00	<u>Instruction Address Compare 3 Effective/Real Mode</u> 00 Effective: IAC3 debug events are based on effective addresses 01 Not Implemented 10 Effective IS0: IAC3 debug events are based on effective addresses and if MSR[IS]=0 11 Effective IS1: IAC3 debug events are based on effective addresses and if MSR[IS]=1
52:53	IAC4US	0b00	<u>Instruction Address Compare 4 User/Supervisor Mode</u> 00 Enabled: IAC4 debug events can occur 01 Reserved 10 Enabled PR0: IAC4 debug events can occur only if MSR[PR]=0 11 Enabled PR1: IAC4 debug events can occur only if MSR[PR]=1

Bit(s):	Field Name:	Init	Description
54:55	IAC4ER	0b00	<u>Instruction Address Compare 4 Effective/Real Mode</u> 00 Effective: IAC4 debug events are based on effective addresses 01 Not Implemented 10 Effective IS0: IAC4 debug events are based on effective addresses and if MSR[IS]=0 11 Effective IS1: IAC4 debug events are based on effective addresses and if MSR[IS]=1
56	///	0b0	<u>Reserved</u>
57	IAC34M	0b0	<u>Instruction Address Compare 3/4 Mode</u> 0 Exact address compare 1 Address bit match
58:63	///	0x0	<u>Reserved</u>

14.5.23 DBCR2 - Debug Control Register 2

Register Short Name:	DBCR2	Read Access:	Hypv
Decimal SPR Number:	310	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	DAC1US	0b00	<u>Data Address Compare 1 User/Supervisor Mode</u> 00 Enabled: DAC1 debug events can occur 01 Reserved: 10 Enabled PR0: DAC1 debug events can occur only if MSR[PR]=0 11 Enabled PR1: DAC1 debug events can occur only if MSR[PR]=1
34:35	DAC1ER	0b00	<u>Data Address Compare 1 Effective/Real Mode</u> 00 Effective: DAC1 debug events are based on effective addresses 01 Not Implemented 10 Effective DS0: DAC1 debug events are based on effective and if MSR[DS]=0 11 Effective DS1: DAC1 debug events are based on effective and if MSR[DS]=1
36:37	DAC2US	0b00	<u>Data Address Compare 2 User/Supervisor Mode</u> 00 Enabled: DAC2 debug events can occur 01 Reserved 10 Enabled PR0: DAC2 debug events can occur only if MSR[PR]=0 11 Enabled PR1: DAC2 debug events can occur only if MSR[PR]=1
38:39	DAC2ER	0b00	<u>Data Address Compare 2 Effective/Real Mode</u> 00 Effective: DAC2 debug events are based on effective addresses 01 Not Implemented 10 Effective DS0: DAC2 debug events are based on effective and if MSR[DS]=0 11 Effective DS1: DAC2 debug events are based on effective and if MSR[DS]=1
40	///	0b0	<u>Reserved</u>
41	DAC12M	0b0	<u>Data Address Compare 1/2 Mode</u> 0 Exact: Exact address compare 1 Bit Match: Address bit match
42:43	///	0b00	<u>Reserved</u>
44:45	DVC1M	0b00	<u>Data Value Compare 1 Mode</u> 00 DVC Disabled: DAC1 debug events can occur 01 DVC All: DAC1 debug events can occur only when all bytes specified by DVC1BE in the data value of the data storage access match their corresponding bytes in DVC1 10 DVC Any: DAC1 debug events can occur only when at least one of the bytes specified by DVC1BE in the data value of the data storage access matches its corresponding byte in DVC1 11 DVC HW: DAC1 debug events can occur only when all bytes specified in DVC1BE within at least one of the halfwords of the data value of the data storage access matches their corresponding bytes in DVC1

Bit(s):	Field Name:	Init	Description
46:47	DVC2M	0b00	<p><u>Data Value Compare 2 Mode</u></p> <p>00 DVC Disabled: DAC2 debug events can occur</p> <p>01 DVC All: DAC2 debug events can occur only when all bytes specified in by DVC2BE in the data value of the data storage access match their corresponding bytes in DVC2</p> <p>10 DVC Any: DAC2 debug events can occur only when at least one of the bytes specified by DVC2BE in the data value of the data storage access matches its corresponding byte in DVC2</p> <p>11 DVC HW: DAC2 debug events can occur only when all bytes specified in DVC2BE within at least one of the halfwords of the data value of the data storage access matches their corresponding bytes in DVC2</p>
48:55	DVC1BE	0x0	<p><u>Data Value Compare 1 Byte Enables</u></p> <p>Specifies which bytes in the aligned data value being read or written by the storage access are compared to the corresponding bytes in DVC1</p>
56:63	DVC2BE	0x0	<p><u>Data Value Compare 2 Byte Enables</u></p> <p>Specifies which bytes in the aligned data value being read or written by the storage access are compared to the corresponding bytes in DVC2</p>

14.5.24 DBCR3 - Debug Control Register 3

Register Short Name:	DBCR3	Read Access:	Hypv
Decimal SPR Number:	848	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	DAC3US	0b00	<u>Data Address Compare 3 User/Supervisor Mode</u> 00 Enabled: DAC3 debug events can occur 01 Reserved 10 Enabled PR0: DAC3 debug events can occur only if MSR[PR]=0 11 Enabled PR1: DAC3 debug events can occur only if MSR[PR]=1
34:35	DAC3ER	0b00	<u>Data Address Compare 3 Effective/Real Mode</u> 00 Effective: DAC3 debug events are based on effective addresses 01 Not Implemented 10 Effective DS0: DAC3 debug events are based on effective and if MSR[DS]=0 11 Effective DS1: DAC3 debug events are based on effective and if MSR[DS]=1
36:37	DAC4US	0b00	<u>Data Address Compare 4 User/Supervisor Mode</u> 00 Enabled: DAC4 debug events can occur 01 Reserved 10 Enabled PR0: DAC4 debug events can occur only if MSR[PR]=0 11 Enabled PR1: DAC4 debug events can occur only if MSR[PR]=1
38:39	DAC4ER	0b00	<u>Data Address Compare 4 Effective/Real Mode</u> 00 Effective: DAC4 debug events are based on effective addresses 01 Not Implemented 10 Effective DS0: DAC4 debug events are based on effective and if MSR[DS]=0 11 Effective DS1: DAC4 debug events are based on effective and if MSR[DS]=1
40	///	0b0	<u>Reserved</u>
41	DAC34M	0b0	<u>Data Address Compare 3/4 Mode</u> 0 Exact address compare 1 Address bit match
42:62	///	0x0	<u>Reserved</u>
63	IVC	0b0	<u>Instruction Value Compare Event</u> 0 Instruction value compare events disabled 1 Instruction value compare events enabled

14.5.25 DBSR - Debug Status Register

Register Short Name:	DBSR	Read Access:	Hypv
Decimal SPR Number:	304	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	WC
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	IDE	0b0	<u>Imprecise Debug Event</u> Set to 1 if MSRDE=0 and a debug event causes its respective Debug Status Register bit to be set to 1
33	UDE	0b0	<u>Unconditional Debug Event</u> Set to 1 if an Unconditional debug event occurred
34:35	MRR	0b00	<u>Most Recent Reset</u> Set to one of three values when a reset occurs: 00 No Action 01 Reset1 10 Reset2 11 Reset3
36	ICMP	0b0	<u>Instruction Complete Debug Event</u> Set to 1 if an Instruction Completion debug event occurred and DBCR0[ICMP]=1
37	BRT	0b0	<u>Branch Taken Debug Event</u> Set to 1 if a Branch Taken debug event occurred and DBCR0[BRT]=1
38	IRPT	0b0	<u>Interrupt Taken Debug Event</u> Set to 1 if an Interrupt Taken debug event occurred and DBCR0[IRPT]=1
39	TRAP	0b0	<u>Trap Instruction Debug Event</u> Set to 1 if a Trap Instruction debug event occurred and DBCR0[TRAP]=1
40	IAC1	0b0	<u>Instruction Address Compare 1 Debug Event</u> Set to 1 if an IAC1 debug event occurred and DBCR0[IAC1]=1
41	IAC2	0b0	<u>Instruction Address Compare 2 Debug Event</u> Set to 1 if an IAC2 debug event occurred and DBCR0[IAC2]=1
42	IAC3	0b0	<u>Instruction Address Compare 3 Debug Event</u> Set to 1 if an IAC3 debug event occurred and DBCR0[IAC3]=1
43	IAC4	0b0	<u>Instruction Address Compare 4 Debug Event</u> Set to 1 if an IAC4 debug event occurred and DBCR0[IAC4]=1
44	DAC1R	0b0	<u>Data Address Compare 1 Read Debug Event</u> Set to 1 if a read-type DAC1 debug event occurred and DBCR0[DAC1]=0b10 or DBCR0[DAC1]=0b11
45	DAC1W	0b0	<u>Data Address Compare 1 Write Debug Event</u> Set to 1 if a write-type DAC1 debug event occurred and DBCR0[DAC1]=0b01 or DBCR0[DAC1]=0b11
46	DAC2R	0b0	<u>Data Address Compare 2 Read Debug Event</u> Set to 1 if a read-type DAC2 debug event occurred and DBCR0[DAC2]=0b10 or DBCR0[DAC2]=0b11
47	DAC2W	0b0	<u>Data Address Compare 2 Write Debug Event</u> Set to 1 if a write-type DAC2 debug event occurred and DBCR0[DAC2]=0b01 or DBCR0[DAC2]=0b11

Bit(s):	Field Name:	Init	Description
48	RET	0b0	<u>Return Debug Event</u> Set to 1 if a Return debug event occurred and DBCR0[RET]=1
49:58	///	0x0	<u>Reserved</u>
59	DAC3R	0b0	<u>Data Address Compare 3 Read Debug Event</u> Set to 1 if a read-type DAC3 debug event occurred and DBCR0[DAC3]=0b10 or DBCR0[DAC3]=0b11
60	DAC3W	0b0	<u>Data Address Compare 3 Write Debug Event</u> Set to 1 if a write-type DAC3 debug event occurred and DBCR0[DAC3]=0b01 or DBCR0[DAC3]=0b11
61	DAC4R	0b0	<u>Data Address Compare 4 Read Debug Event</u> Set to 1 if a read-type DAC4 debug event occurred and DBCR0[DAC4]=0b10 or DBCR0[DAC4]=0b11
62	DAC4W	0b0	<u>Data Address Compare 4 Write Debug Event</u> Set to 1 if a write-type DAC4 debug event occurred and DBCR0[DAC4]=0b01 or DBCR0[DAC4]=0b11
63	IVC	0b0	<u>Instruction Value Compare Event</u> Set to 1 if an IVC debug event occurred with DBCR3[IVC]=1

14.5.26 DBSRWR - Debug Status Register Write Register

Register Short Name:	DBSRWR	Read Access:	None
Decimal SPR Number:	306	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	IDE	0b0	<u>Imprecise Debug Event</u> Sets corresponding DBSR bit
33	UDE	0b0	<u>Unconditional Debug Event</u> Sets corresponding DBSR bit
34:35	MRR	0b00	<u>Most Recent Reset</u> Sets corresponding DBSR bit
36	ICMP	0b0	<u>Instruction Complete Debug Event</u> Sets corresponding DBSR bit
37	BRT	0b0	<u>Branch Taken Debug Event</u> Sets corresponding DBSR bit
38	IRPT	0b0	<u>Interrupt Taken Debug Event</u> Sets corresponding DBSR bit
39	TRAP	0b0	<u>Trap Instruction Debug Event</u> Sets corresponding DBSR bit
40	IAC1	0b0	<u>Instruction Address Compare 1 Debug Event</u> Sets corresponding DBSR bit
41	IAC2	0b0	<u>Instruction Address Compare 2 Debug Event</u> Sets corresponding DBSR bit
42	IAC3	0b0	<u>Instruction Address Compare 3 Debug Event</u> Sets corresponding DBSR bit
43	IAC4	0b0	<u>Instruction Address Compare 4 Debug Event</u> Sets corresponding DBSR bit
44	DAC1R	0b0	<u>Data Address Compare 1 Read Debug Event</u> Sets corresponding DBSR bit
45	DAC1W	0b0	<u>Data Address Compare 1 Write Debug Event</u> Sets corresponding DBSR bit
46	DAC2R	0b0	<u>Data Address Compare 2 Read Debug Event</u> Sets corresponding DBSR bit
47	DAC2W	0b0	<u>Data Address Compare 2 Write Debug Event</u> Sets corresponding DBSR bit
48	RET	0b0	<u>Return Debug Event</u> Sets corresponding DBSR bit
49:58	///	0x0	<u>Reserved</u>
59	DAC3R	0b0	<u>Data Address Compare 3 Read Debug Event</u> Sets corresponding DBSR bit

Bit(s):	Field Name:	Init	Description
60	DAC3W	0b0	<u>Data Address Compare 3 Write Debug Event</u> Sets corresponding DBSR bit
61	DAC4R	0b0	<u>Data Address Compare 4 Read Debug Event</u> Sets corresponding DBSR bit
62	DAC4W	0b0	<u>Data Address Compare 4 Write Debug Event</u> Sets corresponding DBSR bit
63	IVC	0b0	<u>Instruction Value Compare Event</u> Sets corresponding DBSR bit

14.5.27 DEAR - Data Exception Address Register

Register Short Name:	DEAR	Read Access:	Priv
Decimal SPR Number:	61	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GDEAR	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DEAR	0x0	<p><u>Data Exception Address Register</u></p> <p>The DEAR contains the address that was referenced by a Load, Store or Cache Management instruction that caused an Alignment, Data TLB Miss, or Data Storage interrupt.</p>

14.5.28 DEC - Decrementer

Register Short Name:	DEC	Read Access:	Hypv
Decimal SPR Number:	22	Write Access:	Hypv
Initial Value:	0x000000007ffffff	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	DEC	0x7ffffff	<p><u>Decrementer</u></p> <p>The Decrementer (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a Decrementer interrupt after a programmable delay. The contents of the Decrementer are treated as a signed integer.</p>

14.5.29 DECAR - Decrementer Auto-Reload

Register Short Name:	DECAR	Read Access:	Hypv
Decimal SPR Number:	54	Write Access:	Hypv
Initial Value:	0x000000007ffffff	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	DECAR	0x7ffffff	<p><u>Decrementer Auto-Reload</u></p> <p>If TCR[ARE]=1, TSR[DIS] is set to 1, the contents of the Decrementer Auto-Reload Register is then placed into the DEC, and the Decrementer continues decrementing from the reloaded value</p>

14.5.30 DNHDR - Debug Notify Halt Data Register

Register Short Name:	DNHDR	Read Access:	Hypv
Decimal SPR Number:	855	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	dcfg

Bit(s):	Field Name:	Init	Description
32:47	///	0x0	<u>Reserved</u>
48:52	DUI	0x0	<u>Debug Halt Information</u> This data is captured from the DUI field when a dnh instruction is executed.
53	///	0b0	<u>Reserved</u>
54:63	DUIS	0x0	<u>Secondary Debug Halt Information</u> This data is captured from the DUIS field when a dnh instruction is executed.

14.5.31 DSCR - Data Stream Control Register

Register Short Name:	DSCR	Read Access:	Priv
Decimal SPR Number:	17	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:57	///	0x0	<u>Reserved</u>
58	LSD	0b0	<u>Load Stream Disable</u> This bit disables hardware detection and initiation of load streams.
59	SNSE	0b0	<u>Stride-N Stream Enable</u> This bit enables the hardware detection and initiation of load and store streams that have a stride greater than a single cache block. Such store streams are detected only when SSE is also one.
60	SSE	0b0	<u>Store Stream Enable</u> This bit enables hardware detection and initiation of store Streams
61:63	DPFD	0b000	<u>Default Prefetch Depth</u> This field supplies a prefetch depth for hardware-detected streams and for software-defined streams for which a depth of zero is specified or for which dcbt/dcbtst with TH=1010 is not used in their description. Values and their meanings are as follows. 0 default (LPCRDPDFD) 1 none 2 shallowest 3 shallow 4 medium 5 deep 6 deeper 7 deepest

14.5.32 DVC1 - Data Value Compare 1

Register Short Name:	DVC1	Read Access:	Hypv
Decimal SPR Number:	318	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DVC1	0x0	<p><u>Data Value Compare 1</u></p> <p>A DAC1R, DAC1W debug event may be enabled to occur upon loads or stores of a specific data value specified in DVC1. DBCR2[DVC1M] and DBCR2[DVC1BE] control how the contents of the DVC1 is compared with the value.</p>

14.5.33 DVC2 - Data Value Compare 2

Register Short Name:	DVC2	Read Access:	Hypv
Decimal SPR Number:	319	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	DVC2	0x0	<p><u>Data Value Compare 2</u></p> <p>A DAC2R, DAC2W debug event may be enabled to occur upon loads or stores of a specific data value specified in DVC2. DBCR2[DVC2M] and DBCR2[DVC2BE] control how the contents of the DVC2 is compared with the value.</p>

14.5.34 EPCR - Embedded Processor Control Register

Register Short Name:	EPCR	Read Access:	Hypv
Decimal SPR Number:	307	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	EXTGS	0b0	<p><u>External Input interrupt directed to Guest State</u></p> <p>Controls whether an External Input Interrupt is taken in the guest state or the hypervisor state.</p> <p>0 External Inputs interrupts are directed to the hypervisor state. External Input interrupts pend until MSR[GS]=1 or MSR[EE]=1.</p> <p>1 External Inputs interrupts are directed to the guest state. External Input interrupts pend until MSR[GS]=1 and MSR[EE]=1.</p>
33	DTLBGS	0b0	<p><u>Data TLB Error interrupt directed to Guest State</u></p> <p>Controls whether a Data TLB Error Interrupt that occurs in the guest state is taken in the guest state or the hypervisor state.</p> <p>0 Data TLB Error Interrupts that occur in the guest state are directed to the hypervisor state.</p> <p>1 Data TLB Error Interrupts that occur in the guest state are directed to the guest state.</p>
34	ITLBGS	0b0	<p><u>Instruction TLB Error interrupt directed to Guest State</u></p> <p>Controls whether an Instruction TLB Error Interrupt that occurs in the guest state is taken in the guest state or the hypervisor state.</p> <p>0 Instruction TLB Error Interrupts that occur in the guest state are directed to the hypervisor state.</p> <p>1 Instruction TLB Error Interrupts that occur in the guest state are directed to the guest state.</p>
35	DSIGS	0b0	<p><u>Data Storage interrupt directed to Guest State</u></p> <p>Controls whether a Data Storage Interrupt that occurs in the guest state is taken in the guest state or the hypervisor state.</p> <p>0 Data Storage Interrupts that occur in the guest state are directed to the hypervisor state.</p> <p>1 Data Storage Interrupts that occur in the guest state are directed to the guest state, except for a Data Storage Interrupt due to a TLB Ineligible exception is directed to the hypervisor state, regardless of the existence of other exceptions that cause a Data Storage interrupt.</p>
36	ISIGS	0b0	<p><u>Instruction Storage interrupt directed to Guest State</u></p> <p>Controls whether an Instruction Storage Interrupt that occurs in the guest state is taken in the guest state or the hypervisor state.</p> <p>0 Instruction Storage Interrupts that occur in the guest state are directed to the hypervisor state.</p> <p>1 Instruction Storage Interrupts that occur in the guest state are directed to the guest state, except for an Instruction Storage Interrupt due to a TLB Ineligible exception is directed to the hypervisor state, regardless of the existence of other exceptions that cause an Instruction Storage interrupt.</p>
37	DUVD	0b0	<p><u>Disable Hypervisor Debug</u></p> <p>Controls whether Debug Events occur in the hypervisor state.</p> <p>0 Debug events can occur in the hypervisor state.</p> <p>1 Debug events are suppressed in the hypervisor state.</p>

Bit(s):	Field Name:	Init	Description
38	ICM	0b0	<p><u>Interrupt Computation Mode</u></p> <p>Controls the computational mode of the processor when an interrupt occurs that is directed to the hypervisor state. At interrupt time, EHCSR[ICM] is copied into MSR[CM] if the interrupt is directed to the hypervisor state</p> <p>0 Interrupts that are directed to the hypervisor state will execute in 32-bit mode. 1 Interrupts that are directed to the hypervisor state will execute in 64-bit mode.</p>
39	GICM	0b0	<p><u>Guest Interrupt Computation Mode</u></p> <p>Controls the computational mode of the processor when an interrupt occurs that is directed to the guest state. At interrupt time, EHCSR[GICM] is copied into MSR[CM] if the interrupt is directed to the guest state</p> <p>0 Interrupts that are directed to the guest state will execute in 32-bit mode. 1 Interrupts that are directed to the guest state will execute in 64-bit mode.</p>
40	DGTMI	0b0	<p><u>Disable TLB Guest Management Instructions</u></p> <p>Controls whether guest state can execute any TLB management instructions.</p> <p>0 tlbilx, tlbwe, and tlbsrx (for a Logical to Real Address translation hit) are allowed to execute normally when MSR[GS,PR] = 0b10. 1 tlbilx, tlbwe, and tlbsrx always cause an Embedded Hypervisor Privilege Interrupt when MSR[GS,PR] = 0b10.</p>
41	DMIUH	0b0	<p><u>Disable MAS Interrupt Updates for Hypervisor</u></p> <p>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.</p> <p>0 MAS registers are set when a Data or Instruction TLB Error Interrupt or a Data or Instruction Storage Interrupt is taken in the hypervisor. 1 MAS registers updates are disabled and left unchanged when a Data or Instruction TLB Error Interrupt or a Data or Instruction Storage Interrupt is taken in the hypervisor.</p>
42:63	///	0x0	<u>Reserved</u>

14.5.35 EPLC - External Process ID Load Context

Register Short Name:	EPLC	Read Access:	Priv
Decimal SPR Number:	947	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	EPR	0b0	<u>External Load Context PR Bit</u> Used in place of MSR[PR] by the storage access control mechanism when an External Process ID Load instruction is executed. 0 Supervisor mode 1 User mode
33	EAS	0b0	<u>External Load Context AS Bit</u> Used in place of MSR[DS] for translation when an External Process ID Load instruction is executed. 0 Address space 0 1 Address space 1
34	EGS ^{HO}	0b0	<u>External Load Context GS Bit</u> ^{HO} Used in place of MSR[GS] for translation when an External Process ID Load instruction is executed. 0 Embedded Hypervisor state. 1 Guest state. This field is only writable in hypervisor state
35:39	///	0x0	<u>Reserved</u>
40:47	ELPID ^{HO}	0x0	<u>External Load Context Logical Process ID Value</u> ^{HO} Used in place of LPID register 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
48:49	///	0b00	<u>Reserved</u>
50:63	EPID	0x0	<u>External Load Context Process ID Value</u> Used in place of all Process ID register values for translation when an external Process ID Load instruction is executed.

14.5.36 EPSC - External Process ID Store Context

Register Short Name:	EPSC	Read Access:	Priv
Decimal SPR Number:	948	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	EPR	0b0	<u>External Store Context PR Bit</u> Used in place of MSR[PR] by the storage access control mechanism when an External Process ID Store instruction is executed. 0 Supervisor mode 1 User mode
33	EAS	0b0	<u>External Store Context AS Bit</u> Used in place of MSR[DS] for translation when an External Process ID Store instruction is executed. 0 Address space 0 1 Address space 1
34	EGS ^{HO}	0b0	<u>External Store Context GS Bit</u> ^{HO} Used in place of MSR[GS] for translation when an External Process ID Store instruction is executed. 0 Embedded Hypervisor state. 1 Guest state. This field is only writable in hypervisor state
35:39	///	0x0	<u>Reserved</u>
40:47	ELPID ^{HO}	0x0	<u>External Store Context Logical Process ID Value</u> ^{HO} Used in place of LPID register value for load translation when an External PID Store instruction is executed. Compared with TLB[TLPID] during translation. This field is only writable in hypervisor state
48:49	///	0b00	<u>Reserved</u>
50:63	EPID	0x0	<u>External Store Context Process ID Value</u> Used in place of all Process ID register values for translation when an external Process ID Store instruction is executed.

14.5.37 EPTCFG - Embedded Page Table Configuration Register

Register Short Name:	EPTCFG	Read Access:	Hypv
Decimal SPR Number:	350	Write Access:	None
Initial Value:	0x0000000000091942	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:43	///	0x0	<u>Reserved</u>
44:48	PS1	0x12	<u>Page Size 1</u> Indicates whether an indirect entry with page size 2^{PS1} KB combined with sub-page size indicated by SPS1 is supported by the TLB. (A2 supports an indirect page size of 256 MB with sub-page size 64 KB)
49:53	SPS1	0x6	<u>Sub-Page Size 1</u> Indicates whether an indirect entry with sub-page size 2^{SPS1} KB combined with page size indicated by PS1 is supported by the TLB. (A2 supports an indirect page size of 256 MB with sub-page size 64 KB)
54:58	PS0	0xa	<u>Page Size 0</u> Indicates whether an indirect entry with page size 2^{PS0} KB combined with sub-page size indicated by SPS0 is supported by the TLB. (A2 supports an indirect page size of 1 MB with sub-page size 4 KB)
59:63	SPS0	0x2	<u>Sub-Page Size 0</u> Indicates whether an indirect entry with sub-page size 2^{SPS0} KB combined with page size indicated by PS0 is supported by the TLB. (A2 supports an indirect page size of 1 MB with sub-page size 4 KB)

14.5.38 ESR - Exception Syndrome Register

Register Short Name:	ESR	Read Access:	Priv
Decimal SPR Number:	62	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GESR	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:35	///	0b0000	<u>Reserved</u>
36	PIL	0b0	<u>Illegal Instruction exception</u> 1 Indicates Illegal Instruction exception
37	PPR	0b0	<u>Privileged Instruction exception</u> 1 Indicates Privileged Instruction exception
38	PTR	0b0	<u>Trap exception</u> 1 Indicates Trap exception
39	FP	0b0	<u>Floating-point operation</u> 1 Indicates Floating-point operation
40	ST	0b0	<u>Store operation</u> 1 Indicates Store operation
41	///	0b0	<u>Reserved</u>
42	DLK0	0b0	<u>Data Locking Exception 0</u> 1 Indicates a dcbtls, dcbstls, or dcbic instruction was executed with MSR[PR]=1 and MSR[UCLE]=0
43	DLK1	0b0	<u>Data Locking Exception 1</u> 1 Indicates an icbtlc or icblc was executed MSR[PR]=1 and MSR[UCLE]=0
44	AP	0b0	<u>Auxiliary Processor operation</u> 1 Indicates Auxiliary Processor operation
45	PUO	0b0	<u>Unimplemented Operation exception</u> 1 Indicates Unimplemented Operation exception
46	BO	0b0	<u>Byte Ordering exception</u> 1 Indicates Byte Ordering exception
47	PIE	0b0	<u>Imprecise exception</u> 1 Indicates Imprecise exception
48	///	0b0	<u>Reserved</u>
49	UCT	0b0	<u>Unavailable Coprocessor Type</u> 1 indicates that execution of an icswx instruction was attempted which specified a coprocessor type which was marked as unavailable.
50:52	///	0b000	<u>Reserved</u>
53	DATA	0b0	<u>Data Access</u> 1 indicates if the interrupt is due to is a LRAT miss resulting from a Page Table translation of a Load, Store or Cache Management operand address
54	TLBI	0b0	<u>TLB Ineligible</u> 1 indicates a TLB Ineligible exception occurred during a Page Table translation for the instruction causing the interrupt

Bit(s):	Field Name:	Init	Description
55	PT	0b0	<u>Page Table</u> 1 indicates a Page Table Fault or Read or Write Access Control exception occurred during a Page Table translation for the instruction causing the interrupt
56	SPV	0b0	<u>Vector operation</u> 1 Indicates Vector operation
57	EPID	0b0	<u>External Process ID operation</u> 1 indicates the instruction causing the interrupt is an External Process ID instruction
58:63	///	0x0	<u>Reserved</u>

14.5.39 GDEAR - Guest Data Exception Address Register

Register Short Name:	GDEAR	Read Access:	Priv
Decimal SPR Number:	381	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	GDEAR	0x0	<p><u>Guest Data Exception Address Register</u></p> <p>The GDEAR contains the address that was referenced by a Load, Store or Cache Management instruction that caused an Alignment, Data TLB Miss, or Data Storage interrupt when directed to the guest state.</p>

14.5.40 GESR - Guest Exception Syndrome Register

Register Short Name:	GESR	Read Access:	Priv
Decimal SPR Number:	383	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:35	///	0b0000	<u>Reserved</u>
36	PIL	0b0	<u>Illegal Instruction exception</u> 1 Indicates Illegal Instruction exception
37	PPR	0b0	<u>Privileged Instruction exception</u> 1 Indicates Privileged Instruction exception
38	PTR	0b0	<u>Trap exception</u> 1 Indicates Trap exception
39	FP	0b0	<u>Floating-point operation</u> 1 Indicates Floating-point operation
40	ST	0b0	<u>Store operation</u> 1 Indicates Store operation
41	///	0b0	<u>Reserved</u>
42	DLK0	0b0	<u>Data Locking Exception 0</u> 1 Indicates a dcbtls, dcbtstls, or dcbcl instruction was executed in user mode
43	DLK1	0b0	<u>Data Locking Exception 1</u> 1 Indicates an icbtls or icbcl was executed in user mode
44	AP	0b0	<u>Auxiliary Processor operation</u> 1 Indicates Auxiliary Processor operation
45	PUO	0b0	<u>Unimplemented Operation exception</u> 1 Indicates Unimplemented Operation exception
46	BO	0b0	<u>Byte Ordering exception</u> 1 Indicates Byte Ordering exception
47	PIE	0b0	<u>Imprecise exception</u> 1 Indicates Imprecise exception
48	///	0b0	<u>Reserved</u>
49	UCT	0b0	<u>Unavailable Coprocessor Type</u> 1 indicates that execution of an icswx instruction was attempted which specified a coprocessor type which was marked as unavailable in the HACOP or ACOP (if MSR[PR]=1) registers.
50:52	///	0b000	<u>Reserved</u>
53	DATA	0b0	<u>Data Access</u> 1 indicates if the interrupt is due to is a LRAT miss resulting from a Page Table translation of a Load, Store or Cache Management operand address
54	TLBI	0b0	<u>TLB Ineligible</u> 1 indicates a TLB Ineligible exception occurred during a Page Table translation for the instruction causing the interrupt

Bit(s):	Field Name:	Init	Description
55	PT	0b0	<u>Page Table</u> 1 indicates a Page Table Fault or Read or Write Access Control exception occurred during a Page Table translation for the instruction causing the interrupt
56	SPV	0b0	<u>Vector operation</u> 1 Indicates Vector operation
57	EPID	0b0	<u>External Process ID operation</u> 1 indicates the instruction causing the interrupt is an External Process ID instruction
58:63	///	0x0	<u>Reserved</u>

14.5.41 GIVPR - Guest Interrupt Vector Prefix Register

Register Short Name:	GIVPR	Read Access:	Priv
Decimal SPR Number:	447	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:51	GIVPR	0x0	<u>Interrupt Vector Prefix Register</u> Provides the high-order bits of the address of the exception processing routines when in guest state.
52:63	///	0x0	<u>Reserved</u>

14.5.42 GPIR - Guest Processor ID Register

Register Short Name:	GPIR	Read Access:	Priv
Decimal SPR Number:	382	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:49	VPTAG	0x0	<u>Virtual Processor Tag</u> Storage used by the guest OS to identify the virtual processor on which the OS is running
50:63	DBTAG	0x0	<u>Doorbell Tag</u> Used to match guest doorbell messages that are sent to all the processors and virtual processors in a coherence domain. If a sent guest doorbell message tag matches the DBTAG field, a guest doorbell is said to be accepted on the [virtual] processor.

14.5.43 GSPRG0 - Guest Software Special Purpose Register 0

Register Short Name:	GSPRG0	Read Access:	Priv
Decimal SPR Number:	368	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	GSPRG0	0x0	<u>Guest Software Special Purpose Register 0</u> A SPR for software use which has no defined functionality

14.5.44 GSPRG1 - Guest Software Special Purpose Register 1

Register Short Name:	GSPRG1	Read Access:	Priv
Decimal SPR Number:	369	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	GSPRG1	0x0	<u>Guest Software Special Purpose Register 1</u> A SPR for software use which has no defined functionality

14.5.45 GSPRG2 - Guest Software Special Purpose Register 2

Register Short Name:	GSPRG2	Read Access:	Priv
Decimal SPR Number:	370	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	GSPRG2	0x0	<u>Guest Software Special Purpose Register 2</u> A SPR for software use which has no defined functionality

14.5.46 GSPRG3 - Guest Software Special Purpose Register 3

Register Short Name:	GSPRG3	Read Access:	Priv
Decimal SPR Number:	371	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	GSPRG3	0x0	<u>Guest Software Special Purpose Register 3</u> A SPR for software use which has no defined functionality

14.5.47 GSRR0 - Guest Save/Restore Register 0

Register Short Name:	GSRR0	Read Access:	Priv
Decimal SPR Number:	378	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	GSRR0	0x0	<u>Guest Save/Restore Register 0</u> This register is used to save machine state on interrupts directed to the guest state, and to restore machine state when an rfgi is executed. When an interrupt is taken, the GSRR0 is set to the current or next instruction address. When rfgi is executed, instruction execution continues at the address in GSRR0. In general, GSRR0 contains the address of the instruction that caused the interrupt, or the address of the instruction to return to after a critical interrupt is serviced.
62:63	///	0b00	<u>Reserved</u>

14.5.48 GSRR1 - Guest Save/Restore Register 1

Register Short Name:	GSRR1	Read Access:	Priv
Decimal SPR Number:	379	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	CM	0b0	<u>Computation Mode</u> 0 The processor runs in 32-bit mode 1 The processor runs in 64-bit mode
33:34	///	0b00	<u>Reserved</u>
35	GS	0b0	<u>Guest State</u> When set, indicates the processor is running in the Guest State under the control of an hypervisor program. 0 The processor is not running in the guest state. 1 The processor is running in the guest state.
36	///	0b0	<u>Reserved</u>
37	UCLE	0b0	<u>User Cache Locking Enable</u> 0 Cache Locking instructions are privileged 1 Cache Locking instructions can be executed in user mode (MSR[PR]=1)
38	SPV	0b0	<u>Vector Available</u> 0 The processor cannot execute any Vector instruction 1 The processor can execute Vector instructions
39:45	///	0x0	<u>Reserved</u>
46	CE	0b0	<u>Critical Enable</u> 0 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are disabled 1 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are enabled
47	///	0b0	<u>Reserved</u>
48	EE	0b0	<u>External Enable</u> 0 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are disabled. 1 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are enabled. When an interrupt masked by MSR[EE] is directed to the hypervisor state, the interrupt is enabled if MSR[EE]=1 or MSR[GS]=1 except for Guest Processor Doorbell which is enabled if MSR[EE]=1 and MSR[GS]=1. When an interrupt masked by MSR[EE] is directed to the guest state, the interrupt is enabled if MSR[EE]=1 and MSR[GS]=1.
49	PR	0b0	<u>Problem State</u> 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.) 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource
50	FP	0b0	<u>Floating-Point Available</u> 0 The processor cannot execute any floating-point instructions, including floating-point loads, stores and moves. 1 The processor can execute floating-point instructions.

Bit(s):	Field Name:	Init	Description
51	ME	0b0	<u>Machine Check Enable</u> 0 Machine Check interrupts are disabled 1 Machine Check interrupts are enabled
52	FE0	0b0	<u>Floating-Point Exception Mode 0</u> Sets Floating-Point Exception Mode
53	///	0b0	<u>Reserved</u>
54	DE	0b0	<u>Debug Interrupt Enable</u> 0 Debug interrupts are disabled 1 Debug interrupts are enabled if DBCR0[IDM]=1
55	FE1	0b0	<u>Floating-Point Exception Mode 1</u> Sets Floating-Point Exception Mode
56:57	///	0b00	<u>Reserved</u>
58	IS	0b0	<u>Instruction Address Space</u> 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	0b0	<u>Data Address Space</u> 0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry) 1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry)
60:63	///	0b0000	<u>Reserved</u>

14.5.49 HACOP - Hypervisor Available Coprocessor

Register Short Name:	HACOP	Read Access:	Priv
Decimal SPR Number:	351	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:63	CT	0x0	<p><u>Coprocessor Type</u></p> <p>Indicates available coprocessor types for the icswx instruction. Bit n of the register indicates availability of coprocessor type n.</p> <p>0 Coprocessor unavailable. Accesses will generate an unavailable coprocessor type data storage interrupt.</p> <p>1 Coprocessor available.</p>

14.5.50 IAC1 - Instruction Address Compare 1

Register Short Name:	IAC1	Read Access:	Hypv
Decimal SPR Number:	312	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	IAC1	0x0	<u>Instruction Address Compare 1</u> A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified, or to blocks of addresses specified by the combination of the IAC1 and IAC2. Since all instruction addresses are required to be word-aligned, the two low-order bits of the Instruction Address Compare Registers are reserved and do not participate in the comparison to the instruction address.
62:63	///	0b00	<u>Reserved</u>

14.5.51 IAC2 - Instruction Address Compare 2

Register Short Name:	IAC2	Read Access:	Hypv
Decimal SPR Number:	313	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	IAC2	0x0	<u>Instruction Address Compare 2</u> A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified, or to blocks of addresses specified by the combination of the IAC1 and IAC2. Since all instruction addresses are required to be word-aligned, the two low-order bits of the Instruction Address Compare Registers are reserved and do not participate in the comparison to the instruction address.
62:63	///	0b00	<u>Reserved</u>

14.5.52 IAC3 - Instruction Address Compare 3

Register Short Name:	IAC3	Read Access:	Hypv
Decimal SPR Number:	314	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	IAC3	0x0	<u>Instruction Address Compare 3</u> A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified, or to blocks of addresses specified by the combination of the IAC3 and IAC4. Since all instruction addresses are required to be word-aligned, the two low-order bits of the Instruction Address Compare Registers are reserved and do not participate in the comparison to the instruction address.
62:63	///	0b00	<u>Reserved</u>

14.5.53 IAC4 - Instruction Address Compare 4

Register Short Name:	IAC4	Read Access:	Hypv
Decimal SPR Number:	315	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	IAC4	0x0	<u>Instruction Address Compare 4</u> A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified, or to blocks of addresses specified by the combination of the IAC3 and IAC4. Since all instruction addresses are required to be word-aligned, the two low-order bits of the Instruction Address Compare Registers are reserved and do not participate in the comparison to the instruction address.
62:63	///	0b00	<u>Reserved</u>

14.5.54 IAR - Instruction Address Register

Register Short Name:	IAR	Read Access:	Hypv
Decimal SPR Number:	882	Write Access:	Hypv
Initial Value:	0xffffffffffffc	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	bcfg

Bit(s):	Field Name:	Init	Description
0:61	IAR	0x3fffffff ffffff	<u>Instruction Address Register</u> Indicates the address of the current instruction at the completion point, or last instruction which has past the completion point. The completion point is the point at which all interrupts has been process and the instruction is guaranteed to complete.
62:63	///	0b00	<u>Reserved</u>

14.5.55 IESR1 - IU Event Select Register 1

Register Short Name:	IESR1	Read Access:	Priv
Decimal SPR Number:	914	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	INPSELEB0	0b0	<u>Mux Event_Bits(0) Input Select</u> For event mux, bit 0, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:31), 1 = A1_Events(0:31)
33:37	MUXSELEB0	0x0	<u>Mux Event_Bits(0) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 0 to the core event mux. Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
38	INPSELEB1	0b0	<u>Mux Event_Bits(1) Input Select</u> For event mux, bit 1, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:31), 1 = A1_Events(0:31)
39:43	MUXSELEB1	0x0	<u>Mux Event_Bits(1) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 1 to the core event mux. Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
44	INPSELEB2	0b0	<u>Mux Event_Bits(2) Input Select</u> For event mux, bit 2, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:31), 1 = A1_Events(0:31)
45:49	MUXSELEB2	0x0	<u>Mux Event_Bits(2) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 2 to the core event mux. Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
50	INPSELEB3	0b0	<u>Mux Event_Bits(3) Input Select</u> For event mux, bit 3, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:31), 1 = A1_Events(0:31)
51:55	MUXSELEB3	0x0	<u>Mux Event_Bits(3) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 3 to the core event mux. Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
56:63	///	0x0	<u>Reserved</u>

14.5.56 IESR2 - IU Event Select Register 2

Register Short Name:	IESR2	Read Access:	Priv
Decimal SPR Number:	915	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	INPSELEB4	0b0	<u>Mux Event_Bits(4) Input Select</u> For event mux, bit 4, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:31), 1 = B1_Events(0:31)
33:37	MUXSELEB4	0x0	<u>Mux Event_Bits(4) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 4 to the core event mux. Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
38	INPSELEB5	0b0	<u>Mux Event_Bits(5) Input Select</u> For event mux, bit 5, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:31), 1 = B1_Events(0:31)
39:43	MUXSELEB5	0x0	<u>Mux Event_Bits(5) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 5 to the core event mux. Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
44	INPSELEB6	0b0	<u>Mux Event_Bits(6) Input Select</u> For event mux, bit 6, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:31), 1 = B1_Events(0:31)
45:49	MUXSELEB6	0x0	<u>Mux Event_Bits(6) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 6 to the core event mux. Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
50	INPSELEB7	0b0	<u>Mux Event_Bits(7) Input Select</u> For event mux, bit 7, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:31), 1 = B1_Events(0:31)
51:55	MUXSELEB7	0x0	<u>Mux Event_Bits(7) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 7 to the core event mux. Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
56:63	///	0x0	<u>Reserved</u>

14.5.57 IMMR - Instruction Match Mask Register

Register Short Name:	IMMR	Read Access:	Hypv
Decimal SPR Number:	881	Write Access:	Hypv
Initial Value:	0x00000000ffffff	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	MASK	0xffffffff	<u>Instruction Mask</u>

14.5.58 IMPDEP0 - Implementation Dependant Region 0

Register Short Name:	IMPDEP0	Read Access:	Hypv
Decimal SPR Number:	976-991	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N/A
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:	N/A	Scan Ring:	N/A

Bit(s):	Field Name:	Init	Description
0:63	IMPDEP0	0x0	<p><u>Implementation Dependant Fields</u></p> <p>The registers in this range are implemented in the A2 core, but are reserved for attached auxiliary units. If an SPR in this range is not implemented by any attached auxiliary units, mtspr instructions are dropped silently, and mfspr instructions return -1.</p>

14.5.59 IMPDEP1 - Implementation Dependant Region 1

Register Short Name:	IMPDEP1	Read Access:	Priv
Decimal SPR Number:	912-927	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N/A
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:	N/A	Scan Ring:	N/A

Bit(s):	Field Name:	Init	Description
0:63	IMPDEP1	0x0	<p><u>Implementation Dependant Fields</u></p> <p>The registers in this range are implemented in the A2 core, but are reserved for attached auxiliary units. If an SPR in this range is not implemented by any attached auxiliary units, mtspr instructions are dropped silently, and mfspr instructions return -1.</p>

14.5.60 IMR - Instruction Match Register

Register Short Name:	IMR	Read Access:	Hypv
Decimal SPR Number:	880	Write Access:	Hypv
Initial Value:	0x00000000ffffff	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	MATCH	0xffffffff	<u>Instruction Match</u>

14.5.61 IUCR0 - Instruction Unit Configuration Register 0

Register Short Name:	IUCR0	Read Access:	Hypv
Decimal SPR Number:	1011	Write Access:	Hypv
Initial Value:	0x00000000000010fa	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:47	///	0x0	<u>Reserved</u>
48	IC_CLKG_DIS	0b0	<u>Instruction Cache Clock Gating Disable</u> 0 Use clock gating for the instruction cache 1 Disable clock gating for the instruction cache
49	IERAT_CLKG_DIS	0b0	<u>IERAT Clock Gating Disable</u> 0 Use clock gating for the instruction ERAT 1 Disable clock gating for the instruction ERAT
50	CLS ^{RO}	0b0	<u>Cacheline Size</u> ^{RO} 0 L1 Data cache uses 64B cachlines 1 L1 Data cache uses 128B cachelines
51	ICBI_ACK_EN	0b1	<u>ICBI L2 Acknowledge Enable</u> 0 ICBI Acknowledged by A2 1 ICBI Acknowledged by L2
52:55	BP_GS_LEN	0b0000	<u>Gshare History Length</u> Sets length of gshare history
56	BP_BC_EN	0b1	<u>Branch Conditional Predict Enable</u> 1 Enables prediction for branch conditional instructions
57	BP_BCLR_EN	0b1	<u>bclr Predict Enable</u> 1 Enables prediction for branch to link register instructions
58	BP_BCCTR_EN	0b1	<u>bcctr Predict Enable</u> 1 Enables prediction (or hold thread?) for branch to count register instructions
59	BP_SW_EN	0b1	<u>Software Predict Enable</u> 1 Enables software prediction (hints)
60	BP_DY_EN	0b1	<u>Dynamic Predict Enable</u> 1 Enables dynamic hardware prediction (branch history table)
61	BP_ST_EN	0b0	<u>Static Predict Enable</u> 1 Enables static hardware prediction (always predict taken)
62	BP_TI_EN	0b1	<u>Branch History Table Thread Isolation Enable</u> 1 Enables thread isolation for branch history table
63	BP_GS_EN	0b0	<u>Gshare Enable</u> 1 Enables gshare for branch history table

14.5.62 IUCR1 - Instruction Unit Configuration Register 1

Register Short Name:	IUCR1	Read Access:	Hypv
Decimal SPR Number:	883	Write Access:	Hypv
Initial Value:	0x0000000000001000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:49	///	0x0	<u>Reserved</u>
50:51	HIPRI	0b01	<u>High Priority privilege Level</u> A2 has three priority values implemented in hardware. This field configures which value in PPR32[PRI] corresponds to the implementations highest priority. 00 medium normal 01 medium high 10 high 11 very high
52:57	///	0x0	<u>Reserved</u>
58:63	THRES	0x0	<u>Low Priority Minimum Issue Count</u> Sets the amount of cycles between low priority issues, which is set by PPR32[PRI]. The number of cycles is equal to THRES*4. This field is not used when a thread is set to high or medium priority.

14.5.63 IUCR2 - Instruction Unit Configuration Register 2

Register Short Name:	IUCR2	Read Access:	Hypv
Decimal SPR Number:	884	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:39	AXU	0x0	<u>AXU Implementation Dependant Bits</u> These bits are defined for AXU implementation use. See AXU specification for use.
40:63	///	0x0	<u>Reserved</u>

14.5.64 IUDBG0 - Instruction Unit Debug Register 0

Register Short Name:	IUDBG0	Read Access:	Hypv
Decimal SPR Number:	888	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:49	///	0x0	<u>Reserved</u>
50:51	WAY	0b00	<u>Instruction Cache Directory Way Select</u> Selects way for a instruction cache directory read
52:57	ROW	0x0	<u>Instruction Cache Directory Row Select</u> Selects row for a instruction cache directory read
58:61	///	0b0000	<u>Reserved</u>
62	EXEC ^{NP}	0b0	<u>Instruction Cache Directory Read Execute</u> ^{NP} 1 Executes a instruction cache directory read
63	DONE	0b0	<u>Instruction Cache Directory Read Done</u> 1 Indicates a instruction cache directory read operation has completed and IUDBG1/IUDBG2 registers are valid

14.5.65 IUDBG1 - Instruction Unit Debug Register 1

Register Short Name:	IUDBG1	Read Access:	Hypv
Decimal SPR Number:	889	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:52	///	0x0	<u>Reserved</u>
53:55	LRU	0b000	<u>Instruction Cache Directory LRU</u> Indicates value of the LRU in the instruction cache directory
56:59	PARITY	0b0000	<u>Instruction Cache Directory Parity</u> Indicates value of the parity bits in the instruction cache directory
60	ENDIAN	0b0	<u>Instruction Cache Directory Endian</u> 0 Big Endian 1 Little Endian
61:62	///	0b00	<u>Reserved</u>
63	VALID	0b0	<u>Instruction Cache Directory Read Valid</u> 0 directory entry is not valid 1 directory entry is valid

14.5.66 IUDBG2 - Instruction Unit Debug Register 2

Register Short Name:	IUDBG2	Read Access:	Hypv
Decimal SPR Number:	890	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	///	0b00	<u>Reserved</u>
34:63	TAG	0x0	<u>Instruction Cache Directory Tag</u> Indicates value of the tag bit in the instruction cache directory

14.5.67 IULFSR - Instruction Unit LFSR

Register Short Name:	IULFSR	Read Access:	Hypv
Decimal SPR Number:	891	Write Access:	Hypv
Initial Value:	0x000000000000001a	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:59	LFSR	0x1	<u>LFSR random number generator</u>
60	ICRF	0b1	<u>I\$ Select Randomize on Flush</u> 0 Do not randomize 1 Randomize issue priority on a flush for affected threads
61	ICRA	0b0	<u>I\$ Select Randomize Always</u> 0 Do not randomize 1 Continuously randomize issue priority on every cycle (this mode will override Randomize on Flush)
62	ISRF	0b1	<u>Instruction Issue Randomize on Flush</u> 0 Do not randomize 1 Randomize issue priority on a flush for affected threads
63	ISRA	0b0	<u>Instruction Issue Randomize Always</u> 0 Do not randomize 1 Continuously randomize issue priority on every cycle (this mode will override Randomize on Flush)

14.5.68 IULLCR - Instruction Unit Live Lock Control Register

Register Short Name:	IULLCR	Read Access:	Hypv
Decimal SPR Number:	892	Write Access:	Hypv
Initial Value:	0x0000000000020040	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:45	///	0x0	<u>Reserved</u>
46:49	LL_TRIG_DLY	0b1000	<u>IU Live Lock Trigger Delay</u> Sets the number of cycles between events: 0000 Reserved 0001 16,384 - 18,368 0010 32,768 - 35,776 0011 49,152 - 53,184 0100 65,536 - 70,592 0101 81,920 - 88,000 0110 98,304 - 105,408 0111 114,688 - 122,816 1000 131,072 - 140,224 1001 147,456 - 157,632 1010 163,840 - 175,040 1011 180,224 - 192,448 1100 196,608 - 209,856 1101 212,992 - 227,264 1110 229,376 - 244,672 1111 245,760 - 262,080
50:53	///	0b0000	<u>Reserved</u>
54:59	LL_HOLD_DLY	0x4	<u>IU Live Lock Hold Delay</u> Sets the length of time that threads are held. Hold time is equal to 16x the encode in this field. 0 is reserved.
60:62	///	0b000	<u>Reserved</u>
63	IULL_EN	0b0	<u>IU Live Lock Buster Enable</u> When enabled, every TRIG_DLY cycles instruction issue is disabled for HOLD_DLY cycles. Once HOLD_DLY is complete, issue priority is randomized and instruction issue resumes. Bits from IULFSR are used to provide some variability in the trigger delay time.

14.5.69 IVPR - Interrupt Vector Prefix Register

Register Short Name:	IVPR	Read Access:	Hypv
Decimal SPR Number:	63	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:51	IVPR	0x0	<u>Interrupt Vector Prefix Register</u> Provides the high-order bits of the address of the exception processing routines
52:63	///	0x0	<u>Reserved</u>

14.5.70 LESR1 - LQ Event Select Register 1

Register Short Name:	LESR1	Read Access:	Priv
Decimal SPR Number:	920	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	INPSELEB0	0b0	<u>Mux Event_Bits(0) Input Select</u> For event mux, bit 0, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:31), 1 = A1_Events(0:31)
33:37	MUXSELEB0	0x0	<u>Mux Event_Bits(0) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 0 (lq_iu_event_bits(0)). Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
38	INPSELEB1	0b0	<u>Mux Event_Bits(1) Input Select</u> For event mux, bit 1, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:31), 1 = A1_Events(0:31)
39:43	MUXSELEB1	0x0	<u>Mux Event_Bits(1) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 1 (lq_iu_event_bits(1)). Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
44	INPSELEB2	0b0	<u>Mux Event_Bits(2) Input Select</u> For event mux, bit 2, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:31), 1 = A1_Events(0:31)
45:49	MUXSELEB2	0x0	<u>Mux Event_Bits(2) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 2 (lq_iu_event_bits(2)). Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
50	INPSELEB3	0b0	<u>Mux Event_Bits(3) Input Select</u> For event mux, bit 3, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:31), 1 = A1_Events(0:31)
51:55	MUXSELEB3	0x0	<u>Mux Event_Bits(3) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 3 (lq_iu_event_bits(3)). Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
56:63	///	0x0	<u>Reserved</u>

14.5.71 LESR2 - LQ Event Select Register 2

Register Short Name:	LESR2	Read Access:	Priv
Decimal SPR Number:	921	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	INPSELEB4	0b0	<u>Mux Event_Bits(4) Input Select</u> For event mux, bit 4, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:31), 1 = B1_Events(0:31)
33:37	MUXSELEB4	0x0	<u>Mux Event_Bits(4) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 4 (lq_iu_event_bits(4)). Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
38	INPSELEB5	0b0	<u>Mux Event_Bits(5) Input Select</u> For event mux, bit 5, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:31), 1 = B1_Events(0:31)
39:43	MUXSELEB5	0x0	<u>Mux Event_Bits(5) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 5 (lq_iu_event_bits(5)). Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
44	INPSELEB6	0b0	<u>Mux Event_Bits(6) Input Select</u> For event mux, bit 6, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:31), 1 = B1_Events(0:31)
45:49	MUXSELEB6	0x0	<u>Mux Event_Bits(6) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 6 (lq_iu_event_bits(6)). Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
50	INPSELEB7	0b0	<u>Mux Event_Bits(7) Input Select</u> For event mux, bit 7, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:31), 1 = B1_Events(0:31)
51:55	MUXSELEB7	0x0	<u>Mux Event_Bits(7) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 7 (lq_iu_event_bits(7)). Decoded values select Mux 0 ("00000") through Mux 31 ("11111").
56:63	///	0x0	<u>Reserved</u>

14.5.72 LPER - Logical Page Exception Register

Register Short Name:	LPER	Read Access:	Hypv
Decimal SPR Number:	56	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:21	///	0x0	<u>Reserved</u>
22:51	ALPN	0x0	<u>Abbreviated Logical Page Number</u> This field is used to capture the abbreviated logical page number from the PTE translation which caused an LRAT Miss exception.
52:59	///	0x0	<u>Reserved</u>
60:63	LPS	0b0000	<u>Logical Page Size</u> This field is used to capture the logical page size from the PTE translation which caused an LRAT Miss exception.

14.5.73 LPERU - Logical Page Exception Register (Upper)

Register Short Name:	LPERU	Read Access:	Hypv
Decimal SPR Number:	57	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:53	///	0x0	<u>Reserved</u>
54:63	ALPNU	0x0	<u>Abbreviated Logical Page Number (Upper Bits 22:31)</u> This field is used to capture the msb's of the abbreviated logical page number from a PTE translation which caused an LRAT Miss exception (supports 32-bit accesses).

14.5.74 LPIDR - Logical Partition ID Register

Register Short Name:	LPIDR	Read Access:	Hypv
Decimal SPR Number:	338	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:55	///	0x0	<u>Reserved</u>
56:63	LPID	0x0	<u>Logical Partition ID</u> The LPID is part of the virtual address and is used during address translation comparing LPID to the TLPID field in the TLB entry to determine a matching TLB entry.

14.5.75 LR - Link Register

Register Short Name:	LR	Read Access:	Any
Decimal SPR Number:	8	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	LR	0x0	<p><u>Link Register</u></p> <p>The Link Register (LR) is a 64-bit register. It can be used to provide the branch target address for the Branch Conditional to Link Register instruction, and it holds the return address after Branch instructions for which LK=1</p>

14.5.76 LRATCFG - LRAT Configuration Register

Register Short Name:	LRATCFG	Read Access:	Hypv
Decimal SPR Number:	342	Write Access:	None
Initial Value:	0x0000000000542008	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:39	ASSOC	0x0	<u>Associativity</u> Indicates the number of ways that are implemented in this processor's LRAT. This field will always be set to "00000000" for this processor (fully associative LRAT).
40:46	LASIZE	0x2a	<u>Logical Address Size</u> Indicates the number of logical address (LA) bits that are implemented by this processor's LRAT. This field will always be set to "0101010" for this processor (42 bits).
47:49	///	0b000	<u>Reserved</u>
50	LPID	0b1	<u>Logical Partition ID</u> Indicates that the LPID field is supported in the LRAT entries. This bit will always be set to '1' for this processor (A2 does implement the LPID field in LRAT entries).
51	///	0b0	<u>Reserved</u>
52:63	NENTRY	0x8	<u>Number of Entries</u> Indicates the number of entries that are implemented in this processor's LRAT. This field will always be set to "0000_0000_1000" for this processor (8 entries).

14.5.77 LRATPS - LRAT Page Size Register

Register Short Name:	LRATPS	Read Access:	Hypv
Decimal SPR Number:	343	Write Access:	None
Initial Value:	0x0000000051544400	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	///	0b0	<u>Reserved</u>
33	PS30	0b1	<u>Page Size 30</u> Indicates whether a 2 ³⁰ KB (1TB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 1TB page sizes for the LRAT).
34	///	0b0	<u>Reserved</u>
35	PS28	0b1	<u>Page Size 28</u> Indicates whether a 2 ²⁸ KB (256 GB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 256 GB page sizes for the LRAT).
36:38	///	0b000	<u>Reserved</u>
39	PS24	0b1	<u>Page Size 24</u> Indicates whether a 2 ²⁴ KB (16 GB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 16 GB page sizes for the LRAT).
40	///	0b0	<u>Reserved</u>
41	PS22	0b1	<u>Page Size 22</u> Indicates whether a 2 ²² KB (4 GB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 4 GB page sizes for the LRAT).
42	///	0b0	<u>Reserved</u>
43	PS20	0b1	<u>Page Size 20</u> Indicates whether a 2 ²⁰ KB (1 GB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 1 GB page sizes for the LRAT).
44	///	0b0	<u>Reserved</u>
45	PS18	0b1	<u>Page Size 18</u> Indicates whether a 2 ¹⁸ KB (256 MB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 256 MB page sizes for the LRAT).
46:48	///	0b000	<u>Reserved</u>
49	PS14	0b1	<u>Page Size 14</u> Indicates whether a 2 ¹⁴ KB (16 MB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 16 MB page sizes for the LRAT).
50:52	///	0b000	<u>Reserved</u>
53	PS10	0b1	<u>Page Size 10</u> Indicates whether a 2 ¹⁰ KB (1 MB) page size is supported by this processor's LRAT. This bit will always be set to '1' for this processor (A2 supports 1 MB page sizes for the LRAT).
54:63	///	0x0	<u>Reserved</u>

14.5.78 LSUCR0 - Load Store Configuration Register 0

Register Short Name:	LSUCR0	Read Access:	Hypv
Decimal SPR Number:	819	Write Access:	Hypv
Initial Value:	0x00000000000007118	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:48	///	0x0	<u>Reserved</u>
49:51	LCA	0b111	<u>LoadMiss Reload Commit Maximum Attempts</u> Number of Reload attempts for a quadword before the Reservation station creates an issue hole. A value of 0 is reserved
52	///	0b0	<u>Reserved</u>
53:55	SCA	0b001	<u>Store Commit Maximum Attempts</u> Number of Store Commit attempts before the Reservation station creates an issue hole. A value of 0 is reserved.
56:58	///	0b000	<u>Reserved</u>
59	LGE	0b1	<u>Load gathering enabled</u>
60	B2B	0b1	<u>Back-2-Back same L2 type enabled</u>
61	DFWD	0b0	<u>Disable Store Forwarding</u>
62	CLCHK	0b0	<u>Cacheline Check</u>
63	FORD	0b0	<u>Force in Order</u>

14.5.79 MAS0 - MMU Assist Register 0

Register Short Name:	MAS0	Read Access:	Priv
Decimal SPR Number:	624	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	ATSEL	0b0	<u>Array Type Select</u> TLB or LRAT structure selection. When in guest mode (MSR[GS]=1), ATSEL is treated as if it were zero such that the TLB is always selected. 0 - TLB 1 - LRAT
33:44	///	0x0	<u>Reserved</u>
45:47	ESEL	0b000	<u>Entry Select</u> TLB and LRAT Entry selection. Identifies an entry in the selected array to be used for tlbwe and tlbre. Valid values for ESEL are from 0 to TLBnCFG[ASSOC] - 1. When MAS0[ATSEL]=0 (TLB), ESEL becomes effectively a TLB way select and valid values are 0-3 (bit 45 is treated as reserved). When MAS0[ATSEL]=1 (LRAT), valid values are 0-7.
48	///	0b0	<u>Reserved</u>
49	HES	0b0	<u>Hardware Entry Select</u> Determines how the TLB entry way is selected by tlbwe when MAS0[ATSEL] = 0 (TLB). This bit must be set to '0' when MAS0[ATSEL]=1 (LRAT), or an illegal instruction exception may occur for tlbwe. 0 - The TLB entry way is selected by MAS0.ESEL[1:2]. 1 - The TLB entry way is selected by the hardware LRU replacement algorithm.
50:51	WQ	0b00	<u>Write Qualifier</u> Qualifies the TLB write operation performed by tlbwe when MAS0.ATSEL = 0 (TLB). This field must be set to "00" or "11" when MAS0.ATSEL = 1 (LRAT), or an illegal instruction exception may occur for tlbwe. 00 - The selected TLB entry is written regardless of the TLB-reservation. 01 - The selected TLB entry is written if and only if the TLB reservation exists (see tlbsrx instruction). A tlbwe with this value is called a TLB Write Conditional. 10 - The TLB-reservation is cleared; no TLB entry is written. 11 - The selected TLB entry is written regardless of the TLB-reservation.
52:63	///	0x0	<u>Reserved</u>

14.5.80 MAS0_MAS1 - MMU Assist Registers 0 & 1

Register Short Name:	MAS0_MAS1	Read Access:	Priv
Decimal SPR Number:	373	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	MAS0	0x0	<u>MMU Assist Register 0</u> This field is an alias of MAS0(32:63)
32:63	MAS1	0x0	<u>MMU Assist Register 1</u> This field is an alias of MAS1(32:63)

14.5.81 MAS1 - MMU Assist Register 1

Register Short Name:	MAS1	Read Access:	Priv
Decimal SPR Number:	625	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	V	0b0	<u>Valid</u> TLB or LRAT Valid bit: 0 - This TLB or LRAT entry is invalid 1 - This TLB or LRAT entry is valid
33	IPROT	0b0	<u>Invalidate Protect</u> Indicates this TLB entry is protected from invalidate operations due to tlbivax or tlbix instructions, or remote invalidate snoops from other processors. IPROT is not implemented in the LRAT array entries. 0 - This TLB entry is not protected 1 - This TLB entry is protected
34:47	TID	0x0	<u>Translation Identifier</u> This TLB entry's identifier used to compare against the current value of the PID during translations, or against the MAS6.SPID value for searches.
48:49	///	0b00	<u>Reserved</u>
50	IND	0b0	<u>Indirect</u> Designates this TLB entry as an indirect entry which is used by the hardware table walker (HTW) to compute real addresses into the page table. IND is not implemented in the LRAT array entries. 0 - This TLB entry is used to directly translate virtual to real addresses 1 - This TLB entry is used by the HTW as an indirect page table pointer
51	TS	0b0	<u>Translation Space</u> This TLB entry's address space identifier used to compare against the current value of the MSR.IS or DS bit during translations, or against the MAS6.SAS value for searches. TS is not implemented in the LRAT array entries.
52:55	TSIZE	0b0000	<u>Translation Size</u> The selected TLB entry (when MAS0.ATSEL = 0) or LRAT entry (when MAS0.ATSEL = 1) page size value. This implementation supports five page sizes for direct TLB entries (IND=0). All other non-specified page size encodings are treated as reserved. IND=0 direct TLB entries: 0001 = 4KB, 0011 = 64KB, 0101 = 1MB, 0111 = 16MB, 1010 = 1GB, Others = Reserved This implementation supports two page sizes for indirect TLB entries (IND=1). All other non-specified page size encodings are treated as reserved. IND=1 indirect TLB entries: 0101 = 1MB, 1001 = 256MB, Others = Reserved This implementation supports 8 page sizes for LRAT entries. All other non-specified page size encodings are treated as reserved. LRAT entries: 0101 = 1MB, 0111 = 16MB, 1001 = 256MB, 1010 = 1GB, 1011 = 4GB, 1100 = 16GB, 1110 = 256GB, 1111 = 1TB, Others = Reserved

Bit(s):	Field Name:	Init	Description
56:63	///	0x0	<u>Reserved</u>

14.5.82 MAS2 - MMU Assist Register 2

Register Short Name:	MAS2	Read Access:	Priv
Decimal SPR Number:	626	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:51	EPN	0x0	<p><u>Effective Page Number</u> For TLB entries (MAS0.ATSEL=0), this field is used to transfer the entry effective page number. Only bits associated with a page size boundary are significant. The other bits are treated as an offset within this page and ignored.</p> <p>This field is treated as a logical page number (LPN) for LRAT entries (MAS0.ATSEL=1) and used to transfer the LRAT.LPN value.</p> <p>The upper EPN(0:31) bits are instantiated in the 64-bit A2 implementation.</p>
52:58	///	0x0	<u>Reserved</u>
59	W	0b0	<p><u>Write Through</u> This page's write-through storage attribute 0 - This page is not write-through required storage 1 - This page is write-through required storage</p>
60	I	0b0	<p><u>Caching Inhibited</u> This page's caching inhibited storage attribute 0 - This page is not caching inhibited required storage 1 - This page is caching inhibited required storage</p>
61	M	0b0	<p><u>Memory Coherence Required</u> This page's memory coherence required storage attribute 0 - This page is not memory coherence required storage 1 - This page is memory coherence required storage</p>
62	G	0b0	<p><u>Guarded</u> This page's guarded storage attribute 0 - This page is not guarded storage 1 - This page is guarded storage</p>
63	E	0b0	<p><u>Endianness</u> This page's endianness storage attribute 0 - This page is accessed in big endian byte order 1 - This page is accessed in little endian byte order</p>

14.5.83 MAS2U - MMU Assist Register 2 (Upper)

Register Short Name:	MAS2U	Read Access:	Priv
Decimal SPR Number:	631	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	EPNU	0x0	<p><u>Effective Page Number (Upper Bits 0:31)</u> This field is an alias to MAS2.EPN(0:31) and is primarily for use by 32b machine state (CM=0) software.</p> <p>For TLB entries (MAS0.ATSEL=0), this field is to transfer the entry effective page number upper bits 0:31.</p> <p>For LRAT entries (MAS0.ATSEL=1), this field is treated as a logical page number (LPN) and used to transfer the LRAT.LPN(22:31) value.</p>

14.5.84 MAS3 - MMU Assist Register 3

Register Short Name:	MAS3	Read Access:	Priv
Decimal SPR Number:	627	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:52	RPNL	0x0	<p><u>Real Page Number (Lower Bits 32:52)</u> For TLB entries (MAS0.ATSEL=0), this field is used to transfer the lsb's of the entry real page number. RPNL(52) is only significant for indirect TLB entries (IND=1), and is unused for direct TLB entries (IND=0) and LRAT entries (i.e. the TLB entry RPNL(52) bit can only be written to a '1' by a tlbre instruction when MAS1.IND=1 and MAS0.ATSEL=0, and this bit will always be set to '0' after a tlbre instruction if the chosen TLB entry contains IND=0).</p> <p>RPNL(32:51) is treated as a TLB direct entry logical page number (LPNL) when an LRAT is present and enabled. RPNL(32:51) is treated as a real page number (RPNL) for LRAT entries (MAS0.ATSEL=1) and used to transfer the LRAT.RPN(32:51) value. Only bits associated with a particular TLB or LRAT entry page size boundary are significant. The other bits are treated as an offset within this page and ignored. The upper RPNL(22:31) bits are instantiated in the 64-bit A2 implementation in MAS7.</p>
53	///	0b0	<u>Reserved</u>
54	U0	0b0	<p><u>User Definable Storage Attribute 0</u> Specifies a system-dependent storage attribute for this TLB entry. This field is not implemented in LRAT entries. This field has no effect within the A2 core.</p>
55	U1	0b0	<p><u>User Definable Storage Attribute 1</u> Specifies a system-dependent storage attribute for this TLB entry. This field is not implemented in LRAT entries. This field has no effect within the A2 core.</p>
56	U2	0b0	<p><u>User Definable Storage Attribute 2</u> Specifies a system-dependent storage attribute for this TLB entry. This field is not implemented in LRAT entries. This field has no effect within the A2 core.</p>
57	U3	0b0	<p><u>User Definable Storage Attribute 3</u> Specifies a system-dependent storage attribute for this TLB entry. This field is not implemented in LRAT entries. This field has no effect within the A2 core.</p>
58	UX/SPSIZE0	0b0	<p><u>User Mode Execute Enable (IND=0) / SPSIZE0 (IND=1)</u> For direct TLB (IND=0) entries, specifies user mode (MSR.PR=1) execute access permission. See the appropriate access control section of this document for the definition of execute access. 0 - This page does not have execute access permission in user mode (problem state) 1 - This page has execute access permission in user mode (problem state)</p> <p>For indirect TLB (IND=1) entries, specifies sub-page size bit 0.</p>
59	SX/SPSIZE1	0b0	<p><u>Supervisor Mode Execute Enable (IND=0) / SPSIZE1 (IND=1)</u> For direct TLB (IND=0) entries, specifies supervisor mode (MSR.PR=0) execute access permission. See the appropriate access control section of this document for the definition of execute access. 0 - This page does not have execute access permission in supervisor mode (privileged state) 1 - This page has execute access permission in supervisor mode (privileged state)</p> <p>For indirect TLB (IND=1) entries, specifies sub-page size bit 1.</p>

Bit(s):	Field Name:	Init	Description
60	UW/SPSIZE2	0b0	<p><u>User Mode Execute Enable (IND=0) / SPSIZE2 (IND=1)</u></p> <p>For direct TLB (IND=0) entries, specifies user mode (MSR.PR=1) write access permission. See the appropriate access control section of this document for the definition of execute access.</p> <p>0 - This page does not have execute access permission in user mode (problem state) 1 - This page has execute access permission in user mode (problem state)</p> <p>For indirect TLB (IND=1) entries, specifies sub-page size bit 2.</p>
61	SW/SPSIZE3	0b0	<p><u>Supervisor Mode Write Enable (IND=0) / SPSIZE3 (IND=1)</u></p> <p>For direct TLB (IND=0) entries, specifies supervisor mode (MSR.PR=0) write access permission. See the appropriate access control section of this document for the definition of write access.</p> <p>0 - This page does not have write access permission in supervisor mode (privileged state) 1 - This page has write access permission in supervisor mode (privileged state)</p> <p>For indirect TLB (IND=1) entries, specifies sub-page size bit 3.</p>
62	UR/SPSIZE4	0b0	<p><u>User Mode Read Enable (IND=0) / SPSIZE4 (IND=1)</u></p> <p>For direct TLB (IND=0) entries, specifies user mode (MSR.PR=1) read access permission. See the appropriate access control section of this document for the definition of read access.</p> <p>0 - This page does not have read access permission in user mode (problem state) 1 - This page has read access permission in user mode (problem state)</p> <p>For indirect TLB (IND=1) entries, specifies sub-page size bit 4 (treated as reserved by A2 which implements only power of 4 x 1K sub-page sizes).</p>
63	SR/UND	0b0	<p><u>Supervisor Mode Read Enable (IND=0) / UND (IND=1)</u></p> <p>For direct TLB (IND=0) entries, specifies supervisor mode (MSR.PR=0) read access permission. See the appropriate access control section of this document for the definition of read access.</p> <p>0 - This page does not have read access permission in supervisor mode (privileged state) 1 - This page has read access permission in supervisor mode (privileged state)</p> <p>For indirect TLB (IND=1) entries, this bit is undefined (UND).</p>

14.5.85 MAS4 - MMU Assist Register 4

Register Short Name:	MAS4	Read Access:	Priv
Decimal SPR Number:	628	Write Access:	Priv
Initial Value:	0x0000000000000100	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:47	///	0x0	<u>Reserved</u>
48	INDD	0b0	<u>Default Indirect Value</u> Specifies the default value loaded into MAS1.IND and MAS6.SIND on a TLB miss exception.
49:51	///	0b000	<u>Reserved</u>
52:55	TSIZED	0b0001	<u>Default Translation Size Value</u> Specifies the default value loaded into MAS1.TSIZE on a TLB miss exception. If MMUCFG.TWC = 1, TSIZED is also the default value loaded into MAS6.ISIZE upon the exception.
56:58	///	0b000	<u>Reserved</u>
59	WD	0b0	<u>Default Write Through Value</u> Specifies the default value loaded into MAS2.W on a TLB miss exception.
60	ID	0b0	<u>Default Caching Inhibited Value</u> Specifies the default value loaded into MAS2.I on a TLB miss exception.
61	MD	0b0	<u>Default Memory Coherence Required Value</u> Specifies the default value loaded into MAS2.M on a TLB miss exception.
62	GD	0b0	<u>Default Guarded Value</u> Specifies the default value loaded into MAS2.G on a TLB miss exception.
63	ED	0b0	<u>Default Endianess Value</u> Specifies the default value loaded into MAS2.E on a TLB miss exception.

14.5.86 MAS5 - MMU Assist Register 5

Register Short Name:	MAS5	Read Access:	Hypv
Decimal SPR Number:	339	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	SGS	0b0	<u>Search Guest State</u> Specifies the GS value used when searching the TLB during execution of tbsx and tbsrx, and for selecting TLB entries to be invalidated by tbivax or tbilx. The SGS field is compared with the TGS field of each TLB entry to find a matching entry.
33:55	///	0x0	<u>Reserved</u>
56:63	SLPID	0x0	<u>Search Logical Partition Identifier</u> Specifies the LPID value used when searching the TLB during execution of tbsx and tbsrx, and for selecting TLB entries to be invalidated by tbivax or tbilx. The SLPID field is compared with the TLPID field of each TLB entry to find a matching entry.

14.5.87 MAS5_MAS6 - MMU Assist Registers 5 & 6

Register Short Name:	MAS5_MAS6	Read Access:	Hypv
Decimal SPR Number:	348	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	MAS5	0x0	<u>MMU Assist Register 5</u> This field is an alias of MAS5(32:63)
32:63	MAS6	0x0	<u>MMU Assist Register 6</u> This field is an alias of MAS6(32:63)

14.5.88 MAS6 - MMU Assist Register 6

Register Short Name:	MAS6	Read Access:	Priv
Decimal SPR Number:	630	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	///	0b00	<u>Reserved</u>
34:47	SPID	0x0	<u>Search Process Identifier</u> Specifies the value of PID used when searching the TLB during execution of tlbisx. It also defines the PID of the TLB entry to be invalidated by tlbilx with T=1 or T=3 and tlbivax.
48:51	///	0b0000	<u>Reserved</u>
52:55	ISIZE	0b0000	<u>Invalidate Size</u> Specifies the page size of the TLB entry to be invalidated by tlbilx T=3 or tlbivax.
56:61	///	0x0	<u>Reserved</u>
62	SIND	0b0	<u>Search Indirect</u> Specifies the value of IND used when searching the TLB during execution of tlbisx. It also defines the IND of the TLB entry to be invalidated by tlbilx T=3 and tlbivax.
63	SAS	0b0	<u>Search Address Space</u> Specifies the value of AS used when searching the TLB during execution of tlbisx. It also defines the AS of the TLB entry to be invalidated by tlbilx T=3 and tlbivax.

14.5.89 MAS7 - MMU Assist Register 7

Register Short Name:	MAS7	Read Access:	Priv
Decimal SPR Number:	944	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:53	///	0x0	<u>Reserved</u>
54:63	RPNU	0x0	<u>Real Page Number (Upper Bits 22:31)</u> For TLB entries (MAS0.ATSEL=0), this field is used to transfer the msb's of the entry real page number. This value is treated as a TLB entry logical page number (LPNU) when an LRAT is present and enabled. This field is treated as a real page number (RPNU) for LRAT entries (MAS0.ATSEL=1) and used to transfer the LRAT.RPN(22:31) value.

14.5.90 MAS7_MAS3 - MMU Assist Registers 7 & 3

Register Short Name:	MAS7_MAS3	Read Access:	Priv
Decimal SPR Number:	372	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	MAS7	0x0	<u>MMU Assist Register 7</u> This field is an alias of MAS7(32:63)
32:63	MAS3	0x0	<u>MMU Assist Register 3</u> This field is an alias of MAS3(32:63)

14.5.91 MAS8 - MMU Assist Register 8

Register Short Name:	MAS8	Read Access:	Hypv
Decimal SPR Number:	341	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	TGS	0b0	<u>Translation Guest Space</u> Specifies the value that is written to the TLB entry TGS bit by the execution of tlbwe. This bit is loaded from the TLB entry TGS by the execution of tlbre and by the execution of tlbxs that finds a matching entry. The TLB entry TGS identifies that value of the Guest State associated with the TLB entry.
33	VF	0b0	<u>Translation Virtualization Fault</u> Specifies the value that is written to TLB entry VF bit by the execution of tlbwe. This bit is loaded from the TLB entry VF bit by the execution of tlbre and by the execution of tlbxs that finds a matching entry. VF identifies that the TLB entry is used to provide virtualized memory mapped I/O. A data access that uses a TLB entry with the VF field equal to 1 causes a Data Storage interrupt, regardless of the settings of the permission bits. The resulting Data Storage interrupt is always directed to the embedded hypervisor state, regardless of the EHCSR.DSIGS value.
34:55	///	0x0	<u>Reserved</u>
56:63	TLPID	0x0	<u>Translation Logical Partition Identifier</u> Specifies the value that is written to the TLB entry TLPID field by the execution of tlbwe. This field is loaded from the TLB entry TLPID by the execution of tlbre and by the execution of tlbxs that finds a matching entry. The TLB entry TLPID identifies the logical partition associated with the TLB entry.

14.5.92 MAS8_MAS1 - MMU Assist Registers 8 & 1

Register Short Name:	MAS8_MAS1	Read Access:	Hypv
Decimal SPR Number:	349	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	MAS8	0x0	<u>MMU Assist Register 8</u> This field is an alias of MAS8(32:63)
32:63	MAS1	0x0	<u>MMU Assist Register 1</u> This field is an alias of MAS1(32:63)

14.5.93 MCSR - Machine Check Syndrome Register

Register Short Name:	MCSR	Read Access:	Hypv
Decimal SPR Number:	572	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:47	///	0x0	<u>Reserved</u>
48	DPOVR	0b0	<u>DITC Data Port Overrun Condition</u> 1 Indicates an overrun condition detected on a mtdp instruction.
49	DDMH	0b0	<u>Data Cache Directory Multihit Error</u> 1 Indicates multihit condition detected in data cache directory when enabled by XUCR4[MDDMH]=1
50	TLBIVAXSR	0b0	<u>tlbivax Snoop Reject</u> 1 Indicates that a tlbivax snoop (which is tagged with a local core indication) may be rejected back to the L2 when the snoop's LPID mismatches the current core's LPIDR value. This can only occur when CCR2[NOTLB]=1 or MMUCR1[TLBI_REJ]=1.
51	TLBLRUPE	0b0	<u>TLB LRU Parity Error</u> 1 Indicates Parity Error detected for TLB LRU tlbre, tlbxsx, or reload
52	IL2ECC	0b0	<u>Instruction Cache L2 ECC Error</u> 1 Indicates instruction cache detected an L2 uncorrectable ECC error
53	DL2ECC	0b0	<u>Data Cache L2 ECC Error</u> 1 Indicates data cache detected an L2 uncorrectable ECC error
54	DDPE	0b0	<u>Data Cache Directory Parity Error</u> 1 Indicates Parity Error detected in data cache directory when enabled by XUCR0[MDDP]=1
55	EXT	0b0	<u>External Machine Check</u> 1 Indicates external machine check was asserted
56	DCPE	0b0	<u>Data Cache Parity Error</u> 1 Indicates Parity Error detected in data cache when enabled by XUCR0[MDCP]=1
57	IEMH	0b0	<u>I-ERAT Multi-Hit Error</u> 1 Indicates Multiple Entry Hit Error detected for I-ERAT compare
58	DEMh	0b0	<u>D-ERAT Multi-Hit Error</u> 1 Indicates Multiple Entry Hit Error detected for D-ERAT compare
59	TLBMH	0b0	<u>TLB Multi-Hit Error</u> 1 Indicates Multiple Entry Hit Error detected for TLB compare
60	IEPE	0b0	<u>I-ERAT Parity Error</u> 1 Indicates Parity Error detected for I-ERAT eratre, eratsx, or compare
61	DEPE	0b0	<u>D-ERAT Parity Error</u> 1 Indicates Parity Error detected for D-ERAT eratre, eratsx, or compare
62	TLBPE	0b0	<u>TLB Parity Error</u> 1 Indicates Parity Error detected for TLB tlbre, tlbxsx, or reload
63	///	0b0	<u>Reserved</u>

14.5.94 MCSRR0 - Machine Check Save/Restore Register 0

Register Short Name:	MCSRR0	Read Access:	Hypv
Decimal SPR Number:	570	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	SRR0	0x0	<p><u>Critical Save/Restore Register 0</u></p> <p>Machine Check Save/Restore Register 0 (MCSRR0) is used to save machine state on Machine Check interrupts, and to restore machine state when an rfmci is executed. When a Machine Check interrupt is taken, the MCSRR0 is set to the current or next instruction address. When rfmci is executed, instruction execution continues at the address in MCSRR0. In general, MCSRR0 contains the address of an instruction that was executing or about to be executed when the Machine Check exception occurred.</p>
62:63	///	0b00	<u>Reserved</u>

14.5.95 MCSRR1 - Machine Check Save/Restore Register 1

Register Short Name:	MCSRR1	Read Access:	Hypv
Decimal SPR Number:	571	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	CM	0b0	<u>Computation Mode</u> 0 The processor runs in 32-bit mode 1 The processor runs in 64-bit mode
33:34	///	0b00	<u>Reserved</u>
35	GS	0b0	<u>Guest State</u> 0 The processor is in hypervisor state if MSR[PR] = 0. 1 The processor is in guest state.
36	///	0b0	<u>Reserved</u>
37	UCLE	0b0	<u>User Cache Locking Enable</u> 0 Cache Locking instructions are privileged 1 Cache Locking instructions can be executed in user mode (MSR[PR]=1)
38	SPV	0b0	<u>Vector Available</u> 0 The processor cannot execute any Vector instruction 1 The processor can execute Vector instructions
39:45	///	0x0	<u>Reserved</u>
46	CE	0b0	<u>Critical Enable</u> 0 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are disabled 1 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are enabled
47	///	0b0	<u>Reserved</u>
48	EE	0b0	<u>External Enable</u> 0 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are disabled. 1 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are enabled. When an interrupt masked by MSR[EE] is directed to the hypervisor state, the interrupt is enabled if MSR[EE]=1 or MSR[GS]=1 except for Guest Processor Doorbell which is enabled if MSR[EE]=1 and MSR[GS]=1. When an interrupt masked by MSR[EE] is directed to the guest state, the interrupt is enabled if MSR[EE]=1 and MSR[GS]=1.
49	PR	0b0	<u>Problem State</u> 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.) 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource
50	FP	0b0	<u>Floating-Point Available</u> 0 The processor cannot execute any floating-point instructions, including floating-point loads, stores and moves. 1 The processor can execute floating-point instructions.

Bit(s):	Field Name:	Init	Description
51	ME	0b0	<u>Machine Check Enable</u> 0 Machine Check interrupts are disabled 1 Machine Check interrupts are enabled
52	FE0	0b0	<u>Floating-Point Exception Mode 0</u> Sets Floating-Point Exception Mode
53	///	0b0	<u>Reserved</u>
54	DE	0b0	<u>Debug Interrupt Enable</u> 0 Debug interrupts are disabled 1 Debug interrupts are enabled if DBCR0[IDM]=1
55	FE1	0b0	<u>Floating-Point Exception Mode 1</u> Sets Floating-Point Exception Mode
56:57	///	0b00	<u>Reserved</u>
58	IS	0b0	<u>Instruction Address Space</u> 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	0b0	<u>Data Address Space</u> 0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry) 1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry)
60:63	///	0b0000	<u>Reserved</u>

14.5.96 MESR1 - MMU Event Select Register 1

Register Short Name:	MESR1	Read Access:	Priv
Decimal SPR Number:	916	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	INPSELEB0	0b0	<u>Mux Event_Bits(0) Input Select</u> For event mux, bit 0, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:15), 1 = A1_Events(0:15)
33:36	MUXSELEB0	0b0000	<u>Mux Event_Bits(0) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 0 (mm_iu_event_bits(0)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
37	INPSELEB1	0b0	<u>Mux Event_Bits(1) Input Select</u> For event mux, bit 1, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:15), 1 = A1_Events(0:15)
38:41	MUXSELEB1	0b0000	<u>Mux Event_Bits(1) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 1 (mm_iu_event_bits(1)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
42	INPSELEB2	0b0	<u>Mux Event_Bits(2) Input Select</u> For event mux, bit 2, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:15), 1 = A1_Events(0:15)
43:46	MUXSELEB2	0b0000	<u>Mux Event_Bits(2) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 2 (mm_iu_event_bits(2)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
47	INPSELEB3	0b0	<u>Mux Event_Bits(3) Input Select</u> For event mux, bit 3, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:15), 1 = A1_Events(0:15)
48:51	MUXSELEB3	0b0000	<u>Mux Event_Bits(3) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 3 (mm_iu_event_bits(3)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
52:63	///	0x0	<u>Reserved</u>

14.5.97 MESR2 - MMU Event Select Register 2

Register Short Name:	MESR2	Read Access:	Priv
Decimal SPR Number:	917	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	INPSELEB4	0b0	<u>Mux Event_Bits(4) Input Select</u> For event mux, bit 4, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:15), 1 = B1_Events(0:15)
33:36	MUXSELEB4	0b0000	<u>Mux Event_Bits(4) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 4 (mm_iu_event_bits(4)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
37	INPSELEB5	0b0	<u>Mux Event_Bits(5) Input Select</u> For event mux, bit 5, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:15), 1 = B1_Events(0:15)
38:41	MUXSELEB5	0b0000	<u>Mux Event_Bits(5) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 5 (mm_iu_event_bits(5)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
42	INPSELEB6	0b0	<u>Mux Event_Bits(6) Input Select</u> For event mux, bit 6, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:15), 1 = B1_Events(0:15)
43:46	MUXSELEB6	0b0000	<u>Mux Event_Bits(6) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 6 (mm_iu_event_bits(6)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
47	INPSELEB7	0b0	<u>Mux Event_Bits(7) Input Select</u> For event mux, bit 7, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:15), 1 = B1_Events(0:15)
48:51	MUXSELEB7	0b0000	<u>Mux Event_Bits(7) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 7 (mm_iu_event_bits(7)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
52:63	///	0x0	<u>Reserved</u>

14.5.98 MMUCFG - MMU Configuration Register

Register Short Name:	MMUCFG	Read Access:	Hypv
Decimal SPR Number:	1015	Write Access:	None
Initial Value:	0x000000008558341	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:35	///	0b0000	<u>Reserved</u>
36:39	LPIDSIZE	0b1000	<u>Logical Partition Identifier Size</u> Indicates the number of bits in the LPID register that are implemented by the processor. This field will always be set to "1000" for this processor (8 bits).
40:46	RASIZE	0x2a	<u>Real Address Size</u> Indicates the number of real address (RA)bits that are implemented by the processor. This field will always be set to "0101010" for this processor (42 bits).
47	LRAT	0b1	<u>Logical To Real Address Translation</u> Indicates whether the Embedded.Hypervisor.LRAT category is supported by this processor. This bit is part of the boot configuration ring for this processor. 0 = LRAT array is not supported and RPN fields are treated as real page numbers (not logical addresses). 1 = LRAT array is supported and logical address are translated to real addresses as required.
48	TWC	0b1	<u>TLB Write Conditional</u> Indicates whether the Embedded.TLB Write Conditional category is supported by this processor. This bit is part of the boot configuration ring for this processor. 0 = TLB write conditional operations and reservations are not supported. 1 = TLB write conditional operations and reservations are supported.
49:52	///	0b0000	<u>Reserved</u>
53:57	PIDSIZE	0xd	<u>Process Identifier Size</u> Indicates one less than the number of bits in the PID register that are implemented by the processor. This field will always be set to "01101" for this processor (14 bits in PID).
58:59	///	0b00	<u>Reserved</u>
60:61	NTLBS	0b00	<u>Number of TLBS</u> Indicates one less than the number of TLB structures that are implemented by this processor. This field will always be set to "00" for this processor (1 TLB).
62:63	MAVN	0b01	<u>MMU Architecture Version Number</u> Indicates the version number of the architecture of the MMU implemented by the processor. This field will always be set to "01" for this processor (Version 2.0).

14.5.99 MMUCR0 - Memory Management Unit Control Register 0

Register Short Name:	MMUCR0	Read Access:	Hypv
Decimal SPR Number:	1020	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	ECL	0b0	<u>Extended Class</u> Used to transfer the ExtClass field of the selected ERAT entry.
33	TID_NZ	0b0	<u>Translation ID Non-zero</u> Used to transfer the TID_NZ field of the selected ERAT entry.
34	TGS	0b0	<u>Translation Guest State</u> Used to transfer the TGS bit of the selected ERAT entry.
35	TS	0b0	<u>Translation Space</u> Used to transfer the TS bit of the selected ERAT entry.
36:37	TLBSEL	0b00	<u>TLB Select</u> ERAT structure selection: 00 - Reserved 01 - Reserved 10 - I-ERAT 11 - D-ERAT
38:49	///	0x0	<u>Reserved</u>
50:63	TID	0x0	<u>Translation ID</u> Used to transfer the TID field of the selected ERAT entry

14.5.100 MMUCR1 - Memory Management Unit Control Register 1

Register Short Name:	MMUCR1	Read Access:	Hypv
Decimal SPR Number:	1021	Write Access:	Hypv
Initial Value:	0x00000000c0000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32	IRRE	0b0	<u>I-ERAT LRU Round-Robin Enable</u> 0 - LRU normal operation 1 - LRU atomically increments upon eratwe
33	DRRE	0b0	<u>D-ERAT LRU Round-Robin Enable</u> 0 - LRU normal operation 1 - LRU atomically increments upon eratwe
34	REE	0b0	<u>Reference Exception Enable</u> 0 - Not Enabled 1 - Translation hit with R bit cleared generates Instruction Storage Interrupt or Data Storage Interrupt
35	CEE	0b0	<u>Change Exception Enable</u> 0 - Not Enabled 1 - Translation hit with C bit cleared generates Instruction Storage Interrupt or Data Storage Interrupt
36:37	CSINV	0b11	<u>Context Sync Invalidate</u> This field controls how certain ERAT context affecting instructions affect the invalidation of non-protected (EXTCLASS=0) I-ERAT and D-ERAT entries. See the CSINV field description below for a definition of the set of ERAT context affecting instructions. Bit 36: 0 - ERAT context affecting instructions other than isync will invalidate non-protected ERAT entries (enabled) 1 - ERAT context affecting instructions other than isync do not invalidate ERAT entries (disabled) Bit 37: 0 - The isync instruction will invalidate non-protected ERAT entries (enabled) 1 - The isync instruction does not invalidate ERAT entries (disabled)
38:43	PEI	0x0	<u>Parity Error Inject</u> Parity Error Inject Bits: 0=Normal Parity Calculation; 1=Invert Parity (when writing) 38 - I-ERAT WS=0 Parity Error Inject 39 - I-ERAT WS=1 Parity Error Inject 40 - D-ERAT WS=0 Parity Error Inject 41 - D-ERAT WS=1 Parity Error Inject 42 - TLB Parity Error Inject 43 - TLB LRU Parity Error Inject
44	ICTID	0b0	<u>I-ERAT Class Translation ID Enable</u> 0 - I-ERAT Class field operates as a class ID 1 - I-ERAT Class field operates as TID(0:1) bits (of TID(0:13) total value).
45	ITTID	0b0	<u>I-ERAT ThdID Translation ID Enable</u> 0 - I-ERAT ThdID field operates as a thread ID 1 - I-ERAT ThdID field operates as TID(2:5) bits (of TID(0:13) total value).

Bit(s):	Field Name:	Init	Description
46	DCTID	0b0	<u>D-ERAT Class Translation ID Enable</u> 0 - D-ERAT Class field operates as a class ID 1 - D-ERAT Class field operates as TID(0:1) bits (of TID(0:13) total value).
47	DTTID	0b0	<u>D-ERAT ThdID Translation ID Enable</u> 0 - D-ERAT ThdID field operates as a thread ID 1 - D-ERAT ThdID field operates as TID(2:5) bits (of TID(0:13) total value).
48	DCCD	0b0	<u>D-ERAT Class Compare Disable</u> 0 - D-ERAT Class field is used for normal and external PID translation compares. 1 - D-ERAT Class field is ignored for translation compares (mutually exclusive to using external PID ops).
49	TLBWE_BINV	0b0	<u>TLBWE Back Invalidate</u> 0 - No back invalidates are generated to the ERAT's for tlbwe instructions. 1 - When tlbwe with MAS0[HES]=0 instruction overwrites a valid, direct TLB entry without an exception being generated, send a back invalidate to the ERAT's targeting the old virtual address.
50	TLBI_MSB	0b0	<u>TLB Invalidate Most Significant Bit</u> 0 - TLB invalidate snoops from bus assume EPN[31:51] is significant (EPN[27:30] is ignored). 1 - TLB invalidate snoops from bus assume EPN[27:51] is significant.
51	TLBI_REJ	0b0	<u>TLB Invalidate Reject</u> 0 - TLB invalidate snoops from bus are accepted and compared against LPID values in the TLB. 1 - TLB invalidate snoops from bus compare against LPIDR.LPID value for acceptance or rejection.
52	IERRDET	0b0	<u>I-ERAT Error Detect</u> 0 - No Error Detected 1 - I-ERAT error detected and EEN field contains a snapshot of first entry number with error detected
53	DERRDET	0b0	<u>D-ERAT Error Detect</u> 0 - No Error Detected 1 - D-ERAT error detected and EEN field contains a snapshot of first entry number with error detected
54	TERRDET	0b0	<u>TLB Error Detect</u> 0 - No Error Detected 1 - TLB error detected and EEN field contains a snapshot of first entry number with error detected
55:63	EEN	0x0	<u>Error Entry Number</u> I-ERAT, D-ERAT, or TLB entry number for which first error was found after a read of this register

14.5.101 MMUCR2 - Memory Management Unit Control Register 2

Register Short Name:	MMUCR2	Read Access:	Hypv
Decimal SPR Number:	1022	Write Access:	Hypv
Initial Value:	0x00000000000a7531	Duplicated for MT:	N
Slow SPR:	Y	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:39	CLKG_CTL	0x0	<u>MMU Clock Gating Control</u> Power clock gating overrides for various parts of the MMU. Setting these bits has no functional impact.
40:43	EXT_DBG_SEL	0b0000	<u>MMU Extended Debug Select</u> Alternate debug group selects for the MMU. Refer to the MMU unit debug group and trigger selects section of this document. Bit 40: Alternate debug groups 10 and 11 select Bit 41: Alternate debug groups 12 and 13 select Bit 42: Alternate debug groups 14 and 15 select Bit 43: Alternate debug trigger group 3
44:47	PS4	0b1010	<u>TLB Page Size 4 Select</u> 0000 - Disabled (do not apply the hash for this page size) 0001 - Page size = 4KB 0011 - Page size = 64KB 0101 - Page size = 1MB 0111 - Page size = 16MB 1010 - Page size = 1GB Others - Reserved
48:51	PS3	0b0111	<u>TLB Page Size 3 Select</u> 0000 - Disabled (do not apply the hash for this page size) 0001 - Page size = 4KB 0011 - Page size = 64KB 0101 - Page size = 1MB 0111 - Page size = 16MB 1010 - Page size = 1GB Others - Reserved
52:55	PS2	0b0101	<u>TLB Page Size 2 Select</u> 0000 - Disabled (do not apply the hash for this page size) 0001 - Page size = 4KB 0011 - Page size = 64KB 0101 - Page size = 1MB 0111 - Page size = 16MB 1010 - Page size = 1GB Others - Reserved
56:59	PS1	0b0011	<u>TLB Page Size 1 Select</u> 0000 - Disabled (do not apply the hash for this page size) 0001 - Page size = 4KB 0011 - Page size = 64KB 0101 - Page size = 1MB 0111 - Page size = 16MB 1010 - Page size = 1GB Others - Reserved

Bit(s):	Field Name:	Init	Description
60:63	PS0	0b0001	<u>TLB Page Size 0 Select</u> 0000 - Disabled (do not apply the hash for this page size) 0001 - Page size = 4KB 0011 - Page size = 64KB 0101 - Page size = 1MB 0111 - Page size = 16MB 1010 - Page size = 1GB Others - Reserved

14.5.102 MMUCR3 - Memory Management Unit Control Register 3

Register Short Name:	MMUCR3	Read Access:	Priv
Decimal SPR Number:	1023	Write Access:	Priv
Initial Value:	0x000000000000000f	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:48	///	0x0	<u>Reserved</u>
49	X	0b0	<u>Exclusion Range Enable</u> This bit is used to transfer the X bit to/from the selected TLB entry.
50	R	0b0	<u>Reference</u> This bit is used to transfer the R bit to/from the selected TLB entry.
51	C	0b0	<u>Change</u> This bit is used to transfer the C bit to/from the selected TLB entry.
52	ECL	0b0	<u>Extended Class</u> This field is used to transfer the extended class field to/from the selected TLB entry.
53	TID_NZ	0b0	<u>Translation ID Non-zero</u> This field is used to transfer the TID_NZ field from the selected TLB entry.
54:55	Class	0b00	<u>Class</u> This field is used to transfer the Class field to/from the selected TLB entry.
56:57	WLC	0b00	<u>L1 D-Cache Way Locking Class Attribute</u> This field is used to transfer the WLC bits to/from the selected TLB entry.
58	ResvAttr	0b0	<u>Reserved Attributes</u> This field is used to transfer the reserved attributes to/from the selected TLB entry.
59	///	0b0	<u>Reserved</u>
60:63	ThdID	0b1111	<u>Thread Identifier</u> This field is used to transfer the thread ID field to/from the selected TLB entry.

14.5.103 MMUCSR0 - MMU Control and Status Register 0

Register Short Name:	MMUCSR0	Read Access:	Hypv
Decimal SPR Number:	1012	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:60	///	0x0	<u>Reserved</u>
61	TLB0_FI	0b0	<p><u>TLB 0 Full Invalidate</u></p> <p>This bit controls/indicates when an invalidate all function is requested/in progress.</p> <p>0 - When this bit is read as a '0', there is no invalidate all operation in progress. Writing this bit to a zero while an invalidate all operation is in progress is ignored.</p> <p>1 - When this bit is read as a '1', there is an invalidate all operation in progress. Hardware will set this bit to a zero when the invalidate all operation is completed. Writing this bit to a '1' initiates the invalidate all operation (unless one is already in progress, in which case writing this bit to a '1' is ignored).</p>
62:63	///	0b00	<u>Reserved</u>

14.5.104 MSR - Machine State Register

Register Short Name:	MSR	Read Access:	Priv
Decimal SPR Number:	N/A	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32	CM	0b0	<u>Computation Mode</u> 0 The processor runs in 32-bit mode 1 The processor runs in 64-bit mode
33:34	///	0b00	<u>Reserved</u>
35	GS	0b0	<u>Guest State</u> 0 The processor is in hypervisor state if MSR[PR] = 0. 1 The processor is in guest state. MSR[GS] cannot be changed unless MSR[GS]=0.
36	///	0b0	<u>Reserved</u>
37	UCLE	0b0	<u>User Cache Locking Enable</u> 0 Cache Locking instructions are privileged 1 Cache Locking instructions can be executed in user mode (MSR[PR]=1)
38	SPV	0b0	<u>Vector Available</u> 0 The processor cannot execute any Vector instruction 1 The processor can execute Vector instructions
39:45	///	0x0	<u>Reserved</u>
46	CE	0b0	<u>Critical Enable</u> 0 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are disabled 1 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are enabled
47	///	0b0	<u>Reserved</u>
48	EE	0b0	<u>External Enable</u> 0 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are disabled. 1 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are enabled. When an interrupt masked by MSR[EE] is directed to the hypervisor state, the interrupt is enabled if MSR[EE]=1 or MSR[GS]=1 except for Guest Processor Doorbell which is enabled if MSR[EE]=1 and MSR[GS]=1. When an interrupt masked by MSR[EE] is directed to the guest state, the interrupt is enabled if MSR[EE]=1 and MSR[GS]=1.
49	PR	0b0	<u>Problem State</u> 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.) 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource
50	FP	0b0	<u>Floating-Point Available</u> 0 The processor cannot execute any floating-point instructions, including floating-point loads, stores and moves. 1 The processor can execute floating-point instructions.

Bit(s):	Field Name:	Init	Description
51	ME	0b0	<u>Machine Check Enable</u> 0 Machine Check interrupts are disabled 1 Machine Check interrupts are enabled
52	FE0	0b0	<u>Floating-Point Exception Mode 0</u> Sets Floating-Point Exception Mode
53	///	0b0	<u>Reserved</u>
54	DE	0b0	<u>Debug Interrupt Enable</u> 0 Debug interrupts are disabled 1 Debug interrupts are enabled if DBCR0[IDM]=1
55	FE1	0b0	<u>Floating-Point Exception Mode 1</u> Sets Floating-Point Exception Mode
56:57	///	0b00	<u>Reserved</u>
58	IS	0b0	<u>Instruction Address Space</u> 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	0b0	<u>Data Address Space</u> 0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry) 1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry)
60:63	///	0b0000	<u>Reserved</u>

14.5.105 MSRP - Machine State Register Protect

Register Short Name:	MSRP	Read Access:	Hypv
Decimal SPR Number:	311	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:36	///	0x0	<u>Reserved</u>
37	UCLEP	0b0	<u>User Cache Lock Enable Protect</u> 0 Guest privileged state can modify MSR[UCLE]. 1 Guest privileged state cannot modify MSR[UCLE]. When MSRP[UCLEP] = 1, cache locking instructions are not permitted to execute in the guest privileged state and cause an Embedded Hypervisor Privilege exception when executed.
38:53	///	0x0	<u>Reserved</u>
54	DEP	0b0	<u>Debug Enable Protect</u> 0 Guest privileged state can modify MSR[DE]. 1 Guest privileged state cannot modify MSR[DE].
55:63	///	0x0	<u>Reserved</u>

14.5.106 PID - Process ID

Register Short Name:	PID	Read Access:	Priv
Decimal SPR Number:	48	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:49	///	0x0	<u>Reserved</u>
50:63	PID	0x0	<u>Process ID</u> Process ID Register is used by system software to specify which TLB entries are used by the processor to accomplish address translation for loads, stores, and instruction fetches

14.5.107 PIR - Processor ID Register

Register Short Name:	PIR	Read Access:	Priv
Decimal SPR Number:	286	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GPIR	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:53	///	0x0	<u>Reserved</u>
54:61	CID ^{IO}	0x0	<u>Processor Core ID^{IO}</u> Returns the value of the IO pin an_ac_coreid. This can be used to distinguish a processor core from other processor cores in the system
62:63	TID	0b00	<u>Processor Thread ID</u> This field can be used to distinguish the thread from other threads on the processor. Threads are numbered sequentially, with valid values ranging from 0 to 3.

14.5.108 PPR32 - Program Priority Register

Register Short Name:	PPR32	Read Access:	Any
Decimal SPR Number:	898	Write Access:	Any
Initial Value:	0x000000000000c0000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:42	///	0x0	<u>Reserved</u>
43:45	PRI	0b011	<u>Thread Priority</u> 001 very low (privileged) 010 low 011 medium low 100 medium 101 medium high (privileged) 110 high (privileged) 111 very high (hypervisor) Access violations or writing a value of zero will result in a nop.
46:63	///	0x0	<u>Reserved</u>

14.5.109 PVR - Processor Version Register

Register Short Name:	PVR	Read Access:	Priv
Decimal SPR Number:	287	Write Access:	None
Initial Value:	0x00000000004c0100	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:47	VERSION	0x4c	<u>Processor Version</u>
48:63	REVISION	0x100	<u>Processor Revision</u> DDM.m = 0x0M0m

14.5.110 RESR1 - RV Event Select Register 1

Register Short Name:	RESR1	Read Access:	Priv
Decimal SPR Number:	922	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	INPSELEB0	0b0	<u>Mux Event_Bits(0) Input Select</u> For event mux, bit 0, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:15), 1 = A1_Events(0:15)
33:36	MUXSELEB0	0b0000	<u>Mux Event_Bits(0) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 0 (rv_iu_event_bits(0)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
37	INPSELEB1	0b0	<u>Mux Event_Bits(1) Input Select</u> For event mux, bit 1, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:15), 1 = A1_Events(0:15)
38:41	MUXSELEB1	0b0000	<u>Mux Event_Bits(1) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 1 (rv_iu_event_bits(1)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
42	INPSELEB2	0b0	<u>Mux Event_Bits(2) Input Select</u> For event mux, bit 2, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:15), 1 = A1_Events(0:15)
43:46	MUXSELEB2	0b0000	<u>Mux Event_Bits(2) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 2 (rv_iu_event_bits(2)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
47	INPSELEB3	0b0	<u>Mux Event_Bits(3) Input Select</u> For event mux, bit 3, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:15), 1 = A1_Events(0:15)
48:51	MUXSELEB3	0b0000	<u>Mux Event_Bits(3) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 3 (rv_iu_event_bits(3)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
52:63	///	0x0	<u>Reserved</u>

14.5.111 RESR2 - RV Event Select Register 2

Register Short Name:	RESR2	Read Access:	Priv
Decimal SPR Number:	923	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	INPSELEB4	0b0	<u>Mux Event_Bits(4) Input Select</u> For event mux, bit 4, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:15), 1 = B1_Events(0:15)
33:36	MUXSELEB4	0b0000	<u>Mux Event_Bits(4) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 4 (rv_iu_event_bits(4)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
37	INPSELEB5	0b0	<u>Mux Event_Bits(5) Input Select</u> For event mux, bit 5, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:15), 1 = B1_Events(0:15)
38:41	MUXSELEB5	0b0000	<u>Mux Event_Bits(5) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 5 (rv_iu_event_bits(5)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
42	INPSELEB6	0b0	<u>Mux Event_Bits(6) Input Select</u> For event mux, bit 6, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:15), 1 = B1_Events(0:15)
43:46	MUXSELEB6	0b0000	<u>Mux Event_Bits(6) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 6 (rv_iu_event_bits(6)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
47	INPSELEB7	0b0	<u>Mux Event_Bits(7) Input Select</u> For event mux, bit 7, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:15), 1 = B1_Events(0:15)
48:51	MUXSELEB7	0b0000	<u>Mux Event_Bits(7) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 7 (rv_iu_event_bits(7)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
52:63	///	0x0	<u>Reserved</u>

14.5.112 SIAR - Sampled Instruction Address Register

Register Short Name:	SIAR	Read Access:	Priv
Decimal SPR Number:	796	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:	Y	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SIAR	0x0	<p><u>Sampled Instruction Address Register</u></p> <p>Upon activation of a threads performance monitor alert, sampled instruction address and MSR values are loaded into the SIAR as follows:</p> <p>0:61 <= Sampled Instruction Address. From corresponding bits of the Next Instruction Address register</p> <p>62 <= Sampled MSR[GS]</p> <p>63 <= Sampled MSR[PR]</p>

14.5.113 SPRG0 - Software Special Purpose Register 0

Register Short Name:	SPRG0	Read Access:	Priv
Decimal SPR Number:	272	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GSPRG0	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG0	0x0	<u>Software Special Purpose Register 0</u> A SPR for software use which has no defined functionality

14.5.114 SPRG1 - Software Special Purpose Register 1

Register Short Name:	SPRG1	Read Access:	Priv
Decimal SPR Number:	273	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GSPRG1	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG1	0x0	<u>Software Special Purpose Register 1</u> A SPR for software use which has no defined functionality

14.5.115 SPRG2 - Software Special Purpose Register 2

Register Short Name:	SPRG2	Read Access:	Priv
Decimal SPR Number:	274	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GSPRG2	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG2	0x0	<u>Software Special Purpose Register 2</u> A SPR for software use which has no defined functionality

14.5.116 SPRG3 - Software Special Purpose Register 3

Register Short Name:	SPRG3	Read Access:	Priv/Any
Decimal SPR Number:	275/259	Write Access:	Priv/None
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GSPRG3	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG3	0x0	<u>Software Special Purpose Register 3</u> A SPR for software use which has no defined functionality

14.5.117 SPRG4 - Software Special Purpose Register 4

Register Short Name:	SPRG4	Read Access:	Priv/Any
Decimal SPR Number:	276/260	Write Access:	Priv/None
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG4	0x0	<u>Software Special Purpose Register 4</u> A SPR for software use which has no defined functionality

14.5.118 SPRG5 - Software Special Purpose Register 5

Register Short Name:	SPRG5	Read Access:	Priv/Any
Decimal SPR Number:	277/261	Write Access:	Priv/None
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG5	0x0	<u>Software Special Purpose Register 5</u> A SPR for software use which has no defined functionality

14.5.119 SPRG6 - Software Special Purpose Register 6

Register Short Name:	SPRG6	Read Access:	Priv/Any
Decimal SPR Number:	278/262	Write Access:	Priv/None
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG6	0x0	<u>Software Special Purpose Register 6</u> A SPR for software use which has no defined functionality

14.5.120 SPRG7 - Software Special Purpose Register 7

Register Short Name:	SPRG7	Read Access:	Priv/Any
Decimal SPR Number:	279/263	Write Access:	Priv/None
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG7	0x0	<u>Software Special Purpose Register 7</u> A SPR for software use which has no defined functionality

14.5.121 SPRG8 - Software Special Purpose Register 8

Register Short Name:	SPRG8	Read Access:	Hypv
Decimal SPR Number:	604	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SPRG8	0x0	<u>Software Special Purpose Register 8</u> A SPR for software use which has no defined functionality

14.5.122 SRAMD - Shadowed RAMD Register

Register Short Name:	SRAMD	Read Access:	Any
Decimal SPR Number:	894	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	RO
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:63	SRAMD	0x0	<u>Shadowed RAMD</u> This register is loaded with the updated value of RAMD upon a SCOM write to any of the RAMD register addresses (RAMD, RAMDH or RAMDL).

14.5.123 SRR0 - Save/Restore Register 0

Register Short Name:	SRR0	Read Access:	Priv
Decimal SPR Number:	26	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GSRR0	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:61	SRR0	0x0	<u>Save/Restore Register 0</u> This register is used to save machine state on non-critical interrupts, and to restore machine state when an rfi is executed. On a non-critical interrupt, SRR0 is set to the current or next instruction address. When rfi is executed, instruction execution continues at the address in SRR0. In general, SRR0 contains the address of the instruction that caused the non-critical interrupt, or the address of the instruction to return to after a non-critical interrupt is serviced.
62:63	///	0b00	<u>Reserved</u>

14.5.124 SRR1 - Save/Restore Register 1

Register Short Name:	SRR1	Read Access:	Priv
Decimal SPR Number:	27	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:	GSRR1	Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32	CM	0b0	<u>Computation Mode</u> 0 The processor runs in 32-bit mode 1 The processor runs in 64-bit mode
33:34	///	0b00	<u>Reserved</u>
35	GS	0b0	<u>Guest State</u> 0 The processor is in hypervisor state if MSR[PR] = 0. 1 The processor is in guest state.
36	///	0b0	<u>Reserved</u>
37	UCLE	0b0	<u>User Cache Locking Enable</u> 0 Cache Locking instructions are privileged 1 Cache Locking instructions can be executed in user mode (MSR[PR]=1)
38	SPV	0b0	<u>Vector Available</u> 0 The processor cannot execute any Vector instruction 1 The processor can execute Vector instructions
39:45	///	0x0	<u>Reserved</u>
46	CE	0b0	<u>Critical Enable</u> 0 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are disabled 1 Critical Input, Watchdog Timer, and Processor Doorbell Critical interrupts are enabled
47	///	0b0	<u>Reserved</u>
48	EE	0b0	<u>External Enable</u> 0 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are disabled. 1 External Input, Decrementer, Fixed-Interval Timer, Processor Doorbell, Guest Processor Doorbell, and Performance Monitor interrupts are enabled. When an interrupt masked by MSR[EE] is directed to the hypervisor state, the interrupt is enabled if MSR[EE]=1 or MSR[GS]=1 except for Guest Processor Doorbell which is enabled if MSR[EE]=1 and MSR[GS]=1. When an interrupt masked by MSR[EE] is directed to the guest state, the interrupt is enabled if MSR[EE]=1 and MSR[GS]=1.
49	PR	0b0	<u>Problem State</u> 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.) 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource
50	FP	0b0	<u>Floating-Point Available</u> 0 The processor cannot execute any floating-point instructions, including floating-point loads, stores and moves. 1 The processor can execute floating-point instructions.

Bit(s):	Field Name:	Init	Description
51	ME	0b0	<u>Machine Check Enable</u> 0 Machine Check interrupts are disabled 1 Machine Check interrupts are enabled
52	FE0	0b0	<u>Floating-Point Exception Mode 0</u> Sets Floating-Point Exception Mode
53	///	0b0	<u>Reserved</u>
54	DE	0b0	<u>Debug Interrupt Enable</u> 0 Debug interrupts are disabled 1 Debug interrupts are enabled if DBCR0[IDM]=1
55	FE1	0b0	<u>Floating-Point Exception Mode 1</u> Sets Floating-Point Exception Mode
56:57	///	0b00	<u>Reserved</u>
58	IS	0b0	<u>Instruction Address Space</u> 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	0b0	<u>Data Address Space</u> 0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry) 1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry)
60:63	///	0b0000	<u>Reserved</u>

14.5.125 TB - Timebase

Register Short Name:	TB	Read Access:	Any
Decimal SPR Number:	268	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	TBU	0x0	<u>Timebase Upper</u> Provides access to the upper portion of the Timebase
32:63	TBL	0x0	<u>Timebase Lower</u> Provides access to the lower portion of the Timebase

14.5.126 TBL - Timebase Lower

Register Short Name:	TBL	Read Access:	None
Decimal SPR Number:	284	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	TBL	0x0	<u>Timebase Lower</u> Provides access to the lower portion of the Timebase

14.5.127 TBU - Timebase Upper

Register Short Name:	TBU	Read Access:	None/Any
Decimal SPR Number:	285/269	Write Access:	Hypv/None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	TBU	0x0	<u>Timebase Upper</u> Provides access to the upper portion of the Timebase

14.5.128 TCR - Timer Control Register

Register Short Name:	TCR	Read Access:	Hypv
Decimal SPR Number:	340	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:33	WP	0b00	<u>Watchdog Timer Period</u> Specifies one of 4 bit locations of the TimeBase used to signal a Watchdog Timer exception on a transition from 0 to 1. 00 2 ¹⁹ time base clocks 01 2 ²³ time base clocks 10 2 ²⁵ time base clocks 11 2 ³¹ time base clocks
34:35	WRC	0b00	<u>Watchdog Timer Reset Control</u> 00 NoReset: No Watchdog Timer reset request will occur 01 Reset1 request 10 Reset2 request 11 Reset3 request Note: - Type of reset request to cause upon Watchdog Timer exception with TSR[ENW,WIS]=0b11. - These bits are set only by software. Once a 1 has been written to one of these bits, that bit remains a 1 until a reset request occurs. This is to prevent errant code from disabling the Watchdog reset function.
36	WIE	0b0	<u>Watchdog Timer Interrupt Enable</u> 0 Disable Watchdog Timer interrupt 1 Enable Watchdog Timer interrupt
37	DIE	0b0	<u>Decrementer Interrupt Enable</u> 0 Disable Decrementer interrupt 1 Enable Decrementer interrupt
38:39	FP	0b00	<u>Fixed-Interval Timer Period</u> Specifies one of 4 bit locations of the TimeBase used to signal a Fixed-Interval Timer exception on a transition from 0 to 1. 00 2 ¹¹ time base clocks 01 2 ¹⁵ time base clocks 10 2 ¹⁹ time base clocks 11 2 ²³ time base clocks
40	FIE	0b0	<u>Fixed-Interval Timer Interrupt Enable</u> 0 Disable Fixed Interval Timer interrupt 1 Enable Fixed Interval Timer interrupt
41	ARE	0b0	<u>Auto-Reload Enable</u> 0 Disable auto reload 1 Enable auto reload
42	UDIE	0b0	<u>User Decrementer Interrupt Enable</u> 0 Disable User Decrementer interrupt 1 Enable User Decrementer interrupt

Bit(s):	Field Name:	Init	Description
43:50	///	0x0	<u>Reserved</u>
51	UD	0b0	<u>User Decrementer Available</u> 0 mtspr or mfspr to the UDEC register causes a Illegal Instruction exception 1 mtspr or mfspr to the UDEC register succeeds Note: Changing this bit requires a CSI for the next insturction to see the new context
52:63	///	0x0	<u>Reserved</u>

14.5.129 TENC - Thread Enable Clear Register

Register Short Name:	TENC	Read Access:	Hypv
Decimal SPR Number:	439	Write Access:	Hypv
Initial Value:	0x0000000000000001	Duplicated for MT:	N
Slow SPR:	N	Notes:	WC
Guest Supervisor Mapping:		Scan Ring:	bcfg

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:59	///	0x0	<u>Reserved</u>
60:63	TEN	0b0001	<p><u>Thread Enable Clear</u></p> <p>For $t < 4$, bit 63-t corresponds to thread t. When bit 63-t is set to 1, thread t is disabled, if it is not already. When bit 63-t is set 0, thread t is unaffected.</p> <p>When bit 63-t is read, the current value of the thread enable is returned.</p>

14.5.130 TENS - Thread Enable Set Register

Register Short Name:	TENS	Read Access:	Hypv
Decimal SPR Number:	438	Write Access:	Hypv
Initial Value:	0x0000000000000001	Duplicated for MT:	N
Slow SPR:	N	Notes:	WS
Guest Supervisor Mapping:		Scan Ring:	bcfg

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:59	///	0x0	<u>Reserved</u>
60:63	TEN	0b0001	<u>Thread Enable Set</u> For $t < 4$, bit 63-t corresponds to thread t. When bit 63-t is set to 1, thread t is enabled, if it is not already. When bit 63-t is set 0, thread t is unaffected. When bit 63-t is read, the current value of the thread enable is returned.

14.5.131 TENSr - Thread Enable Status Register

Register Short Name:	TENSr	Read Access:	Hypv
Decimal SPR Number:	437	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:59	///	0x0	<u>Reserved</u>
60:63	TENSr	0b0000	<u>Thread Enable Status Register</u> Bit 63-t of the TENSr corresponds to thread t.

14.5.132 TIR - Thread Identification Register

Register Short Name:	TIR	Read Access:	Hypv
Decimal SPR Number:	446	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32:61	///	0x0	<u>Reserved</u>
62:63	TID	0b00	<u>Processor Thread ID</u> This field can be used to distinguish the thread from other threads on the processor. Threads are numbered sequentially, with valid values ranging from 0 to 3.

14.5.133 TLB0CFG - TLB 0 Configuration Register

Register Short Name:	TLB0CFG	Read Access:	Hypv
Decimal SPR Number:	688	Write Access:	None
Initial Value:	0x00000000407a200	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:39	ASSOC	0x4	<u>Associativity</u> Indicates the number of ways that are implemented in this processor's TLB0. This field will always be set to "0000100" for this processor (4 ways).
40:44	///	0x0	<u>Reserved</u>
45	PT	0b1	<u>Page Table</u> Indicates whether this TLB can be loaded from the hardware page table. This bit is part of the boot configuration ring for this processor. 0=TLB is not eligible to be loaded from the hardware page table (attempts to install page table entries by hardware walker will result in TLB Ineligible exceptions) 1=TLB can be loaded from the hardware page table.
46	IND	0b1	<u>Indirect</u> Indicates that an indirect entry can be created in this TLB and that there is a corresponding EPTCFG register that defines the page sizes and sub-page sizes. This bit is part of the boot configuration ring for this processor. 0=indirect entries are not supported and this TLB treats the IND bit as reserved (this infers software TLB management only). 1=indirect entries are supported (infers hardware page table walking is supported).
47	GTWE	0b1	<u>Guest TLB Write Enable</u> Indicates that a guest supervisor can write to this TLB because an LRAT array exists for this TLB. This bit is part of the boot configuration ring for this processor. 0=Guest cannot write TLB entries without hypervisor intervention 1=Guest may write TLB entries which will be translated via the LRAT.
48	IPROT	0b1	<u>Invalidate Protect</u> Indicates whether invalidation protection is implemented by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 does support the invalidate protect bit in TLB 0 entries).
49	///	0b0	<u>Reserved</u>
50	HES	0b1	<u>Hardware Entry Select</u> Indicates whether hardware entry selection is supported by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 does support hardware calculation of the entry number for TLB 0 for tlwe instructions when MAS0.HES=1).
51	///	0b0	<u>Reserved</u>
52:63	NENTRY	0x200	<u>Number of Entries</u> Indicates the number of entries that are implemented in this processor's TLB 0. This field will always be set to "0010_0000_0000" for this processor (512 entries).

14.5.134 TLB0PS - TLB 0 Page Size Register

Register Short Name:	TLB0PS	Read Access:	Hypv
Decimal SPR Number:	344	Write Access:	None
Initial Value:	0x0000000000104444	Duplicated for MT:	N
Slow SPR:	Y	Notes:	HM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:42	///	0x0	<u>Reserved</u>
43	PS20	0b1	<u>Page Size 20</u> Indicates whether a 2 ²⁰ KB (1 GB) page size is supported by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 supports 1 GB page sizes for TLB 0).
44:48	///	0x0	<u>Reserved</u>
49	PS14	0b1	<u>Page Size 14</u> Indicates whether a 2 ¹⁴ KB (16 MB) page size is supported by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 supports 16 MB page sizes for TLB 0).
50:52	///	0b000	<u>Reserved</u>
53	PS10	0b1	<u>Page Size 10</u> Indicates whether a 2 ¹⁰ KB (1 MB) page size is supported by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 supports 1 MB page sizes for TLB 0).
54:56	///	0b000	<u>Reserved</u>
57	PS6	0b1	<u>Page Size 6</u> Indicates whether a 2 ⁶ KB (64 KB) page size is supported by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 supports 64 KB page sizes for TLB 0).
58:60	///	0b000	<u>Reserved</u>
61	PS2	0b1	<u>Page Size 2</u> Indicates whether a 2 ² KB (4 KB) page size is supported by this processor's TLB 0. This bit will always be set to '1' for this processor (A2 supports 4 KB page sizes for TLB 0).
62:63	///	0b00	<u>Reserved</u>

14.5.135 TRACE - Hardware Trace Macro Control Register

Register Short Name:	TRACE	Read Access:	None
Decimal SPR Number:	1006	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	DATA	0x0	<p><u>Trace Control Data</u></p> <p>This register may be used to control trace features on some L2 implementations. Writing to this register causes a store like transaction on the L2 interface, with a TTYPE of MTSPR_TRACE if the expression (CCR2[EN_TRACE] and (XUCR0[TRACE_UM] or not MSR[PR]))=1 for the executing thread. If the expression is false, the operation is treated as a nop. The data written to this field is placed in the address on the L2 interface according to the table below. See L2 specification for details on this feature.</p> <p>Trace L2 req_ra</p> <p>-----</p> <p>50:59 34:43 60 45 61 48 62 47 63 46</p>

14.5.136 TSR - Timer Status Register

Register Short Name:	TSR	Read Access:	Hypv
Decimal SPR Number:	336	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	WC,AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	ENW	0b0	<u>Enable Next Watchdog Timer</u> 0 Action taken on next Watchdog Timer exception will be to set TSR{ENW} 1 Action taken on next Watchdog Timer exception is governed by TSR{WIS}
33	WIS	0b0	<u>Watchdog Timer Interrupt Status</u> 0 A Watchdog Timer event has not occurred. 1 A Watchdog Timer event has occurred. When (MSR{CE}=1 or MSR{GS}=1) and TCR{WIE}=1, a Watchdog Timer interrupt is taken.
34:35	WRS	0b00	<u>Watchdog Timer Reset Status</u> 00 No Reset: No Watchdog Timer reset has occurred 01 Reset1: A Watchdog Timer initiated Reset1 reset occurred 10 Reset2: A Watchdog Timer initiated Reset2 reset occurred 11 Reset3: A Watchdog Timer initiated Reset3 reset occurred
36	DIS	0b0	<u>Decrementer Interrupt Status</u> A Decrementer event has occurred
37	FIS	0b0	<u>Fixed-Interval Timer Interrupt Status</u> A Fixed-Interval Timer event has occurred
38	UDIS	0b0	<u>User Decrementer Interrupt Status</u> A User Decrementer event has occurred
39:63	///	0x0	<u>Reserved</u>

14.5.137 UDEC - User Decrementer

Register Short Name:	UDEC	Read Access:	Any
Decimal SPR Number:	550	Write Access:	Any
Initial Value:	0x000000007ffffff	Duplicated for MT:	Y
Slow SPR:	N	Notes:	AM
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	UDEC	0x7ffffff	<p><u>User Decrementer</u></p> <p>The User Decrementer (UDEC) is a 32-bit decrementing counter that provides a mechanism for causing a User Decrementer interrupt after a programmable delay. The contents of the User Decrementer are treated as a signed integer.</p> <p>Note: If TCR[UD]=0, this accesses to this register are treated as an illegal SPR.</p>

14.5.138 VRSAVE - Vector Register Save

Register Short Name:	VRSAVE	Read Access:	Any
Decimal SPR Number:	256	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:63	VRSAVE	0x0	<u>Vector Register Save</u> Provided for application and operating system use, may be used to indicate which VRs are currently being used by a program

14.5.139 XER - Fixed Point Exception Register

Register Short Name:	XER	Read Access:	Any
Decimal SPR Number:	1	Write Access:	Any
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
0:31	///	0x0	<u>Reserved</u>
32	SO	0b0	<u>Summary Overflow</u> The Summary Overflow bit is set to 1 whenever an instruction (except mtspr) sets the Overflow bit.
33	OV	0b0	<u>Overflow</u> The Overflow bit is set to indicate that an overflow has occurred during execution of an instruction.
34	CA	0b0	<u>Carry</u> Carry bit from extend arithmetic ops.
35:56	///	0x0	<u>Reserved</u>
57:63	SI	0x0	<u>String Index</u> This field specifies the number of bytes to be transferred by a Load String Indexed or Store String Indexed instruction.

14.5.140 XESR1 - XU Event Select Register 1

Register Short Name:	XESR1	Read Access:	Priv
Decimal SPR Number:	918	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	INPSELEB0	0b0	<u>Mux Event_Bits(0) Input Select</u> For event mux, bit 0, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:15), 1 = A1_Events(0:15)
33:36	MUXSELEB0	0b0000	<u>Mux Event_Bits(0) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 0 (fx0/1_iu_event_bits(0)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
37	INPSELEB1	0b0	<u>Mux Event_Bits(1) Input Select</u> For event mux, bit 1, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:15), 1 = A1_Events(0:15)
38:41	MUXSELEB1	0b0000	<u>Mux Event_Bits(1) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 1 (fx0/1_iu_event_bits(1)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
42	INPSELEB2	0b0	<u>Mux Event_Bits(2) Input Select</u> For event mux, bit 2, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:15), 1 = A1_Events(0:15)
43:46	MUXSELEB2	0b0000	<u>Mux Event_Bits(2) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 2 (fx0/1_iu_event_bits(2)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
47	INPSELEB3	0b0	<u>Mux Event_Bits(3) Input Select</u> For event mux, bit 3, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = A0_Events(0:15), 1 = A1_Events(0:15)
48:51	MUXSELEB3	0b0000	<u>Mux Event_Bits(3) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 3 (fx0/1_iu_event_bits(3)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
52:63	///	0x0	<u>Reserved</u>

14.5.141 XESR2 - XU Event Select Register 2

Register Short Name:	XESR2	Read Access:	Priv
Decimal SPR Number:	919	Write Access:	Priv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	INPSELEB4	0b0	<u>Mux Event_Bits(4) Input Select</u> For event mux, bit 4, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:15), 1 = B1_Events(0:15)
33:36	MUXSELEB4	0b0000	<u>Mux Event_Bits(4) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 4 (fx0/1_iu_event_bits(4)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
37	INPSELEB5	0b0	<u>Mux Event_Bits(5) Input Select</u> For event mux, bit 5, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:15), 1 = B1_Events(0:15)
38:41	MUXSELEB5	0b0000	<u>Mux Event_Bits(5) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 5 (fx0/1_iu_event_bits(5)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
42	INPSELEB6	0b0	<u>Mux Event_Bits(6) Input Select</u> For event mux, bit 6, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:15), 1 = B1_Events(0:15)
43:46	MUXSELEB6	0b0000	<u>Mux Event_Bits(6) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 6 (fx0/1_iu_event_bits(6)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
47	INPSELEB7	0b0	<u>Mux Event_Bits(7) Input Select</u> For event mux, bit 7, determines which group of performance event inputs are selected to drive the bank of 2:1 muxes. 0 = B0_Events(0:15), 1 = B1_Events(0:15)
48:51	MUXSELEB7	0b0000	<u>Mux Event_Bits(7) 2:1 Mux Select</u> Determines which 2:1 mux is gated for driving event mux bit 7 (fx0/1_iu_event_bits(7)). Decoded values select Mux 0 ("0000") through Mux 15 ("1111").
52:63	///	0x0	<u>Reserved</u>

14.5.142 XUCR0 - Execution Unit Configuration Register 0

Register Short Name:	XUCR0	Read Access:	Hypv
Decimal SPR Number:	1014	Write Access:	Hypv
Initial Value:	0x000000000000008c0	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:36	CLKG_CTL	0x0	<u>XU Clock Gating Control</u> Debug feature. Setting a bit to 1 disables the use of clock gating for the specified unit. 0 should be used for maximum power savings. (0) FXUA, DATA (1) CNTRL, DERAT (2) CPL, FXUB (3) L2CMDQ (4) SPR
37:40	TRACE_UM	0b0000	<u>Hardware Trace Control Register User Mode Enable</u> Each bit 37+t, corresponds to thread t: 0 Writes to TRACE SPR in user mode (MSR[PR]=1) behave as a nop 1 Writes to TRACE SPR in user mode (MSR[PR]=1) are enabled
41	MBAR_ACK	0b0	<u>Memory Barrier Acknowledge</u> 0 lwsync and mbar complete internal to the core 1 lwsync and mbar complete after sync_done is received on the A2 L2 interface (same behavior as hwsync).
42	TLBSYNC	0b0	<u>TLBSYNC Ack Behavior</u> 0 tlbsync acknowledged by A2 1 tlbsync acknowledged by L2
43:47	///	0x0	<u>Reserved</u>
48	CLS ^{RO}	0b0	<u>Cacheline Size^{RO}</u> 0 L1 Data cache uses 64B cachlines 1 L1 Data cache uses 128B cachelines
49	AFLSTA	0b0	<u>Force Load/Store Alignment for AXU</u> 0 Normal Operation. Supported misaligned access are handled natively by hardware. 1 An alignment exception occurs on AXU storage access instructions if data address is not on an operand boundary. Note: This bit is ORed with the interface signal iu_xu_is2_axu_ldst_forceexcept
50	MDDP	0b0	<u>Machine Check on Data Cache Directory Parity Error</u> 0 Data cache directory parity errors are recovered by hardware 1 Data cache directory parity errors are recovered by software (machine check interrupt)
51	CRED	0b0	<u>L2 Credit Control</u> 0 No restrictions when the A2 core has 1 store credit and 1 load credit 1 The A2 Core can only send one load or store (but not both) when the A2 core has 1 store credit and 1 load credit NOTE: The A2 core must be quiesced before changing this bit.
52	REL ^{RO}	0b1	<u>L2 Reload Control^{RO}</u> 0 Critical Quadword first and data every other cycle 1 Critical Quadword first and data in back to back cycles Note: This field is read only and can only be set by the chip configuration ring.

Bit(s):	Field Name:	Init	Description
53	MDCP	0b0	<u>Machine Check on Data Cache Parity Error</u> 0 Data cache parity errors are recovered by hardware 1 Data cache parity errors are recovered by software (machine check interrupt)
54	TCS	0b0	<u>Timer Clock Select</u> 0 Core Clock 1 External pulse
55	FLSTA	0b0	<u>Force Load/Store Alignment for Integer</u> 0 Normal Operation. Supported misaligned access are handled natively by hardware. 1 An alignment exception occurs on integer storage access instructions if data address is not on an operand boundary.
56	L2SIW ^{RO}	0b1	<u>L2 Store Interface Width</u> ^{RO} 0 16B 1 32B Note: This field is read only and can only be set by the chip configuration ring.
57	FLH2L2 ^{RO}	0b1	<u>Forward Load Hits to L2</u> ^{RO} 0 Load or Store hits operate normally 1 Force all load hits to be forwarded to L2. Force all store hits to invalidate the L1 cache. Note: This field is read only and can only be set by the chip configuration ring. Note: This bit is ANDed with the interface signal <code>an_ac_flh2l2_gate</code> . See the user's manual for the auxiliary processor implementation for more details.
58	DCDIS	0b0	<u>Data Cache Disable</u> 0 Data Cache is enabled 1 Data Cache is disabled Software Note: Changing the state of <code>DC_DIS</code> does not change the state of the Dcache. In order to maintain Dcache coherency the data cache should be invalidated using a <code>dc</code> instruction before the Dcache is re-enabled.
59	WLK	0b0	<u>Data Cache Way Locking Enable</u> 0 L1 Data Cache way locking is disabled 1 L1 Data Cache way Locking is enabled and TLB[WLC] specifies the L1 Replacement Management Table Entry
60	CSLC	0b0	<u>Cache Snoop Lock Clear</u> Sticky bit set by hardware if a <code>dc</code> bi snoop (either internally or externally generated) invalidated a locked cache block. Note that the lock bit 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 <code>dc</code> bi snoop that invalidated a locked line 1 The cache has encountered a <code>dc</code> bi snoop that invalidated a locked line
61	CUL	0b0	<u>Cache Unable to Lock</u> Sticky bit set by hardware and cleared by writing 0 to this bit location. 0 Indicates a lock set or lock clear instruction was effective in the cache 1 Indicates a lock set or lock clear instruction was not effective in the cache
62	CLO	0b0	<u>Cache Lock Overflow</u> Sticky bit set by hardware and cleared by writing 0 to this bit location. 0 Indicates a lock overflow condition was not encountered in the cache 1 Indicates a lock overflow condition was encountered in the cache

Bit(s):	Field Name:	Init	Description
63	CLFC ^{NP}	0b0	<u>Cache Lock Bits Flash Clear</u> ^{NP} Writing a 1 during a flash clear operation causes an undefined operation. Writing a 0 during a flash clear operation is ignored. Clearing occurs regardless of the enable (CE) value. 0 Default 1 Hardware initiates a cache lock bits flash clear operation. Resets to 0 when the operation completes.

14.5.143 XUCR1 - Execution Unit Configuration Register 1

Register Short Name:	XUCR1	Read Access:	Hypv
Decimal SPR Number:	851	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	Y
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	ccfg

Bit(s):	Field Name:	Init	Description
32:56	///	0x0	<u>Reserved</u>
57:59	LL_TB_SEL	0b000	<u>LiveLock Buster Hang Pulse Timbase Select</u> Selects Pulse for the LiveLock avoidance logic: 000 2 ⁹ time base clocks 001 2 ¹¹ time base clocks 010 2 ¹³ time base clocks 011 2 ¹⁵ time base clocks 100 2 ¹⁹ time base clocks 101 2 ²³ time base clocks 110 2 ²⁵ time base clocks 111 2 ²⁷ time base clocks
60:61	LL_STATE ^{RO}	0b00	<u>LiveLock Buster State</u> ^{RO} Indicates the current state of the LiveLock avoidance logic: 00 Normal Operation 01 Potential Livelock 10 Livelock: attempting forward progress on this thread only 11 Livelock MT: attempting forward progress on another thread
62	LL_SEL	0b0	<u>LiveLock Buster Hang Pulse Select</u> 0 Hang Pulse derived from timbase as selected by LL_TB_SEL 1 Hang Pulse derived from external pulse
63	LL_EN	0b0	<u>LiveLock Buster Disable</u> 0 Disables LiveLock Buster Logic 1 Enables LiveLock Buster Logic

14.5.144 XUCR2 - Execution Unit Configuration Register 2

Register Short Name:	XUCR2	Read Access:	Hypv
Decimal SPR Number:	1016	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:39	RMT3	0x0	<p><u>L1 Replacement Management Table Entry 3</u></p> <p>A RMT entry indicates which set(s) are eligible for replacement for a given data cache miss. Each RMT entry is 8 bits, 1-bit corresponding to each way in the data cache. The value of each bit indicates the following:</p> <p>0 Way is not eligible for replacement 1 Way is eligible for replacement</p>
40:47	RMT2	0x0	<p><u>L1 Replacement Management Table Entry 2</u></p> <p>A RMT entry indicates which set(s) are eligible for replacement for a given data cache miss. Each RMT entry is 8 bits, 1-bit corresponding to each way in the data cache. The value of each bit indicates the following:</p> <p>0 Way is not eligible for replacement 1 Way is eligible for replacement</p>
48:55	RMT1	0x0	<p><u>L1 Replacement Management Table Entry 1</u></p> <p>A RMT entry indicates which set(s) are eligible for replacement for a given data cache miss. Each RMT entry is 8 bits, 1-bit corresponding to each way in the data cache. The value of each bit indicates the following:</p> <p>0 Way is not eligible for replacement 1 Way is eligible for replacement</p>
56:63	RMT0	0x0	<p><u>L1 Replacement Management Table Entry 0</u></p> <p>A RMT entry indicates which set(s) are eligible for replacement for a given data cache miss. Each RMT entry is 8 bits, 1-bit corresponding to each way in the data cache. The value of each bit indicates the following:</p> <p>0 Way is not eligible for replacement 1 Way is eligible for replacement</p>

14.5.145 XUCR4 - Execution Unit Configuration Register 4

Register Short Name:	XUCR4	Read Access:	Hypv
Decimal SPR Number:	853	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	N	Notes:	
Guest Supervisor Mapping:		Scan Ring:	dcfg

Bit(s):	Field Name:	Init	Description
32:45	///	0x0	<u>Reserved</u>
46	MMU_MCHK	0b0	<p><u>MMU Machine Check Control</u></p> <p>This bit is used in conjunction with the CCR2.NOTLB bit to determine hardware behavior after an ERAT or TLB address translation parity or multihit error is detected. When CCR2.NOTLB=1 (ERAT-only mode), this bit is effectively ignored and the instruction that caused the parity or multihit error is flushed and a machine check exception occurs. When CCR2.NOTLB=0 (TLB mode), this bit determines behavior as follows:</p> <p>0 The instruction generating the parity or multihit is flushed, no machine check exception is generated, and the structure detecting the error is flash invalidated for all entries (in the case of an ERAT detection), or all entries in the congruence class (in the case of a TLB detection). This invalidation ignores entry protection state.</p> <p>1 The instruction generating the parity or multihit is flushed, and a machine check exception is generated (no invalidation occurs to TLB or ERAT entries).</p>
47	MDDMH	0b0	<p><u>Machine Check on Data Cache Directory Multihit</u></p> <p>0 Data cache directory multihit errors are recovered by hardware</p> <p>1 Data cache directory multihit errors are recovered by software (machine check interrupt)</p>
48:55	///	0x0	<u>Reserved</u>
56:57	TCD	0b00	<p><u>Timer Clock Divide</u></p> <p>This field selects how the timer update pulse is divided:</p> <p>00 DIV1: Timer clock is not divided</p> <p>01 DIV4: Timer clock is divided by 4</p> <p>10 DIV8: Timer clock is divided by 8</p> <p>11 DIV16: Timer clock is divided by 16</p>
58:63	///	0x0	<u>Reserved</u>

14.5.146 XUDBG0 - Execution Unit Debug Register 0

Register Short Name:	XUDBG0	Read Access:	Hypv
Decimal SPR Number:	885	Write Access:	Hypv
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:48	///	0x0	<u>Reserved</u>
49:51	WAY	0b000	<u>Data Cache Directory Way Select</u> Selects way for a data cache directory read
52:57	ROW	0x0	<u>Data Cache Directory Row Select</u> Selects row for a data cache directory read
58:61	///	0b0000	<u>Reserved</u>
62	EXEC ^{NP}	0b0	<u>Data Cache Directory Read Execute</u> ^{NP} 1 Executes a data cache directory read
63	DONE	0b0	<u>Data Cache Directory Read Done</u> 1 Indicates a data cache directory read operation has completed and XUDBG1/XUDBG2 registers are valid

14.5.147 XUDBG1 - Execution Unit Debug Register 1

Register Short Name:	XUDBG1	Read Access:	Hypv
Decimal SPR Number:	886	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32:44	///	0x0	<u>Reserved</u>
45:48	WATCH	0b0000	<u>Data Cache Directory Watch Bits</u> 0 Directory entry has no watch set 1 Directory entry has watch set
49:55	LRU	0x0	<u>Data Cache Directory LRU</u> Indicates value of the LRU in the data cache directory
56:59	PARITY	0b0000	<u>Data Cache Directory Parity</u> Indicates value of the parity bits in the data cache directory
60:61	///	0b00	<u>Reserved</u>
62	LOCK	0b0	<u>Data Cache Directory Lock Bits</u> 0 Directory entry is unlocked 1 Directory entry is locked
63	VALID	0b0	<u>Data Cache Directory Read Valid</u> 0 directory entry is not valid 1 directory entry is valid

14.5.148 XUDBG2 - Execution Unit Debug Register 2

Register Short Name:	XUDBG2	Read Access:	Hypv
Decimal SPR Number:	887	Write Access:	None
Initial Value:	0x0000000000000000	Duplicated for MT:	N
Slow SPR:	Y	Notes:	
Guest Supervisor Mapping:		Scan Ring:	func

Bit(s):	Field Name:	Init	Description
32	///	0b0	<u>Reserved</u>
33:63	TAG	0x0	<u>Data Cache Directory Tag</u> Indicates value of the tag bit in the data cache directory

15. SCOM Accessible Registers

The serial communication (SCOM) interface provides access to registers used for pervasive operations. A SCOM satellite within the PC unit provides the external connections and address decode needed for accessing these registers. All SCOM accessible registers reside within the PC unit. Access to other core registers through the SCOM interface is enabled by the Ram Instruction, Ram Command, and Ram Data Registers. These registers are used for debug access to core facilities and to enable “instruction stuffing” into a stopped thread’s pipeline.

15.1 Serial Communications (SCOM) Description

The SCOM interface is the primary method for pervasive access to A2 registers in the chip. This section provides a brief introduction to SCOM as it relates to register access within the A2 core. An overview of the SCOM components and connections is shown in Figure 15-1 on page 704.

Register accesses initiated by master devices (JTAG, POR engine, alter/display unit) go through the pervasive control bus (PCB) to the chiplet-level SCOM controller. Upper bits of the SCOM address determine the destination ring number that the device uses for routing the packet and which SCOM satellite is selected. The SCOM satellite completes the address decode and performs the read or write operation.

The external interface for the SCOM satellite is comprised of a 2-wire serial connection for data and control; in addition, 4 bits are used for programming the satellite address. The serial interface is further described by direction of data flow: DL (downlink from the controller to the SCOM satellite) and UL (uplink from the SCOM satellite back to the controller). The following signals make up the SCOM interface to the A2 core:

DL-CCH (1 bit)	Downlink control channel. Controls clock and power-gating and satellite reset.
UL-CCH (1 bit)	Uplink control channel to the next SCOM satellite in the chain or the controller.
DL-DCH (1 bit)	Downlink data channel. Carries the address and data packets.
UL-DCH (1 bit)	Uplink data channel to the next SCOM satellite in the chain or the controller.
Satellite ID (4 bits)	Core inputs tied high or low that set the satellite ID on its SCOM ring. Compared against the SCOM address to select a device.

Figure 15-2 on page 704 shows the basic use of the SCOM interface signals during SCOM read and write operations. The address information sent by the PCB to the controller consists of 13 bits and is broken up into the following fields:

Ring Number (3 bits)	The controller uses this field to select the ring number for forwarding the packet. Valid ring numbers for connecting to a SCOM satellite are rings 1 through 7 (0b001 through 0b111). The ring number field is stripped off the serial address bits forwarded to the SCOM satellites on the DCH signal.
Satellite Number (4 bits)	The SCOM satellite compares these bits against its own satellite ID and, on a match, responds to the packet by decoding the remaining address bits.
Register Number (6 bits)	Allows access of up to 64 registers from a single SCOM satellite.

Figure 3. Chip Level Infrastructure Example to Access SCOM Registers in the A2 Core

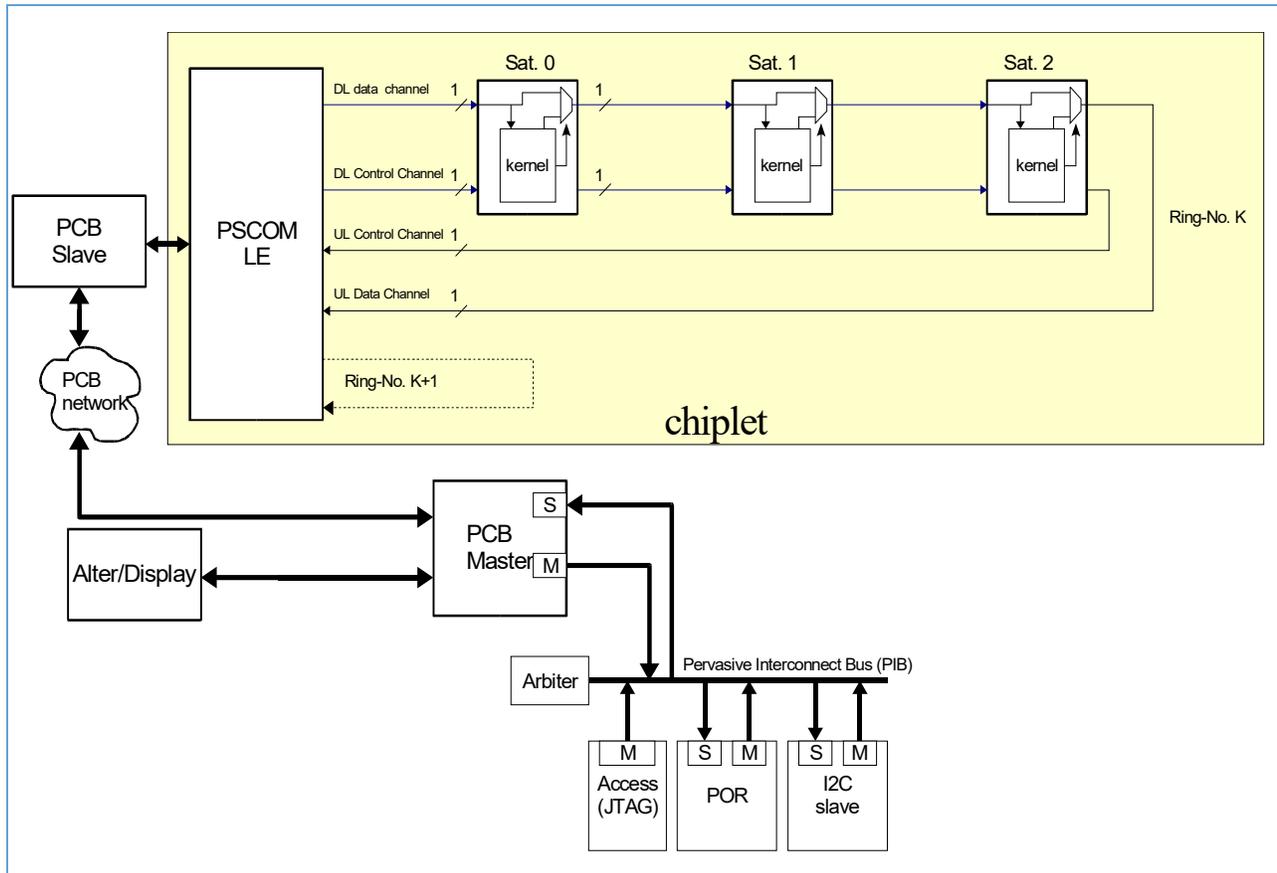
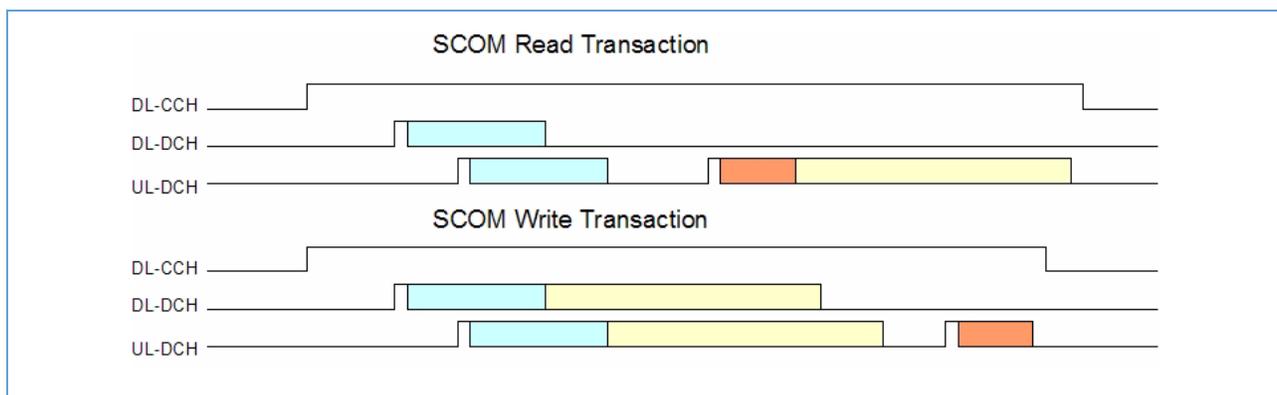


Figure 4. Principle Timing of Information Carried on CCH and DCH



15.2 SCOM Register Summary

15.2.1 SCOM register access methods

SCOM registers can have multiple access methods besides basic read and write. Certain registers have SCOM addresses that provide a *Reset with AND mask* or *Set with OR mask* capability. This section describes the access methods used by PC unit SCOM registers as well as the terminology used within the register tables.

15.2.1.1 Register access terminology

- **RW** - SCOM address provides both read and write access
- **WO** - Write-only address
- **RO** - Read-only address
- **WOAND** - Write-only by AND mask
- **WOOR** - Write-only by OR mask

The SCOM register tables differentiate between two types of reserved bits. Reserved bits that are **unimplemented** do not have an associated latch at the SCOM register location, and will always return a 0 value. Reserved bits that are **spare** access an existing, but unused, latch. Spare bit locations can be set or cleared by SCOM writes, and if set will return a 1 upon a read access.

15.2.1.2 Reset with AND mask (WOAND)

The register bit is reset when the corresponding bit written to the AND mask is a '0'. Refer to the example below:

```
100101100 <= Initial state of register
111010100 <= SCOM write to AND mask
100000100 <= New state of register
```

15.2.1.3 Set with OR mask (WOOR)

The mask is ORed bit-by-bit with the corresponding register. A bit is set when the original register bit was set, or when the corresponding bit written to the OR mask is a '1'. Refer to the example below:

```
000100110 <= Initial state of register
101100000 <= SCOM write to OR mask
101100110 <= New state of register
```

15.2.2 SCOM register summary table

Table 15-1 lists the SCOM accessible registers in order of ascending register address. Each read or write operation transfers 64 bits of data over the SCOM interface. Not all of the SCOM accessible registers implement the full 64 bits; with the unused bits being designated as reserved. Reserved bits will be read as 0, and should be written as 0. Writing 1 to a reserved bit location could result in parity errors.

Table 1. PC Unit SCOM Register Summary

Reg Address *1		Register Name	Description	Size	Access	Scan Ring
Hex	Dec					
x'00'	0	FIR0	Fault Isolation Register 0	32	RW	bcfg
x'01'	1	FIR0	Fault Isolation Register 0	32	WOAND	bcfg
x'02'	2	FIR0	Fault Isolation Register 0	32	WOOR	bcfg
x'03'	3	FIR0A0	FIR0 Action0 Register	32	RW	bcfg
x'04'	4	FIR0A1	FIR0 Action1 Register	32	RW	bcfg
x'05'	5		Reserved			
x'06'	6	FIR0M	FIR0 Mask Register	32	RW	bcfg
x'07'	7	FIR0M	FIR0 Mask Register	32	WOAND	bcfg
x'08'	8	FIR0M	FIR0 Mask Register	32	WOOR	bcfg
x'09'	9	ERRINJ	Error Injection Register	32	RW	func
x'0A'	10	FIR1	Fault Isolation Register 1	32	RW	bcfg
x'0B'	11	FIR1	Fault Isolation Register 1	32	WOAND	bcfg
x'0C'	12	FIR1	Fault Isolation Register 1	32	WOOR	bcfg
x'0D'	13	FIR1A0	FIR1 Action0 Register	32	RW	bcfg
x'0E'	14	FIR1A1	FIR1 Action1 Register	32	RW	bcfg
x'0F'	15		Reserved			
x'10'	16	FIR1M	FIR1 Mask Register	32	RW	bcfg
x'11'	17	FIR1M	FIR1 Mask Register	32	WOAND	bcfg
x'12'	18	FIR1M	FIR1 Mask Register	32	WOOR	bcfg
x'13'	19	FIR01RD	FIR0 and FIR1 Register Summary	64	RO	bcfg
x'14'	20	FIR2	Fault Isolation Register 2	32	RW	bcfg
x'15'	21	FIR2	Fault Isolation Register 2	32	WOAND	bcfg
x'16'	22	FIR2	Fault Isolation Register 2	32	WOOR	bcfg
x'17'	23	FIR2A0	FIR2 Action0 Register	32	RW	bcfg
x'18'	24	FIR2A1	FIR2 Action1 Register	32	RW	bcfg
x'19'	25		Reserved			
x'1A'	26	FIR2M	FIR2 Mask Register	32	RW	bcfg
x'1B'	27	FIR2M	FIR2 Mask Register	32	WOAND	bcfg
x'1C'	28	FIR2M	FIR2 Mask Register	32	WOOR	bcfg
x'1D' through x'27' 29 through 39			Reserved			
x'28'	40	RAMIC	Ram Instruction and Command Registers	64	RW	func
x'29'	41	RAMI	Ram Instruction Register	32	RW	func
x'2A'	42	RAMC	Ram Command Register	32	RW	func
x'2B'	43	RAMC	Ram Command Register	32	WOAND	func

Table 1. PC Unit SCOM Register Summary

Reg Address *1		Register	Description	Size	Access	Scan Ring
Hex	Dec	Name				
x'2C'	44	RAMC	Ram Command Register	32	WOOR	func
x'2D'	45	RAMD	Ram Data Register	64	RW	func
x'2E'	46	RAMDH	Ram Data Register High	32	RW	func
x'2F'	47	RAMDL	Ram Data Register Low	32	RW	func
x'30'	48	THRCTL	Thread Control and Status Register	32	RW	bcfg
x'31'	49	THRCTL	Thread Control and Status Register	32	WOAND	bcfg
x'32'	50	THRCTL	Thread Control and Status Register	32	WOOR	bcfg
x'33'	51	PCCR0	PC Configuration Register 0	32	RW	dcfg
x'34'	52	PCCR0	PC Configuration Register 0	32	WOAND	dcfg
x'35'	53	PCCR0	PC Configuration Register 0	32	WOOR	dcfg
x'36'	54	SPATTN	Special Attention Register	32	RW	bcfg
x'37'	55	SPATTN	Special Attention Register	32	WOAND	bcfg
x'38'	56	SPATTN	Special Attention Register	32	WOOR	bcfg
x'39' through x'3A' 57 through 58			Reserved			
x'3B'	59	ARDSR	AXU/RV Debug Select Register	32	RW	dcfg
x'3C'	60	IDSR	IU Debug Select Register	32	RW	dcfg
x'3D'	61	MPDSR	MMU/PC Debug Select Register	32	RW	dcfg
x'3E'	62	XDSR	XU Debug Select Register	32	RW	dcfg
x'3F'	63	LDSR	LQ Debug Select Register	32	RW	dcfg

Note:

1) The register address columns show the final decode value for the 6 bit register address controlled by the SCOM Satellite. The full SCOM address is "Ring Number || Satellite Number || Register Address". The values applied for ring number and satellite number are implementation dependent, and beyond the scope of this document.

15.2.3 SCOM register descriptions (alphabetical order)

15.2.3.1 AXU/RV Debug Select Register (ARDSR)

Table 2. AXU and RV Debug Select Register

Short Name	ARDSR		Access	RW
Register Address	x'3B' RW		Scan Ring Initial Value	dcfg 0x0000000000000000
Bit Num	Function	Init	Description	

Table 2. AXU and RV Debug Select Register			
0:31	Reserved (Unimplemented)	0	
AXU Debug Mux1 Controls (4:1 Debug Mux)			
32:33	Debug Group Mux Select	0	Selects which debug group is driven to the debug mux output: 00 - Debug Group 0 01 - Debug Group 1 10 - Debug Group 2 11 - Debug Group 3
34:36	Reserved (Spare)	0	
37:38	Debug Group Rotate Select	0	Selects how the 16-bit rotate function shifts the debug mux output data: 00 - Debug Group data (0:63) - No Rotate 01 - Debug Group data (48:63 & 0:47) 10 - Debug Group data (32:63 & 0:31) 11 - Debug Group data (16:63 & 0:15)
39	Debug Group Output Select (0:15)	0	Determines which signal group is put on Trace Data Out (0:15) 0 - Trace Data In (0:15) is routed onto the trace bus 1 - Debug group rotate output (0:15) is placed onto the trace bus
40	Debug Group Output Select (16:31)	0	Determines which signal group is put on Trace Data Out (16:31) 0 - Trace Data In (16:31) is routed onto the trace bus 1 - Debug group rotate output (16:31) is placed onto the trace bus
41	Debug Group Output Select (32:47)	0	Determines which signal group is put on Trace Data Out (32:47) 0 - Trace Data In (32:47) is routed onto the trace bus 1 - Debug group rotate output (32:47) is placed onto the trace bus
42	Debug Group Output Select (48:63)	0	Determines which signal group is put on Trace Data Out (48:63) 0 - Trace Data In (48:63) is routed onto the trace bus 1 - Debug group rotate output (48:63) is placed onto the trace bus
43:44	Trigger Group Mux Select	0	Selects which trigger group is driven to the mux output: 00 - Trigger Group 0 01 - Trigger Group 1 10 - Trigger Group 2 11 - Trigger Group 3
45	Trigger Group Rotate Select	0	Selects how the 6-bit rotate function shifts the trigger mux output data: 0 - Trigger Group data (0:11) - No Rotate 1 - Trigger Group data (6:11 & 0:5)
46	Trigger Group Output Select (0:5)	0	Determines which signal group is put on Trigger Data Out (0:5) 0 - Trigger Data In (0:5) is routed onto the trigger bus 1 - Trigger group rotate output (0:5) is placed onto the trigger bus
47	Trigger Group Output Select (6:11)	0	Determines which signal group is put on Trigger Data Out (6:11) 0 - Trigger Data In (6:11) is routed onto the trigger bus 1 - Trigger group rotate output (6:11) is placed onto the trigger bus
RV Debug Mux1 Controls (4:1 Debug Mux)			
48:49	Debug Group Mux Select	0	Selects which debug group is driven to the debug mux output: 00 - Debug Group 0 01 - Debug Group 1 10 - Debug Group 2 11 - Debug Group 3
50:52	Reserved (Spare)	0	

Table 2. AXU and RV Debug Select Register

53:54	Debug Group Rotate Select	0	Selects how the 16-bit rotate function shifts the debug mux output data: 00 - Debug Group data (0:63) - No Rotate 01 - Debug Group data (48:63 & 0:47) 10 - Debug Group data (32:63 & 0:31) 11 - Debug Group data (16:63 & 0:15)
55	Debug Group Output Select (0:15)	0	Determines which signal group is put on Trace Data Out (0:15) 0 - Trace Data In (0:15) is routed onto the trace bus 1 - Debug group rotate output (0:15) is placed onto the trace bus
56	Debug Group Output Select (16:31)	0	Determines which signal group is put on Trace Data Out (16:31) 0 - Trace Data In (16:31) is routed onto the trace bus 1 - Debug group rotate output (16:31) is placed onto the trace bus
57	Debug Group Output Select (32:47)	0	Determines which signal group is put on Trace Data Out (32:47) 0 - Trace Data In (32:47) is routed onto the trace bus 1 - Debug group rotate output (32:47) is placed onto the trace bus
58	Debug Group Output Select (48:63)	0	Determines which signal group is put on Trace Data Out (48:63) 0 - Trace Data In (48:63) is routed onto the trace bus 1 - Debug group rotate output (48:63) is placed onto the trace bus
59:60	Trigger Group Mux Select	0	Selects which trigger group is driven to the mux output: 00 - Trigger Group 0 01 - Trigger Group 1 10 - Trigger Group 2 11 - Trigger Group 3
61	Trigger Group Rotate Select	0	Selects how the 6-bit rotate function shifts the trigger mux output data: 0 - Trigger Group data (0:11) - No Rotate 1 - Trigger Group data (6:11 & 0:5)
62	Trigger Group Output Select (0:5)	0	Determines which signal group is put on Trigger Data Out (0:5) 0 - Trigger Data In (0:5) is routed onto the trigger bus 1 - Trigger group rotate output (0:5) is placed onto the trigger bus
63	Trigger Group Output Select (6:11)	0	Determines which signal group is put on Trigger Data Out (6:11) 0 - Trigger Data In (6:11) is routed onto the trigger bus 1 - Trigger group rotate output (6:11) is placed onto the trigger bus

15.2.3.2 Error Injection Register (ERRINJ)

Note: Although bits of the Error Injection register can be set at any time through SCOM writes, the error inject signals are gated by an *error inject enable* (PCCR0(34) = '1'). After activation, an error inject signal will remain active until the corresponding error bit in the FIR has been latched.

Table 3. Error Injection Register

Note: Bits 55 through 63 are specific to thread 1, and are unimplemented on single-threaded versions of A20.

Short Name	ERRINJ	Access	RW
Register Address	x'09' RW	Scan Ring Initial Value	func 0x00000000_00000000

Table 3. Error Injection Register

Note: Bits 55 through 63 are specific to thread 1, and are unimplemented on single-threaded versions of A20.

Bit Num	Function	Init	Description
0:31	Reserved (Unimplemented)	0	
32	I-Cache parity error	0	Causes I-Cache parity error.
33	I-Cache directory parity error	0	Causes I-Cache directory parity error.
34	I-Cache directory multihit error	0	Causes an I-Cache directory multihit error.
35	D-Cache parity error	0	Causes D-Cache parity error.
36	D-Cache directory ldp parity error	0	Causes D-Cache directory load pipeline parity error.
37	D-Cache directory stp parity error	0	Causes D-Cache directory store pipeline parity error.
38	D-Cache directory ldp multihit error	0	Causes a D-Cache directory load pipeline multihit error.
39	D-Cache directory stp multihit error	0	Causes a D-Cache directory store pipeline multihit error.
40	SCOM register parity error	0	Forces parity error on SCOM register write.
41	LQ prefetcher parity error	0	Causes a parity error in the LQ prefetcher array.
42	LQ reload queue parity error	0	Causes a parity error in the LQ relq array.
43:44	Reserved (Spare)	0	
45	SPRG array ECC error, T0	0	Causes ECC error in the SPRG array for thread 0.
46	FX0 register file parity error, T0	0	Causes parity error in the FX0 regfile array for thread 0.
47	FX1 register file parity error, T0	0	Causes parity error in the FX1 regfile array for thread 0.
48	LQ register file parity error, T0	0	Causes parity error in the LQ regfile array for thread 0.
49	FU register file parity error, T0	0	Causes parity error in the FU regfile array for thread 0.
50	Livelock buster attempted, T0	0	Livelock buster logic activated for thread 0.
51	Livelock buster failed, T0	0	Livelock buster logic active, and fails to free-up hang condition
52	IU completion array parity error, T0	0	Causes parity error in the CP array for thread 0.
53:54	Reserved (Spare)	0	
55	SPRG array ECC error, T1	0	Causes ECC error in the SPRG array for thread 1.
56	FX0 register file parity error, T1	0	Causes parity error in the FX0 regfile array for thread 1.
57	FX1 register file parity error, T1	0	Causes parity error in the FX1 regfile array for thread 1.
58	LQ register file parity error, T1	0	Causes parity error in the LQ regfile array for thread 1.
59	FU register file parity error, T1	0	Causes parity error in the FU regfile array for thread 1.
60	Livelock buster attempted, T1	0	Livelock buster logic activated for thread 1.
61	Livelock buster failed, T1	0	Livelock buster logic active, and fails to free-up hang condition
62	IU completion array parity error, T1	0	Causes parity error in the CP array for thread 1.
63	Reserved (Spare)	0	

15.2.3.3 Fault Isolation Registers and their associated Action and Mask registers

The A20 core implements 3 sets of Fault Isolation Registers. Each FIR group is comprised of the following registers (where x is 0, 1 or 2):

- Fault Isolation Register x (FIRx)
- FIRx Action 0 Register (FIRxA0)
- FIRx Action 1 Register (FIRxA1)
- FIRx Mask Register (FIRxM)

Bits in the Mask, Action0 and Action1 registers have a 1-to-1 correspondence to the FIR, and together determine how an unmasked error is reported. The table below describes the function of the Mask and Action bits:

Mask(n)	Action0(n)	Action1(n)	Resulting action for FIR(n)
1	X	X	Masked - error is latched in FIR, but does not get reported
X	0	0	Masked - error is latched in FIR, but does not get reported.
0	0	1	Recoverable - error latched and reported as recoverable
0	1	0	System Checkstop - error latched and reported as checkstop; new errors blocked from setting FIR.
0	1	1	Local Checkstop - error latched and reported as a local core checkstop.

Table 4. Fault Isolation Register 0

Short Name	FIR0	Access	RW, WOAND, WOOR
Register Address	x'00' RW x'01' WOAND x'02' WOOR	Scan Ring Initial Value	bcfg 0x00000000_00000000
Bit Num	Function	Init	Description
0:31	Reserved (Unimplemented)	0	
32	max_recov_err_cntr_value	0	The recoverable error counter has incremented to its max value of b'1111'. Additional unmasked recoverable errors will wrap the counter to 0, before it continues a new count.
33	iu_pc_err_icache_parity	0	An instruction cache parity error was detected. The failing cacheline is invalidated.
34	iu_pc_err_icachedir_parity	0	An instruction cache directory parity error was detected. The failing cacheline is invalidated.
35	iu_pc_err_icachedir_multihit	0	An I-Cache directory multihit error was detected. The failing cacheline is invalidated.
36	lq_pc_err_dcache_parity	0	A data cache parity error was detected. Refer to Note 1 for description of recovery actions.
37	lq_pc_err_dcachedir_ldp_parity	0	A data cache directory parity error was detected in the load pipeline. Refer to Note 2 for description of recovery actions.
38	lq_pc_err_dcachedir_stp_parity	0	A data cache directory parity error was detected in the store pipeline. The failing cacheline is invalidated.
39	lq_pc_err_dcachedir_ldp_multihit	0	A D-Cache directory multihit error was detected in the load pipeline. Refer to Note 3 for description of recovery actions.

Table 4. Fault Isolation Register 0

40	lq_pc_err_dcachedir_stp_multihit	0	A D-Cache directory multihit error was detected in the store pipeline. The failing cacheline is invalidated.
41	iu_pc_err_ierat_parity	0	An I-ERAT parity error was detected. Refer to Note 4 for description of recovery actions.
42	iu_pc_err_ierat_multihit	0	A multiple entry hit error was detected by the I-ERAT compare logic. Refer to Note 4 for description of recovery actions.
43	lq_pc_err_derat_parity	0	A D-ERAT parity error was detected. Refer to Note 4 for description of recovery actions.
44	lq_pc_err_derat_multihit	0	A multiple entry hit error was detected by the D-ERAT compare logic. Refer to Note 4 for description of recovery actions.
45	mm_pc_err_tlb_parity	0	A TLB parity error was detected. Refer to Note 4 for description of recovery actions.
46	mm_pc_err_tlb_multihit	0	A multiple entry hit error was detected by the TLB compare logic. Refer to Note 4 for description of recovery actions.
47	mm_pc_err_tlb_lru_parity	0	A TLB LRU parity error was detected. Refer to Note 4 for description of recovery actions.
48	mm_pc_err_local_snoop_reject	0	A local back-invalidate snoop was rejected due to a LPAR ID mismatch.
49	lq_pc_err_l2intra_ecc	0	An ECC error was detected on data sent to the core on the L2 interface. The L2 will resend the data.
50	lq_pc_err_l2intra_ue	0	An uncorrectable error was detected on data sent to the core on the L2 interface. This error should be reported as a checkstop.
51	lq_pc_err_invlid_reld	0	The Load-store unit received load data from the L2 when there were no outstanding load requests. This error should be reported as a checkstop.
52	lq_pc_err_l2credit_overrun	0	The store queue or load queue received more credits from the L2 interface than is allowable. This error should be reported as a checkstop.
53	scom_reg_parity_err	0	A parity error was detected in either PCCR0 or the mask portion of the SPATTN register. The specific register that caused the error can be determined by scanning out error reporting macro data from the PC unit bcfg scan rings.
54	scom_reg_ack_err	0	An invalid SCOM register access occurred either through an invalid address, or by invalid read/write request to a valid address. This bit can also be caused by an error in the SCOM satellite.
55	fir_regs_parity_err	0	Parity error detected in one of the FIR related registers (Action0, Action1 or Mask registers). The specific register that caused the error can be determined by scanning out error reporting macro data from the PC unit bcfg scan rings. This error should be reported as a checkstop.
56	lq_pc_err_prefetcher_parity	0	The prefetcher array detected a parity error on its output data. The failing array entry will be overwritten when the invalid address is replaced.
57	lq_pc_err_relq_parity	0	A parity error was detected in the reload queue. The corresponding cache line will not be validated. If the instruction is an ldawx the watchlost bit is set. For dcbtls and dbctstls instructions, XUCR0[CSLC] is set.
58:59	Reserved (Spare)	0	
60:63	Reserved (Unimplemented)	0	

Table 4. Fault Isolation Register 0

Notes:

- *1 - When set, XUCR0[MDCP] enables reporting of D-Cache parity errors as machine checks for software recovery. The default setting enables hardware recovery.
- *2 - When set, XUCR0[MDDP] enables reporting of D-Cache directory parity errors as machine checks for software recovery. The default setting enables hardware recovery.
- *3 - When set, XUCR4[MDDMH] enables reporting of D-Cache directory multihit errors as machine checks for software recovery. The default setting enables hardware recovery.
- *4 - With these control bit settings, XUCR4[MMU_MCHK]=1 and CCR2[NOTLB]=0, or whenever CCR2[NOTLB]=1, ERAT/TLB parity and multihit errors will be reported as machine checks for software recovery. Hardware recovery is attempted when CCR2[NOTLB]=0 and XUCR4[MMU_MCHK]=0. Refer to the XUCR4[MMU_MCHK] bit description for details.

Table 5. FIR0 Action and Mask Registers

	Fir0 Action Register 0	Fir0 Action Register 0	Fir0 Mask Register
Short Name	FIR0A0	FIR0A1	FIR0M
Address and Access Modes	x'03' RW	x'04' RW	x'06' RW x'07' WOAND x'08' WOOR
Initial Value	0x00000000_00003900	0x00000000_FFFFFFFC0	0x00000000_FFFFFFFC0
Scan Ring	bcfg	bcfg	bcfg

Bit Num	Function	Act 0 Init	Act 1 Init	Mask Init	Description
0:31	Reserved (Unimplemented)	0	0	0	
32	max_recov_err_cntr_value	0	1	1	Recoverable error counter max value - recoverable.
33	iu_pc_err_icache_parity	0	1	1	I-Cache recoverable parity error.
34	iu_pc_err_icachedir_parity	0	1	1	I-Cache directory recoverable parity error.
35	iu_pc_err_icachedir_multihit	0	1	1	I-Cache directory recoverable multihit error.
36	lq_pc_err_dcache_parity	0	1	1	D-Cache recoverable parity error.
37	lq_pc_err_dcachedir_ldp_parity	0	1	1	D-Cache directory load pipeline recoverable parity error.
38	lq_pc_err_dcachedir_stp_parity	0	1	1	D-Cache directory store pipeline recoverable parity error.
39	lq_pc_err_dcachedir_ldp_multihit	0	1	1	D-Cache directory load pipeline recoverable multihit error.
40	lq_pc_err_dcachedir_stp_multihit	0	1	1	D-Cache directory store pipeline recoverable multihit error.
41	iu_pc_err_ierat_parity	0	1	1	I-ERAT recoverable parity error.
42	iu_pc_err_ierat_multihit	0	1	1	I-ERAT recoverable multihit error.
43	lq_pc_err_derat_parity	0	1	1	D-ERAT recoverable parity error.
44	lq_pc_err_derat_multihit	0	1	1	D-ERAT recoverable multihit error.

45	mm_pc_err_tlb_parity	0	1	1	TLB recoverable parity error.
46	mm_pc_err_tlb_multihit	0	1	1	TLB recoverable multihit error.
47	mm_pc_err_tlb_lru_parity	0	1	1	TLB LRU recoverable parity error.
48	mm_pc_err_local_snoop_reject	0	1	1	Local back-invalidate snoop rejected. Should be set to recoverable.
49	lq_pc_err_l2intrf_ecc	0	1	1	L2 interface recoverable error.
50	lq_pc_err_l2intrf_ue	1	1	1	L2 interface checkstop error.
51	lq_pc_err_invld_reld	1	1	1	Load-store unit checkstop error.
52	lq_pc_err_l2credit_overrun	1	1	1	Store or load queue credit overrun checkstop error.
53	scom_reg_parity_err	0	1	1	SCOM register recoverable parity error.
54	scom_reg_ack_err	0	1	1	SCOM access recoverable error.
55	fir_regs_parity_err	1	1	1	FIR related register (Action0, Action1 or Mask) checkstop error.
56	lq_pc_err_prefetcher_parity	0	1	1	Prefetcher array recoverable parity error.
57	lq_pc_err_relq_parity	0	1	1	Reload queue recoverable parity error.
57:59	Reserved (Spare)	0	0	0	
60:63	Reserved (Unimplemented)	0	0	0	

Table 6. FIR0 and FIR1 Registers (Read Only)

Short Name	FIR01RD	Access	RO
Register Address	x'13' RO	Scan Ring Initial Value	bcfg 0x0000000000000000
Bit Num	Function	Init	Description
0:31	FIR0(32 to 63)	0	Provides single read operation of both FIR0 and FIR1 in implementations supporting 64 bit access. Refer to the FIR0 and FIR1 registers for individual bit descriptions.
32:63	FIR1(32 to 63)	0	

Table 7. Fault Isolation Register 1

Short Name	FIR1	Access	RW, WOAND, WOOR
Register Address	x'0A' RW x'0B' WOAND x'0C' WOOR	Scan Ring Initial Value	bcfg 0x00000000_00000000
Bit Num	Function	Init	Description

Table 7. Fault Isolation Register 1

0:31	Reserved (Unimplemented)	0	
32	xu_pc_err_sprg_ecc, T0	0	ECC correctible error detected on data read out of the SPRG array by thread 0. Hardware error recovery will correct the data.
33	xu_pc_err_sprg_ue, T0	0	An uncorrectable error was detected on data read out of the SPRG array by thread 0. This error should be reported as a checkstop.
34	xu_pc_err_regfile_parity, T0	0	A FX0 or FX1 register file parity error was detected by thread 0. Hardware error recovery will correct the data and update the array.
35	xu_pc_err_regfile_ue, T0	0	An uncorrectable error was detected on data read out of either the FX0 or FX1 register file by thread 0. This error should be reported as a checkstop.
36	lq_pc_err_regfile_parity, T0	0	A LQ register file parity error was detected by thread 0. Hardware error recovery will correct the data and update the array.
37	lq_pc_err_regfile_ue, T0	0	An uncorrectable error was detected on data read out of the LQ register file by thread 0. This error should be reported as a checkstop.
38	fu_pc_err_regfile_parity, T0	0	A FU register file parity error was detected by thread 0. Hardware error recovery will correct the data and update the array.
39	fu_pc_err_regfile_ue, T0	0	An uncorrectable error was detected on data read out of the FU register file by thread 0. This error should be reported as a checkstop.
40	iu_pc_err_cpArray_parity, T0	0	A parity error was detected on data read out of the completion array for thread 0. Recovery method is TBD.
41	iu_pc_err_ucode_illegal, T0	0	The microcode engine detected an illegal instruction on thread 0. Default action is to cause a checkstop.
42	iu_pc_err_mchk_disabled, T0	0	A machine check interrupt occurred on thread 0 while machine checks were not enabled. This error should be reported as a checkstop.
43	xu_pc_err_llbust_attempt, T0	0	The XU livelock buster logic has detected a hang condition for thread 0. The thread priority will be increased.
44	xu_pc_err_llbust_failed, T0	0	The XU livelock buster's attempt to fix a thread 0 hang was not successful within the selected delay threshold period. Forward progress on another hung thread will be attempted before returning to this one.
45	xu_pc_err_wdt_reset, T0	0	A watchdog timer reset was requested by thread 0.
46	iu_pc_err_debug_event, T0	0	A debug compare event on thread 0 occurred and was enabled to set a bit in the FIR. Default action is to cause a checkstop.
47:51	Reserved (Spare)	0	
52:63	Reserved (Unimplemented)	0	

Table 8. FIR1 Action and Mask Registers

	Fir1 Action Register 0	Fir1 Action Register 0	Fir1 Mask Register
Short Name	FIR1A0	FIR1A1	FIR1M
Address and Access Modes	x'0D' RW	x'0E' RW	x'10' RW x'11' WOAND x'12' WOOR

Initial Value	0x00000000_55660000	0x00000000_FFFE0000	0x00000000_FFFE0000
Scan Ring	bcfg	bcfg	bcfg

Bit Num	Function	Act 0 Init	Act 1 Init	Mask Init	Description
0:31	Reserved (Unimplemented)	0	0	0	
32	xu_pc_err_sprg_ecc, T0	0	1	1	SPRG array recoverable error on thread 0.
33	xu_pc_err_sprg_ue, T0	1	1	1	SPRG array checkstop error on thread 0.
34	xu_pc_err_regfile_parity, T0	0	1	1	FX0 or FX1 register file recoverable error on thread 0.
35	xu_pc_err_regfile_ue, T0	1	1	1	FX0 or FX1 register file checkstop error on thread 0.
36	lq_pc_err_regfile_parity, T0	0	1	1	LQ register file recoverable error on thread 0.
37	lq_pc_err_regfile_ue, T0	1	1	1	LQ register file checkstop error on thread 0.
38	fu_pc_err_regfile_parity, T0	0	1	1	FU register file recoverable error on thread 0.
39	fu_pc_err_regfile_ue, T0	1	1	1	FU register file checkstop error on thread 0.
40	iu_pc_err_cpArray_parity, T0	0	1	1	Completion array recoverable error on thread 0.
41	iu_pc_err_ucode_illegal, T0	1	1	1	Illegal microcoded instruction checkstop error on thread 0.
42	iu_pc_err_mchk_disabled, T0	1	1	1	A machine check interrupt occurred on thread 0 while machine checks were not enabled. This error should be reported as a checkstop.
43	xu_pc_err_llbust_attempt, T0	0	1	1	XU livelock buster logic recoverable error on thread 0.
44	xu_pc_err_llbust_failed, T0	0	1	1	XU livelock buster logic recoverable error on thread 0.
45	xu_pc_err_wdt_reset, T0	1	1	1	Watchdog timer reset requested on thread 0 - checkstop.
46	iu_pc_err_debug_event, T0	1	1	1	Debug compare event on thread 0.
47:51	Reserved (Spare)	0	0	0	
52:63	Reserved (Unimplemented)	0	0	0	

Table 9. Fault Isolation Register 2

Note: All bits in this register are specific to thread 1, and are unimplemented on single-threaded versions of A2o.

Short Name	FIR2	Access	RW, WOAND, WOOR
Register Address	x'14' RW x'15' WOAND x'16' WOOR	Scan Ring Initial Value	bcfg 0x00000000_00000000
Bit Num	Function	Init	Description
0:31	Reserved (Unimplemented)	0	
32	xu_pc_err_sprg_ecc, T1	0	ECC correctable error detected on data read out of the SPRG array by thread 1. Hardware error recovery will correct the data.

Table 9. Fault Isolation Register 2

Note: All bits in this register are specific to thread 1, and are unimplemented on single-threaded versions of A2o.

33	xu_pc_err_sprg_ue, T1	0	An uncorrectable error was detected on data read out of the SPRG array by thread 1. This error should be reported as a checkstop.
34	xu_pc_err_regfile_parity, T1	0	A FX0 or FX1 register file parity error was detected by thread 1. Hardware error recovery will correct the data and update the array.
35	xu_pc_err_regfile_ue, T1	0	An uncorrectable error was detected on data read out of the FX0 or FX1 register file by thread 1. This error should be reported as a checkstop.
36	lq_pc_err_regfile_parity, T1	0	A LQ register file parity error was detected by thread 1. Hardware error recovery will correct the data and update the array.
37	lq_pc_err_regfile_ue, T1	0	An uncorrectable error was detected on data read out of the LQ register file by thread 1. This error should be reported as a checkstop.
38	fu_pc_err_regfile_parity, T1	0	A FU register file parity error was detected by thread 1. Hardware error recovery will correct the data and update the array.
39	fu_pc_err_regfile_ue, T1	0	An uncorrectable error was detected on data read out of the FU register file by thread 1. This error should be reported as a checkstop.
40	iu_pc_err_cpArray_parity, T1	0	A parity error was detected on data read out of the completion array for thread 1. Recovery method is TBD.
41	iu_pc_err_ucode_illegal, T1	0	The microcode engine detected an illegal instruction on thread 1. Default action is to cause a checkstop.
42	iu_pc_err_mchk_disabled, T1	0	A machine check interrupt occurred on thread 1 while machine checks were not enabled. This error should be reported as a checkstop.
43	xu_pc_err_llbust_attempt, T1	0	The XU livelock buster logic has detected a hang condition for thread 1. The thread priority will be increased.
44	xu_pc_err_llbust_failed, T1	0	The XU livelock buster's attempt to fix a thread 1 hang was not successful within the selected delay threshold period. Forward progress on another hung thread will be attempted before returning to this one.
45	xu_pc_err_wdt_reset, T1	0	A watchdog timer reset was requested by thread 1.
46	iu_pc_err_debug_event, T1	0	A debug compare event on thread 1 occurred and was enabled to set a bit in the FIR. Default action is to cause a checkstop.
47:51	Reserved (Spare)	0	
52:63	Reserved (Unimplemented)	0	

Table 10. FIR2 Action and Mask Registers

Note: All bits in this register are specific to thread 1, and are unimplemented on single-threaded versions of A2o.

	Fir2 Action Register 0	Fir2 Action Register 0	Fir2 Mask Register
Short Name	FIR2A0	FIR2A1	FIR2M
Address and Access Modes	x'17' RW	x'18' RW	x'1A' RW x'1B' WOAND x'1C' WOOR
Initial Value	0x00000000_55660000	0x00000000_FFFE0000	0x00000000_FFFE0000

Bit Num	Function	Act 0 Init	Act 1 Init	Mask Init	Description
0:31	Reserved (Unimplemented)	0	0	0	
32	xu_pc_err_sprg_ecc, T1	0	1	1	SPRG array recoverable error on thread 1.
33	xu_pc_err_sprg_ue, T1	1	1	1	SPRG array checkstop error on thread 1.
34	xu_pc_err_regfile_parity, T1	0	1	1	FX0 or FX1 register file recoverable error on thread 1.
35	xu_pc_err_regfile_ue, T1	1	1	1	FX0 or FX1 register file checkstop error on thread 1.
36	lq_pc_err_regfile_parity, T1	0	1	1	LQ register file recoverable error on thread 1.
37	lq_pc_err_regfile_ue, T1	1	1	1	LQ register file checkstop error on thread 1.
38	fu_pc_err_regfile_parity, T1	0	1	1	FU register file recoverable error on thread 1.
39	fu_pc_err_regfile_ue, T1	1	1	1	FU register file checkstop error on thread 1.
40	iu_pc_err_cpArray_parity, T1	0	1	1	Completion array recoverable error on thread 1.
41	iu_pc_err_ucose_illegal, T1	1	1	1	Illegal microcoded instruction checkstop error on thread 1.
42	xu_pc_err_mchk_disabled, T1	1	1	1	A machine check interrupt occurred on thread 1 while machine checks were not enabled. This error should be reported as a checkstop.
43	xu_pc_err_llbust_attempt, T1	0	1	1	XU livelock buster logic recoverable error on thread 1.
44	xu_pc_err_llbust_failed, T1	0	1	1	XU livelock buster logic recoverable error on thread 1.
45	xu_pc_err_wdt_reset, T1	1	1	1	Watchdog timer reset requested on thread 1 - checkstop.
46	iu_pc_err_debug_event, T1	1	1	1	Debug compare event on thread 1.
47:51	Reserved (Spare)	0	0	0	
52:63	Reserved (Unimplemented)	0	0	0	

15.2.3.4 IU Debug Select Register (IDSR)

Table 11. IU Debug Select Register				
Short Name	IDSR	Access	RW	
Register Address	x'3C' RW	Scan Ring Initial Value	dcfg 0x0000000000000000	
Bit Num	Function	Init	Description	
0:31	Reserved (Unimplemented)	0		
IU Debug Mux1 Controls (16:1 Debug Mux)				

Table 11. IU Debug Select Register

32:35	Debug Group Mux Select	0	Selects which debug group is driven to the debug mux output: 0000 - Debug Group 0 0001 - Debug Group 1 0010 - Debug Group 2 1111 - Debug Group 15
36	Reserved (Spare)	0	
37:38	Debug Group Rotate Select	0	Selects how the 16-bit rotate function shifts the debug mux output data: 00 - Debug Group data (0:63) - No Rotate 01 - Debug Group data (48:63 & 0:47) 10 - Debug Group data (32:63 & 0:31) 11 - Debug Group data (16:63 & 0:15)
39	Debug Group Output Select (0:15)	0	Determines which signal group is put on Trace Data Out (0:15) 0 - Trace Data In (0:15) is routed onto the trace bus 1 - Debug group rotate output (0:15) is placed onto the trace bus
40	Debug Group Output Select (16:31)	0	Determines which signal group is put on Trace Data Out (16:31) 0 - Trace Data In (16:31) is routed onto the trace bus 1 - Debug group rotate output (16:31) is placed onto the trace bus
41	Debug Group Output Select (32:47)	0	Determines which signal group is put on Trace Data Out (32:47) 0 - Trace Data In (32:47) is routed onto the trace bus 1 - Debug group rotate output (32:47) is placed onto the trace bus
42	Debug Group Output Select (48:63)	0	Determines which signal group is put on Trace Data Out (48:63) 0 - Trace Data In (48:63) is routed onto the trace bus 1 - Debug group rotate output (48:63) is placed onto the trace bus
43:44	Trigger Group Mux Select	0	Selects which trigger group is driven to the mux output: 00 - Trigger Group 0 01 - Trigger Group 1 10 - Trigger Group 2 11 - Trigger Group 3
45	Trigger Group Rotate Select	0	Selects how the 6-bit rotate function shifts the trigger mux output data: 0 - Trigger Group data (0:11) - No Rotate 1 - Trigger Group data (6:11 & 0:5)
46	Trigger Group Output Select (0:5)	0	Determines which signal group is put on Trigger Data Out (0:5) 0 - Trigger Data In (0:5) is routed onto the trigger bus 1 - Trigger group rotate output (0:5) is placed onto the trigger bus
47	Trigger Group Output Select (6:11)	0	Determines which signal group is put on Trigger Data Out (6:11) 0 - Trigger Data In (6:11) is routed onto the trigger bus 1 - Trigger group rotate output (6:11) is placed onto the trigger bus
IU Debug Mux2 Controls (16:1 Debug Mux)			
48:51	Debug Group Mux Select	0	Selects which debug group is driven to the debug mux output: 0000 - Debug Group 0 0001 - Debug Group 1 0010 - Debug Group 2 1111 - Debug Group 15
52	Reserved (Spare)	0	

Table 11. IU Debug Select Register

53:54	Debug Group Rotate Select	0	Selects how the 16-bit rotate function shifts the debug mux output data: 00 - Debug Group data (0:63) - No Rotate 01 - Debug Group data (48:63 & 0:47) 10 - Debug Group data (32:63 & 0:31) 11 - Debug Group data (16:63 & 0:15)
55	Debug Group Output Select (0:15)	0	Determines which signal group is put on Trace Data Out (0:15) 0 - Trace Data In (0:15) is routed onto the trace bus 1 - Debug group rotate output (0:15) is placed onto the trace bus
56	Debug Group Output Select (16:31)	0	Determines which signal group is put on Trace Data Out (16:31) 0 - Trace Data In (16:31) is routed onto the trace bus 1 - Debug group rotate output (16:31) is placed onto the trace bus
57	Debug Group Output Select (32:47)	0	Determines which signal group is put on Trace Data Out (32:47) 0 - Trace Data In (32:47) is routed onto the trace bus 1 - Debug group rotate output (32:47) is placed onto the trace bus
58	Debug Group Output Select (48:63)	0	Determines which signal group is put on Trace Data Out (48:63) 0 - Trace Data In (48:63) is routed onto the trace bus 1 - Debug group rotate output (48:63) is placed onto the trace bus
59:60	Trigger Group Mux Select	0	Selects which trigger group is driven to the mux output: 00 - Trigger Group 0 01 - Trigger Group 1 10 - Trigger Group 2 11 - Trigger Group 3
61	Trigger Group Rotate Select	0	Selects how the 6-bit rotate function shifts the trigger mux output data: 0 - Trigger Group data (0:11) - No Rotate 1 - Trigger Group data (6:11 & 0:5)
62	Trigger Group Output Select (0:5)	0	Determines which signal group is put on Trigger Data Out (0:5) 0 - Trigger Data In (0:5) is routed onto the trigger bus 1 - Trigger group rotate output (0:5) is placed onto the trigger bus
63	Trigger Group Output Select (6:11)	0	Determines which signal group is put on Trigger Data Out (6:11) 0 - Trigger Data In (6:11) is routed onto the trigger bus 1 - Trigger group rotate output (6:11) is placed onto the trigger bus

15.2.3.5 LQ Debug Select Register (LDSR)**Table 12. LQ Debug Select Register**

Short Name	LDSR	Access	RW
Register Address	x'3F' RW	Scan Ring Initial Value	dcfg 0x0000000000000000
Bit Num	Function	Init	Description
0:31	Reserved (Unimplemented)	0	
LQ Debug Mux1 Controls (32:1 Debug Mux)			

Table 12. LQ Debug Select Register

32:36	Debug Group Mux Select	0	Selects which debug group is driven to the debug mux output: 00000 - Debug Group 0 00001 - Debug Group 1 00010 - Debug Group 2 11111 - Debug Group 31
37:38	Debug Group Rotate Select	0	Selects how the 16-bit rotate function shifts the debug mux output data: 00 - Debug Group data (0:63) - No Rotate 01 - Debug Group data (48:63 & 0:47) 10 - Debug Group data (32:63 & 0:31) 11 - Debug Group data (16:63 & 0:15)
39	Debug Group Output Select (0:15)	0	Determines which signal group is put on Trace Data Out (0:15) 0 - Trace Data In (0:15) is routed onto the trace bus 1 - Debug group rotate output (0:15) is placed onto the trace bus
40	Debug Group Output Select (16:31)	0	Determines which signal group is put on Trace Data Out (16:31) 0 - Trace Data In (16:31) is routed onto the trace bus 1 - Debug group rotate output (16:31) is placed onto the trace bus
41	Debug Group Output Select (32:47)	0	Determines which signal group is put on Trace Data Out (32:47) 0 - Trace Data In (32:47) is routed onto the trace bus 1 - Debug group rotate output (32:47) is placed onto the trace bus
42	Debug Group Output Select (48:63)	0	Determines which signal group is put on Trace Data Out (48:63) 0 - Trace Data In (48:63) is routed onto the trace bus 1 - Debug group rotate output (48:63) is placed onto the trace bus
43:44	Trigger Group Mux Select	0	Selects which trigger group is driven to the mux output: 00 - Trigger Group 0 01 - Trigger Group 1 10 - Trigger Group 2 11 - Trigger Group 3
45	Trigger Group Rotate Select	0	Selects how the 6-bit rotate function shifts the trigger mux output data: 0 - Trigger Group data (0:11) - No Rotate 1 - Trigger Group data (6:11 & 0:5)
46	Trigger Group Output Select (0:5)	0	Determines which signal group is put on Trigger Data Out (0:5) 0 - Trigger Data In (0:5) is routed onto the trigger bus 1 - Trigger group rotate output (0:5) is placed onto the trigger bus
47	Trigger Group Output Select (6:11)	0	Determines which signal group is put on Trigger Data Out (6:11) 0 - Trigger Data In (6:11) is routed onto the trigger bus 1 - Trigger group rotate output (6:11) is placed onto the trigger bus
LQ Debug Mux2 Controls (32:1 Debug Mux)			
48:52	Debug Group Mux Select	0	Selects which debug group is driven to the debug mux output: 00000 - Debug Group 0 00001 - Debug Group 1 00010 - Debug Group 2 11111 - Debug Group 31

Table 12. LQ Debug Select Register

53:54	Debug Group Rotate Select	0	Selects how the 16-bit rotate function shifts the debug mux output data: 00 - Debug Group data (0:63) - No Rotate 01 - Debug Group data (48:63 & 0:47) 10 - Debug Group data (32:63 & 0:31) 11 - Debug Group data (16:63 & 0:15)
55	Debug Group Output Select (0:15)	0	Determines which signal group is put on Trace Data Out (0:15) 0 - Trace Data In (0:15) is routed onto the trace bus 1 - Debug group rotate output (0:15) is placed onto the trace bus
56	Debug Group Output Select (16:31)	0	Determines which signal group is put on Trace Data Out (16:31) 0 - Trace Data In (16:31) is routed onto the trace bus 1 - Debug group rotate output (16:31) is placed onto the trace bus
57	Debug Group Output Select (32:47)	0	Determines which signal group is put on Trace Data Out (32:47) 0 - Trace Data In (32:47) is routed onto the trace bus 1 - Debug group rotate output (32:47) is placed onto the trace bus
58	Debug Group Output Select (48:63)	0	Determines which signal group is put on Trace Data Out (48:63) 0 - Trace Data In (48:63) is routed onto the trace bus 1 - Debug group rotate output (48:63) is placed onto the trace bus
59:60	Trigger Group Mux Select	0	Selects which trigger group is driven to the mux output: 00 - Trigger Group 0 01 - Trigger Group 1 10 - Trigger Group 2 11 - Trigger Group 3
61	Trigger Group Rotate Select	0	Selects how the 6-bit rotate function shifts the trigger mux output data: 0 - Trigger Group data (0:11) - No Rotate 1 - Trigger Group data (6:11 & 0:5)
62	Trigger Group Output Select (0:5)	0	Determines which signal group is put on Trigger Data Out (0:5) 0 - Trigger Data In (0:5) is routed onto the trigger bus 1 - Trigger group rotate output (0:5) is placed onto the trigger bus
63	Trigger Group Output Select (6:11)	0	Determines which signal group is put on Trigger Data Out (6:11) 0 - Trigger Data In (6:11) is routed onto the trigger bus 1 - Trigger group rotate output (6:11) is placed onto the trigger bus

15.2.3.6 MMU/PC Debug Select Register (MPDSR)**Table 13. MMU and PC Debug Select Register**

Short Name	MPDSR	Access	RW
Register Address	x'3D' RW	Scan Ring Initial Value	dcfg 0x0000000000000000
Bit Num	Function	Init	Description
0:31	Reserved (Unimplemented)	0	
MMU Debug Mux1 Controls (16:1 Debug Mux)			

Table 13. MMU and PC Debug Select Register

32:35	Debug Group Mux Select	0	Selects which debug group is driven to the debug mux output: 0000 - Debug Group 0 0001 - Debug Group 1 0010 - Debug Group 2 1111 - Debug Group 15
36	Reserved (Spare)	0	
37:38	Debug Group Rotate Select	0	Selects how the 16-bit rotate function shifts the debug mux output data: 00 - Debug Group data (0:63) - No Rotate 01 - Debug Group data (48:63 & 0:47) 10 - Debug Group data (32:63 & 0:31) 11 - Debug Group data (16:63 & 0:15)
39	Debug Group Output Select (0:15)	0	Determines which signal group is put on Trace Data Out (0:15) 0 - Trace Data In (0:15) is routed onto the trace bus 1 - Debug group rotate output (0:15) is placed onto the trace bus
40	Debug Group Output Select (16:31)	0	Determines which signal group is put on Trace Data Out (16:31) 0 - Trace Data In (16:31) is routed onto the trace bus 1 - Debug group rotate output (16:31) is placed onto the trace bus
41	Debug Group Output Select (32:47)	0	Determines which signal group is put on Trace Data Out (32:47) 0 - Trace Data In (32:47) is routed onto the trace bus 1 - Debug group rotate output (32:47) is placed onto the trace bus
42	Debug Group Output Select (48:63)	0	Determines which signal group is put on Trace Data Out (48:63) 0 - Trace Data In (48:63) is routed onto the trace bus 1 - Debug group rotate output (48:63) is placed onto the trace bus
43:44	Trigger Group Mux Select	0	Selects which trigger group is driven to the mux output: 00 - Trigger Group 0 01 - Trigger Group 1 10 - Trigger Group 2 11 - Trigger Group 3
45	Trigger Group Rotate Select	0	Selects how the 6-bit rotate function shifts the trigger mux output data: 0 - Trigger Group data (0:11) - No Rotate 1 - Trigger Group data (6:11 & 0:5)
46	Trigger Group Output Select (0:5)	0	Determines which signal group is put on Trigger Data Out (0:5) 0 - Trigger Data In (0:5) is routed onto the trigger bus 1 - Trigger group rotate output (0:5) is placed onto the trigger bus
47	Trigger Group Output Select (6:11)	0	Determines which signal group is put on Trigger Data Out (6:11) 0 - Trigger Data In (6:11) is routed onto the trigger bus 1 - Trigger group rotate output (6:11) is placed onto the trigger bus
PC Debug Mux1 Controls (4:1 Debug Mux)			
48:49	Debug Group Mux Select	0	Selects which debug group is driven to the debug mux output: 00 - Debug Group 0 01 - Debug Group 1 10 - Debug Group 2 11 - Debug Group 3
50:52	Reserved (Spare)	0	

Table 13. MMU and PC Debug Select Register

53:54	Debug Group Rotate Select	0	Selects how the 16-bit rotate function shifts the debug mux output data: 00 - Debug Group data (0:63) - No Rotate 01 - Debug Group data (48:63 & 0:47) 10 - Debug Group data (32:63 & 0:31) 11 - Debug Group data (16:63 & 0:15)
55	Debug Group Output Select (0:15)	0	Determines which signal group is put on Trace Data Out (0:15) 0 - Trace Data In (0:15) is routed onto the trace bus 1 - Debug group rotate output (0:15) is placed onto the trace bus
56	Debug Group Output Select (16:31)	0	Determines which signal group is put on Trace Data Out (16:31) 0 - Trace Data In (16:31) is routed onto the trace bus 1 - Debug group rotate output (16:31) is placed onto the trace bus
57	Debug Group Output Select (32:47)	0	Determines which signal group is put on Trace Data Out (32:47) 0 - Trace Data In (32:47) is routed onto the trace bus 1 - Debug group rotate output (32:47) is placed onto the trace bus
58	Debug Group Output Select (48:63)	0	Determines which signal group is put on Trace Data Out (48:63) 0 - Trace Data In (48:63) is routed onto the trace bus 1 - Debug group rotate output (48:63) is placed onto the trace bus
59:60	Trigger Group Mux Select	0	Selects which trigger group is driven to the mux output: 00 - Trigger Group 0 01 - Trigger Group 1 10 - Trigger Group 2 11 - Trigger Group 3
61	Trigger Group Rotate Select	0	Selects how the 6-bit rotate function shifts the trigger mux output data: 0 - Trigger Group data (0:11) - No Rotate 1 - Trigger Group data (6:11 & 0:5)
62	Trigger Group Output Select (0:5)	0	Determines which signal group is put on Trigger Data Out (0:5) 0 - Trigger Data In (0:5) is routed onto the trigger bus 1 - Trigger group rotate output (0:5) is placed onto the trigger bus
63	Trigger Group Output Select (6:11)	0	Determines which signal group is put on Trigger Data Out (6:11) 0 - Trigger Data In (6:11) is routed onto the trigger bus 1 - Trigger group rotate output (6:11) is placed onto the trigger bus

15.2.3.7 PC Configuration Register 0 (PCCR0)**Table 14. PC Configuration Register 0**

Note: Bits 57 through 59 are specific to thread 1, and are unimplemented on single-threaded versions of A2o.

Short Name	PCCR0	Access	RW, WOAND, WOOR
Register Address	x'33' RW x'34' WOAND x'35' WOOR	Scan Ring Initial Value	dcfg 0x0000000000000000
Bit Num	Function	Init	Description
0:31	Reserved (Unimplemented)	0	

Table 14. PC Configuration Register 0

Note: Bits 57 through 59 are specific to thread 1, and are unimplemented on single-threaded versions of A2o.

32	EnabDebugMode	0	<p>This bit places the core in debug mode.</p> <p>It is used to enable debug logic such as the trace and trigger mux controls and buses.</p> <p>Enabling debug mode allows various debug functions to be performed (i.e. instruction stepping and miscellaneous debug controls such as THRCTL[DisabAsynclrpts, DisabTimebaselrpts, DisabDeclrpts]).</p>
33	EnabRamOperations	0	<p>This bit enables Ram mode operation through the RAMI, RAMC and RAMD registers.</p> <p>It is gated with various RAMC control bits, such as: RamMode, RamExecute, EnabMsrOverrides and FlushThread.</p>
34	EnabErrorInjection	0	<p>This bit enables control signals set in the ERRINJ register to force errors in order to test error recovery methods.</p>
35	EnabExtDebugStop	0	<p>When set, this bit enables the input signal <i>an_ac_debug_stop</i> to stop all threads.</p>
36	DisabRamXstopReport	0	<p>Setting this bit will block the reporting of checkstop errors outside of the core when in Ram mode. A checkstop error would still be indicated by the RAMC_{Checkstop} bit, and the core's local FIR.</p>
37	EnabFastClockstop	0	<p>This bit enables a checkstop error to directly force all core tholds active, thereby quickly stopping clocks.</p> <p>Note: The core must be in debug mode (PC Configuration Register 0, bit 32 = 1) for this bit to be valid.</p>
38	DisabPowerSavings	0	<p>This bit blocks power-savings controls from raising the run tholds, and thereby forcing off the associated latch clocks.</p> <p>Other power-savings control signals (i.e. <i>ac_an_rvwinkle_mode</i>) will still be active, but all core latch clocks will remain enabled.</p>
39	DisabOverrunChks	0	<p>Setting this bit stops overrun checking logic from blocking Ram or instruction step operations when an overrun condition is detected.</p> <p>Also, overrun conditions will not be indicated through the appropriate status bits (RAMC[Overrun] and THRCTL[InstrStepOverrun]).</p>
40:43	Reserved (Spare)	0	
44:47	Reserved (Unimplemented)	0	
48:51	RecovErrorCounter	0	<p>This 4 bit counter increments whenever an unmasked recoverable error occurs. When the count value reaches 15, an error bit will be set in FIR0.</p> <p>The count value can be read to obtain the current value, or written to preset or clear it.</p> <p>Note: Write access to the Recoverable Error Counter is only supported through the "RW" SCOM address.</p>
52	Reserved (Unimplemented)	0	<p>Additional actions that can be selected when a debug compare event occurs for the indicated thread (sets DBCR0[EDM] status bit).</p>
53:55	T0_DBA	000	<p>Debug Action Select:</p> <p>000 - No action</p> <p>001 - Reserved (no action)</p> <p>010 - Stop Specified Thread</p>
56	Reserved (Unimplemented)	0	<p>011 - Stop All Threads</p>
57:59	T1_DBA	000	<p>100 - Activate the thread's debug event error (Sets FIR bit)</p> <p>101 - Activate External Signal (<i>ac_an_debug_trigger</i> pulse)</p> <p>110 - Activate External Signal and Stop Specified Thread</p> <p>111 - Activate External Signal and Stop All Threads</p>
60:63	Reserved (Unimplemented)	0	

15.2.3.8 Ram Data Registers (RAMD, RAMDH, RAMDL)

Table 15. Ram Data Register

Short Name	RAMD		Access	RW
Register Address	x'2D' RW		Scan Ring Initial Value	func 0x0000000000000000
Bit Num	Function	Init	Description	
0:63	Ram Data(0 to 63)	0	<p>When in Ram Mode, the results of any instruction operation are written to the Ram Data registers.</p> <p>Provides read/write control over the Ram Data registers in implementations supporting 64 bit access.</p> <p>The Ram Data registers are updated upon activation of RAMC_{Done}.</p>	

Table 16. Ram Data Register High

Short Name	RAMDH		Access	RW
Register Address	x'2E' RW		Scan Ring Initial Value	func 0x0000000000000000
Bit Num	Function	Init	Description	
0:31	Reserved (Unimplemented)	0		
32:63	Ram Data(0 to 31)	0	<p>When in Ram Mode, the results of any instruction operation are written to the Ram Data registers.</p> <p>The Ram Data registers are updated upon activation of RAMC_{Done}.</p>	

Table 17. Ram Data Register Low

Short Name	RAMDL		Access	RW
Register Address	x'2F' RW		Scan Ring Initial Value	func 0x0000000000000000
Bit Num	Function	Init	Description	
0:31	Reserved (Unimplemented)	0		
32:63	Ram Data(32 to 63)	0	<p>When in Ram Mode, the results of any instruction operation are written to the Ram Data registers.</p> <p>The Ram Data registers are updated upon activation of RAMC_{Done}.</p>	

15.2.3.9 Ram Instruction and Command Registers (RAMC, RAMI, RAMIC)

Note: Although bits of the Ram Command register can be set at any time through SCOM writes, the Ram mode function and control signals are only active when Ram Mode Enable (PCCR0(33) = '1') has been set.

Table 18. Ram Command Register				
Short Name	RAMC		Access	RW, WOAND, WOOR
Register Address	x'2A' RW x'2B' WOAND x'2C' WOOR		Scan Ring Initial Value	func 0x0000000000000000
Bit Num	Function	Init	Description	
0:31	Reserved (Unimplemented)	0		
32	Extend_InstrTgt1	0	Provides highest order bit of Tgt1 field when using scratch registers (r32 through r35) as the instruction target.	
33	Extend_InstrSrc1	0	Provides highest order bit of Src1 field when using scratch registers (r32 through r35) as an instruction source.	
34	Extend_InstrSrc2	0	Provides highest order bit of Src2 field when using scratch registers (r32 through r35) as an instruction source.	
35	Extend_InstrSrc3	0	Provides highest order bit of Src3 field when using scratch registers (r32 through r35) as an instruction source.	
36:43	Reserved (Unimplemented)	0		
44	RamMode	0	Sets Ram Mode for the selected thread. Note: Ram operations must be enabled (PC Configuration Register 0, bit 33 = 1) for this bit to be valid.	
45	Reserved (Unimplemented)	0		
46	RamThread	0	Encoded thread selects for Ram operation. 0 - Thread 0 1 - Thread 1	
47	RamExecute	0	When set, the Ram Instruction will be forced into the processor pipeline for the selected thread. This bit is non-persistent; it is pulsed for one cycle and reset. Note: Ram operations must be enabled (PC Configuration Register 0, bit 33 = 1) with RamMode active, for the Ram Execute signal to be valid.	
48	EnabMsrOverrides	0	This bit enables the override of certain MSR bits for the Rammed thread. This capability allows access to SPRs for debug where normal program permissions would restrict that access. It can also be used to force debug interrupts active or inactive. Note: Ram operations must be enabled (PC Configuration Register 0, bit 33 = 1) with RamMode active, for the MSR Override Enable signal to be valid.	
49	Override_MSR[PR]	0	Along with EnabMsrOverrides, determines the Problem State for the thread. It replaces the MSR output, but does not alter the actual register bit. Note: Ram operations must be enabled (PC Configuration Register 0, bit 33 = 1) with RamMode active, for the MSR[PR] Override signal to be valid.	

Table 18. Ram Command Register

50	Override_MSR[GS]	0	<p>Along with EnabMsrOverrides, determines the Guest State for the thread. It replaces the MSR output, but does not alter the actual register bit.</p> <p>Note: Ram operations must be enabled (PC Configuration Register 0, bit 33 = 1) with RamMode active, for the MSR[GS] Override signal to be valid.</p>
51	Override_MSR[DE]	0	<p>Along with EnabMsrOverrides, determines if Debug Interrupts are enabled for the thread. It replaces the MSR output, but does not alter the actual register bit.</p> <p>Note: Ram operations must be enabled (PC Configuration Register 0, bit 33 = 1) with RamMode active, for the MSR[DE] Override signal to be valid.</p>
52	Reserved (Spare)	0	
53:54	FlushThread	0	<p>Reserved for Engineering Use Only</p> <p>When activated, completion logic will flush the current instruction.</p> <p>53 - flushes thread 0 54 - flushes thread 1</p> <p>These bits are non-persistent; they are pulsed for one cycle and reset.</p> <p>Note: Ram operations must be enabled (PC Configuration Register 0, bit 33 = 1) for the Flush Thread signal to be valid.</p>
55:58	Reserved (Spare)	0	
59	Unsupported	0	<p>Status bit indicating that an invalid Ram instruction was issued. Operations not supported for Ramming are:</p> <ul style="list-style-type: none"> Any instruction defined to be implemented via microcode. Any instruction requiring microcode intervention in order to complete. This includes unaligned load or store instructions which are processed through microcode as individual byte accesses. <p>The unsupported Rammed instructions are treated as no-ops.</p> <p>Activation of this bit also sets THRCTL[RamErrorStatus], which provides cumulative results for multiple Ram operations.</p>
60	Overrun	0	<p>Status bit indicating that the Rammed instruction resulted in an overrun condition. An overrun is indicated when any of the following occurs:</p> <ul style="list-style-type: none"> Executing a Ram instruction while the thread is still running (THRCTL[Tx_RUN] is active). Executing a second Ram instruction before RAMC[Done] has gone active for the first Ram instruction. Ram Mode ends while a Rammed instruction is in process (RAMC[Done] had not yet gone active). <p>As a result of the detected overrun, the Rammed instruction will be blocked (no execution attempted).</p> <p>Activation of this bit also sets THRCTL[RamErrorStatus], which provides cumulative results for multiple Ram operations.</p>
61	Interrupt	0	<p>Status bit indicating that the Rammed instruction resulted in an enabled exception. Interrogation of interrupt facilities (SRR0, SRR1, etc) may be required in order to determine the cause of the exception.</p> <p>Activation of this bit also sets THRCTL[RamErrorStatus], which provides cumulative results for multiple Ram operations.</p>

Table 18. Ram Command Register

62	Checkstop	0	Status bit indicating that while Ram mode was active, one of the FIR registers contained an enabled checkstop error. Examination of the FIR registers is required to determine the exact cause of the checkstop; which may or may not be related to a Rammed operation. While in Ram mode, the reporting of checkstops outside the core (to the chiplet FIR) can be blocked by setting PC Configuration Register 0, bit 36 = 1. Activation of this bit also sets THRCTL[RamErrorStatus], which provides cumulative results for a series of Ram operations.
63	Done	0	Status bit indicating that the previously executed Ram Instruction has completed This bit is cleared when RAMC[RamExecute] is activated.

Table 19. Ram Instruction Register

Short Name	RAMI	Access	RW
Register Address	x'29' RW	Scan Ring Initial Value	func 0x0000000000000000
Bit Num	Function	Init	Description
0:31	Reserved (Unimplemented)	0	
32:63	Ram Instruction	0	Instruction to be executed through a Ram operation. Refer to RAMC register for related Ram control bits.

Table 20. Ram Instruction and Command Registers

Short Name	RAMIC	Access	RW
Register Address	x'28' RW	Scan Ring Initial Value	func 0x0000000000000000
Bit Num	Function	Init	Description
0:31	RAMI(32 to 63)	0	Provides read/write control over the Ram Instruction and Command registers in implementations supporting 64 bit access.
32:63	RAMC(32 to 63)	0	Refer to the RAMI and RAMC registers for individual bit descriptions.

15.2.3.10 Special Attention Register (SPATTN)

The Special Attention Register (SPATTN) is a 32 bit SCOM accessible register used to control reporting of special attentions outside of the core. Functionally, the SPATTN register is divided into two halves: special attention sources, and their corresponding mask bits. SPATTN(32:47) provide information on which special attention sources are active. SPATTN(48:63) contain the corresponding mask bit for each special attention source.

The mask bits are initialized to 1, which blocks reporting of all special attention sources. When a mask bit is cleared, the corresponding special attention source bit is enabled to report a special attention outside of the core when active. A special attention will be reported either through an actual special attention condition, or through a SCOM write that sets the source bit. The A2O core reports special attentions (per thread) through the `ac_an_special_attn(0:1)` output.

Table 21. Special Attention Register

Note: Bits 33 and 49 are specific to thread 1, and are unimplemented on single-threaded versions of A2o.

Short Name	SPATTN		Access	RW, WOAND, WOOR
Register Address	x'36' RW x'37' WOAND x'38' WOOR	Scan Ring Initial Value	bcfg 0x000000000000C000	
Bit Num	Function	Init	Description	
0:31	Reserved (Unimplemented)	0		
32	Attention instruction, T0	0	Execution of an attention (attn) instruction by a thread will set the corresponding SPATTN register bit.	
33	Attention instruction, T1	0	Note: CCR2[EN_ATTEN] must be set in order for the attention instruction to update the SPATTN register. With CCR2[EN_ATTEN] cleared, an attention is treated as an illegal instruction type program interrupt.	
34:47	Reserved (Unimplemented)	0		
48	Attention instruction mask, T0	1	When this bit is 1, reporting of special attentions through SPATTN(32) is masked off. When cleared, setting SPATTN(32) will activate <code>ac_an_special_attn(0)</code> to report a thread 0 special attention.	
49	Attention instruction mask, T1	1	When this bit is 1, reporting of special attentions through SPATTN(33) is masked off. When cleared, setting SPATTN(33) will activate <code>ac_an_special_attn(1)</code> to report a thread 1 special attention.	
50:63	Reserved (Unimplemented)	0		

15.2.3.11 Thread Control and Status Register (THRCTL)

Table 22. Thread Control and Status Register

Note: Bits 33, 37, 41, 48 through 51 are specific to thread 1, and are unimplemented on single-threaded versions of A2o.

Short Name	THRCTL		Access	RW, WOAND, WOOR
Register Address	x'30' RW x'31' WOAND x'32' WOOR	Scan Ring Initial Value	bcfg 0x0000000000000000	
Bit Num	Field Name	Init	Description	
0:31	Reserved (Unimplemented)	0		

Table 22. Thread Control and Status Register

Note: Bits 33, 37, 41, 48 through 51 are specific to thread 1, and are unimplemented on single-threaded versions of A2o.

32	T0_Stop	0	When set, this thread will stop instruction fetch and enter a stopped state. Instructions currently in the pipeline will continue to completion. When reset, program execution resumes at the next instruction address available before stopping.
33	T1_Stop	0	In addition to a SCOM write, these bits may be set by the following conditions: <ul style="list-style-type: none"> An enabled checkstop error Either execution of a dnh instruction, or a debug compare event (when PCCR0[Tx_DBA] bits are configured to stop the thread upon occurrence of the compare event). An attn instruction when configured by CCR2[EN_ATTN].
34:35	Reserved (Unimplemented)	0	
36	T0_Step	0	Writing a '1' to this location causes one instruction for this thread to be issued. This bit will be reset upon completion of the stepped instruction.
37	T1_Step	0	Note: Prior to activating Tx_Step, the corresponding thread should be stopped (Tx_Run=0) to avoid an overrun occurring. Instruction stepping can be performed when the thread is stopped due to activation of either THRCTL[Tx_Stop] or the <i>an_ac_debug_stop</i> input. Note: The core must be in debug mode (PC Configuration Register 0, bit 32 = 1) for the single-step signals to be valid.
38:39	Reserved (Unimplemented)	0	
40	T0_Run	0	Status bit indicating that the thread is in a running state when set.
41	T1_Run	0	When '0', the thread is stopped. This bit is read only. Writes will have no effect.
42	Reserved (Unimplemented)	0	
43	DebugStopInput	0	Status bit indicating that an external debug stop request is active. An external debug stop occurs when the <i>an_ac_debug_stop</i> input signal is active and enabled through the PCCR0[EnabExtDebugStop] bit. The external debug stop input affects all core threads. This bit is read only. Writes will have no effect.
44:47	T0_StopRequest	0	These bits summarize the T0 stop requests that have been activated. <ul style="list-style-type: none"> 44 - A power management related stop. This could be the result of a power-savings (wait) instruction, the <i>an_ac_pm_thread_stop</i> or <i>an_ac_pm_fetch_halt</i> inputs. This bit is read only. Writes will have no effect. 45 - An enabled checkstop error has been detected. 46 - A debug related stop; either through executing a dnh instruction, or through an external mode debug compare event. 47 - An attn instruction (enabled by CCR2[EN_ATTN]) occurred. Bits 45 through 47 are writeable; they are activated by the same stop controls which cause THRCTL[T0_Stop] to be set.

Table 22. Thread Control and Status Register

Note: Bits 33, 37, 41, 48 through 51 are specific to thread 1, and are unimplemented on single-threaded versions of A2o.

48:51	T1_StopRequest	0	<p>These bits summarize the T1 stop requests that have been activated.</p> <p>48 - A power management related stop. This could be the result of a power-savings (wait) instruction, the <i>an_ac_pm_thread_stop</i> or <i>an_ac_pm_fetch_halt</i> inputs. This bit is read only. Writes will have no effect.</p> <p>49 - An enabled checkstop error has been detected.</p> <p>50 - A debug related stop; either through executing a dnh instruction, or through an external mode debug compare event.</p> <p>51 - An attn instruction (enabled by CCR2[EN_ATTN]) occurred.</p> <p>Bits 49 through 51 are writeable; they are activated by the same stop controls which cause THRCTL[T1_Stop] to be set.</p>
52	DisabAsynclrpts	0	<p>This bit provides a global disable to any thread's asynchronous interrupts as long as the associated thread is stopped through pervasive (THRCTL[Tx_Stop], <i>an_ac_debug_stop</i> or <i>an_ac_pm_thread_stop</i>) controls. The asynchronous interrupts are re-enabled whenever the thread is put in a running state (this includes activation during a single-step pulse).</p> <p>Note: The core must be in debug mode (PC Configuration Register 0, bit 32 = 1) for this signal to be valid.</p>
53	DisabTimebase	0	<p>Setting this bit will block incrementing of the Timebase whenever all threads are stopped through pervasive (THRCTL[Tx_Stop], <i>an_ac_debug_stop</i> or <i>an_ac_pm_thread_stop</i>) controls. The Timebase count will continue whenever any thread is in a running state (this includes activation during a single-step pulse).</p> <p>Note: The core must be in debug mode (PC Configuration Register 0, bit 32 = 1) for this signal to be valid.</p>
54	DisabDecrem	0	<p>Setting this bit blocks the counting of any thread's decremter, as long as that thread is stopped through pervasive (THRCTL[Tx_Stop], <i>an_ac_debug_stop</i> or <i>an_ac_pm_thread_stop</i>) controls. Decrementer counting will be re-enabled whenever the thread is put in a running state (this includes activation during a single-step pulse).</p> <p>Note: The core must be in debug mode (PC Configuration Register 0, bit 32 = 1) for this signal to be valid.</p>
55:61	Reserved (Spare)	0	
62	InstrStepOverrun	0	<p>This bit is set when instruction stepping on any thread results in an overrun condition. The overrun occurs when THRCTL[Tx_Stop] is activated while the corresponding thread is still running (THRCTL[Tx_Run]=1).</p> <p>As a result of the detected overrun, the instruction-step operation will be blocked (no execution attempted) and THRCTL[Tx_Stop] will be cleared.</p>
63	RamErrorStatus	0	<p>This bit is activated whenever a Ram operation results in setting one of the RAMC status bits (RAMC[Unsupported, Overrun, Interrupt, Check-stop]).</p> <p>It provides cumulative status for multiple Ram operations when the RAMC status bits are not read between new Rammed instruction.</p>

15.2.3.12 XU Debug Select Register (XDSR)

Table 23. XU Debug Select Register

Short Name	XDSR	Access	RW

Table 23. XU Debug Select Register

Register Address	x'3E' RW	Scan Ring Initial Value	dcfg 0x0000000000000000
Bit Num	Function	Init	Description
0:31	Reserved (Unimplemented)	0	
XU Debug Mux1 Controls (4:1 Debug Mux)			
32:33	Debug Group Mux Select	0	Selects which debug group is driven to the debug mux output: 00 - Debug Group 0 01 - Debug Group 1 10 - Debug Group 2 11 - Debug Group 3
34:36	Reserved (Spare)	0	
37:38	Debug Group Rotate Select	0	Selects how the 16-bit rotate function shifts the debug mux output data: 00 - Debug Group data (0:63) - No Rotate 01 - Debug Group data (48:63 & 0:47) 10 - Debug Group data (32:63 & 0:31) 11 - Debug Group data (16:63 & 0:15)
39	Debug Group Output Select (0:15)	0	Determines which signal group is put on Trace Data Out (0:15) 0 - Trace Data In (0:15) is routed onto the trace bus 1 - Debug group rotate output (0:15) is placed onto the trace bus
40	Debug Group Output Select (16:31)	0	Determines which signal group is put on Trace Data Out (16:31) 0 - Trace Data In (16:31) is routed onto the trace bus 1 - Debug group rotate output (16:31) is placed onto the trace bus
41	Debug Group Output Select (32:47)	0	Determines which signal group is put on Trace Data Out (32:47) 0 - Trace Data In (32:47) is routed onto the trace bus 1 - Debug group rotate output (32:47) is placed onto the trace bus
42	Debug Group Output Select (48:63)	0	Determines which signal group is put on Trace Data Out (48:63) 0 - Trace Data In (48:63) is routed onto the trace bus 1 - Debug group rotate output (48:63) is placed onto the trace bus
43:44	Trigger Group Mux Select	0	Selects which trigger group is driven to the mux output: 00 - Trigger Group 0 01 - Trigger Group 1 10 - Trigger Group 2 11 - Trigger Group 3
45	Trigger Group Rotate Select	0	Selects how the 6-bit rotate function shifts the trigger mux output data: 0 - Trigger Group data (0:11) - No Rotate 1 - Trigger Group data (6:11 & 0:5)
46	Trigger Group Output Select (0:5)	0	Determines which signal group is put on Trigger Data Out (0:5) 0 - Trigger Data In (0:5) is routed onto the trigger bus 1 - Trigger group rotate output (0:5) is placed onto the trigger bus
47	Trigger Group Output Select (6:11)	0	Determines which signal group is put on Trigger Data Out (6:11) 0 - Trigger Data In (6:11) is routed onto the trigger bus 1 - Trigger group rotate output (6:11) is placed onto the trigger bus
Spare Debug Mux Controls (TBD:1 Debug Mux)			
48:63	Reserved (Spare)	0	



Appendix A. Processor Instruction Summary

This appendix lists all of the A2O Core instructions, summarized alphabetically by mnemonic. Extended mnemonics are not included in the opcode list. Reserved-nop opcodes are included.

A.1 Instruction Formats

Instructions are 4 bytes long. Instruction addresses are always word aligned.

Instruction bits 0 through 5 always contain the primary opcode. Many instructions have an extended opcode in another field. Remaining instruction bits contain additional fields. All instruction fields belong to one of the following categories:

- Defined

These instructions contain values, such as opcodes, that cannot be altered. The instruction format diagrams specify the values of defined fields.

- Variable



These fields contain operands, such as GPR selectors and immediate values, that can vary from execution to execution. The instruction format diagrams specify the operands in the variable fields.

- Reserved

Bits in reserved fields should be set to 0. In the instruction format diagrams, /, //, or /// denotes a reserved field, in a register, instruction, field, or bit string.

If any bit in a defined field does not contain the expected value, the instruction is illegal, and an illegal instruction exception occurs. If any bit in a reserved field does not contain 0, the instruction form is invalid; its result is architecturally undefined. The A2O Core executes all invalid instruction forms without causing an illegal instruction exception.

A.2 Implemented Instructions Sorted by Mnemonic

The Form column in *Table A-1* refers to the arrangement of valid field combinations within the 4-byte instruction. See the *Power ISA, V 2.06B* for a definition of the terms used in this column and the Category column.

In the Implemented column, “Y” indicates that the A2 core does implement this instruction. An “N” indicates that this instruction is not implemented.

In the Microcoded column, “Y” indicates that the A2 implementation is via microcode.

Table A-1. A2 Core Instructions by Mnemonic (Sheet 1 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
31	266	XO	add	B	Y		6:10	11:15	16:20		1	Add
31	266	XO	add.	B	Y		6:10	11:15	16:20		2	Add and Record
31	10	XO	addc	B	Y		6:10	11:15	16:20		2	Add with Carry
31	10	XO	addc.	B	Y		6:10	11:15	16:20		2	Add with Carry and Record
31	10	XO	addco	B	Y		6:10	11:15	16:20		2	Add with Carry and Overflow
31	10	XO	addco.	B	Y		6:10	11:15	16:20		2	Add with Carry and Overflow and Record
31	138	XO	adde	B	Y		6:10	11:15	16:20		2	Add Extended
31	138	XO	adde.	B	Y		6:10	11:15	16:20		2	Add Extended with Record
31	138	XO	addeo	B	Y		6:10	11:15	16:20		2	Add Extended with Overflow
31	138	XO	addeo.	B	Y		6:10	11:15	16:20		2	Add Extended with Overflow and Record
31	74	XO	addg6s	B	Y		6:10	11:15	16:20			Add and Generate Sixes
14		D	addi	B	Y		6:10	11:15			1	Add Immediate
12		D	addic	B	Y		6:10	11:15			2	Add Immediate and Carry
13		D	addic.	B	Y		6:10	11:15			2	Add Immediate with Carry and Record
15		D	addis	B	Y		6:10	11:15			1	Add Immediate Shifted
31	234	XO	addme	B	Y		6:10	11:15			2	Add to Minus One Extended
31	234	XO	addme.	B	Y		6:10	11:15			2	Add to Minus One Extended and Record
31	234	XO	addmeo	B	Y		6:10	11:15			2	Add to Minus One Extended with Overflow
31	234	XO	addmeo.	B	Y		6:10	11:15			2	Add to Minus One Extended with Overflow and Record
31	266	XO	addo	B	Y		6:10	11:15	16:20		2	Add with Overflow
31	266	XO	addo.	B	Y		6:10	11:15	16:20		2	Add with Overflow and Record
31	202	XO	addze	B	Y		6:10	11:15			2	Add to Zero Extended
31	202	XO	addze.	B	Y		6:10	11:15			2	Add to Zero Extended and Record
31	202	XO	addzeo	B	Y		6:10	11:15			2	Add to Zero Extended with Overflow
31	202	XO	addzeo.	B	Y		6:10	11:15			2	Add to Zero Extended with Overflow and Record
31	28	X	and	B	Y		11:15	6:10	16:20		2	And
31	28	X	and.	B	Y		11:15	6:10	16:20		2	And and Record
31	60	X	andc	B	Y		11:15	6:10	16:20		2	And with Complement
31	60	X	andc.	B	Y		11:15	6:10	16:20		2	And with Complement and Record
28		D	andi.	B	Y		11:15	6:10			2	And Immediate and Record

Table A-1. A2 Core Instructions by Mnemonic (Sheet 2 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
29		D	andis.	B	Y		11:15	6:10			2	And Immediate Shifted and Record
0	256	TAG	attn	SP	Y						6	Attention
18		I	b	B	Y						3	Branch
18		I	ba	B	Y						3	Branch Absolute
16		B	bc	B	Y						3	Branch Conditional
16		B	bca	B	Y						3	Branch Conditional Absolute
19	528	XL	bcctr	B	Y						3	Branch Conditional to Count
19	528	XL	bcctrl	B	Y						3	Branch Conditional to Count and Link
16		B	bcl	B	Y						3	Branch Conditional and Link
16		B	bcla	B	Y						3	Branch Conditional Absolute and Link
19	16	XL	bclr	B	Y						3	Branch Conditional to Link Register
19	16	XL	bclrl	B	Y						3	Branch Conditional to Link Register and Link
19	560	XL	bctar	B	Y						3	Branch Conditional to Branch Target Address Register
19	560	XL	bctarl	B	Y						3	Branch Conditional to Branch Target Address Register and Link
18		I	bl	B	Y						3	Branch and Link
18		I	bla	B	Y						3	Branch Absolute and Link
31	252	X	bpermd	64	Y	Y	11:15	6:10	16:20			Bit Permute Doubleword
4	527	EVX	brinc	SP	N		6:10	11:15	16:20			Bit Reversed Increment
31	314	X	cbcdtd	B	Y		11:15	6:10				Convert Binary Coded Decimal To Declets
31	282	X	cdtbcd	B	Y		11:15	6:10				Convert Declets To Binary Coded Decimal
31	0	X	cmp	B	Y			11:15	16:20		2	Compare
31	508	X	cmpb	B	Y		11:15	6:10	16:20		2	Compare Byte
11		D	cmpi	B	Y			11:15			2	Compare Immediate
31	32	X	cmpl	B	Y			11:15	16:20		2	Compare Logical
10		D	cmpli	B	Y			11:15			2	Compare Logical Immediate
31	58	X	cntlzd	64	Y	Y	11:15	6:10				Count Leading Zeros Doubleword
31	58	X	cntlzd.	64	Y	Y	11:15	6:10				Count Leading Zeros Doubleword and Record
31	26	X	cntlzw	B	Y	Y	11:15	6:10				Count Leading Zeros Word
31	26	X	cntlzw.	B	Y	Y	11:15	6:10				Count Leading Zeros Word and Record
19	257	XL	crand	B	Y						3	Condition Register And

Table A-1. A2 Core Instructions by Mnemonic (Sheet 3 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
19	129	XL	crandc	B	Y						3	Condition Register And with Complement
19	289	XL	creqv	B	Y						3	Condition Register Equal
19	225	XL	crnand	B	Y						3	Condition Register AND
19	33	XL	crnor	B	Y						3	Condition Register NOR
19	449	XL	cror	B	Y						3	Condition Register OR
19	417	XL	crorc	B	Y						3	Condition Register OR with Complement
19	193	XL	crxor	B	Y						3	Condition Register XOR
31	758	X	dcba	E	Y			11:15	16:20			Data Cache Block Allocate
31	86	X	dcbf	B	Y			11:15	16:20		3	Data Cache Block Flush
31	127	X	dcbfep	E.PD	Y			11:15	16:20		3	Data Cache Block Flush by External PID
31	470	X	dcbi	E	Y			11:15	16:20		3	Data Cache Block Invalidate
31	390	X	dcblc	E.CL	Y		7:10	11:15	16:20		3	Data Cache Block Lock Clear
31	54	X	dcbst	B	Y			11:15	16:20		3	Data Cache Block Store
31	63	X	dcbstep	E.PD	Y			11:15	16:20		3	Data Cache Block Store by External PID
31	278	X	dcbt	B	Y			11:15	16:20		3	Data Cache Block Touch
31	319	X	dcbtcp	E.PD	Y			11:15	16:20		3	Data Cache Block Touch by External PID
31	166	X	dcbtls	E.CL	Y			11:15	16:20		3	Data Cache Block Touch and Lock Set
31	246	X	dcbtst	B	Y			11:15	16:20		3	Data Cache Block Touch for Store
31	255	X	dcbtstep	E.PD	Y			11:15	16:20		3	Data Cache Block Touch for Store by External PID
31	134	X	dcbtstls	E.CL	Y			11:15	16:20		3	Data Cache Block Touch for Store and Lock Set
31	1014	X	dcbz	B	Y			11:15	16:20		3	Data Cache Block set to Zero
31	1023	X	dcbzep	E.PD	Y			11:15	16:20		3	Data Cache Block set to Zero by External PID
31	454	X	dci	E.CI	Y						3	Data Cache Invalidate
31	326	X	dcread	E.CD	N		6:10	11:15	16:20			Data Cache Read [Alternate Encoding]
31	486	X	dcread	E.CD	N		6:10	11:15	16:20			Data Cache Read
31	489	XO	divd	64	Y		6:10	11:15	16:20		36	Divide Doubleword
31	489	XO	divd.	64	Y		6:10	11:15	16:20		36	Divide Doubleword and Record
31	425	XO	divde	64	Y		6:10	11:15	16:20		68	Divide Doubleword Extended
31	425	XO	divde.	64	Y		6:10	11:15	16:20		68	Divide Doubleword Extended and Record

Table A-1. A2 Core Instructions by Mnemonic (Sheet 4 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
31	425	XO	divdeo	64	Y		6:10	11:15	16:20		68	Divide Doubleword Extended with Overflow
31	425	XO	divdeo.	64	Y		6:10	11:15	16:20		68	Divide Doubleword Extended with Overflow and Record
31	393	XO	divdeu	64	Y		6:10	11:15	16:20		68	Divide Doubleword Extended
31	393	XO	divdeu.	64	Y		6:10	11:15	16:20		68	Divide Doubleword Extended and Record
31	393	XO	divdeuo	64	Y		6:10	11:15	16:20		68	Divide Doubleword Extended with Overflow
31	393	XO	divdeuo.	64	Y		6:10	11:15	16:20		68	Divide Doubleword Extended with Overflow and Record
31	489	XO	divdo	64	Y		6:10	11:15	16:20		36	Divide Doubleword with Overflow
31	489	XO	divdo.	64	Y		6:10	11:15	16:20		36	Divide Doubleword with Overflow and Record
31	457	XO	divdu	64	Y		6:10	11:15	16:20		36	Divide Doubleword Unsigned
31	457	XO	divdu.	64	Y		6:10	11:15	16:20		36	Divide Doubleword Unsigned and Record
31	457	XO	divduo	64	Y		6:10	11:15	16:20		36	Divide Doubleword Unsigned with Overflow
31	457	XO	divduo.	64	Y		6:10	11:15	16:20		36	Divide Doubleword Unsigned with Overflow and Record
31	491	XO	divw	B	Y		6:10	11:15	16:20		20	Divide Word
31	491	XO	divw.	B	Y		6:10	11:15	16:20		20	Divide Word and Record
31	427	XO	divwe	B	Y		6:10	11:15	16:20		36	Divide Word Extended
31	427	XO	divwe.	B	Y		6:10	11:15	16:20		36	Divide Word Extended and Record
31	427	XO	divweo	B	Y		6:10	11:15	16:20		36	Divide Word Extended with Overflow
31	427	XO	divweo.	B	Y		6:10	11:15	16:20		36	Divide Word Extended with Overflow and Record
31	395	XO	divweu	B	Y		6:10	11:15	16:20		36	Divide Word Extended Unsigned
31	395	XO	divweu.	B	Y		6:10	11:15	16:20		36	Divide Word Extended Unsigned and Record
31	395	XO	divweuo	B	Y		6:10	11:15	16:20		36	Divide Word Extended Unsigned with Overflow
31	395	XO	divweuo.	B	Y		6:10	11:15	16:20		36	Divide Word Extended Unsigned with Overflow and Record
31	491	XO	divwo	B	Y		6:10	11:15	16:20		20	Divide Word with Overflow
31	491	XO	divwo.	B	Y		6:10	11:15	16:20		20	Divide Word with Overflow and Record
31	459	XO	divwu	B	Y		6:10	11:15	16:20		20	Divide Word Unsigned
31	459	XO	divwu.	B	Y		6:10	11:15	16:20		20	Divide Word Unsigned and Record
31	459	XO	divwuo	B	Y		6:10	11:15	16:20		20	Divide Word Unsigned with Overflow

Table A-1. A2 Core Instructions by Mnemonic (Sheet 5 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
31	459	XO	divwuo.	B	Y		6:10	11:15	16:20		20	Divide Word Unsigned with Overflow and Record
31	78	X	d1mzb	LMV	N		11:15	6:10	16:20			Determine Leftmost Zero Byte
31	78	X	d1mzb.	LMV	N		11:15	6:10	16:20			Determine Leftmost Zero Byte and Record
19	198	XF	dnh	E.ED	Y							Debugger Notify Halt
19	402	XL	doze	S	N							Doze
			dss		N							Data Stream Stop
			dst		N							Data Stream Touch
			dstst		N							Data Stream Touch for Store
31	310	X	eciwx	EC	N		6:10	11:15	16:20			External Control in Word Indexed
31	438	X	ecowx	EC	N		6:10	11:15	16:20			External Control out Word Indexed
31	270	XL	ehpriv	E.HV	Y						1	Generate Embedded Hypervisor Privilege Exception
31	854	X	eieio	S	N							Enforce In-order Execution of I/O
31	284	X	eqv	B	Y		11:15	6:10	16:20		2	Equivalent
31	284	X	eqv.	B	Y		11:15	6:10	16:20		2	Equivalent and Record
31	51	X	erat1lx	E.A2	Y			11:15	16:20		7	ERAT Invalidate Local Indexed
31	819	X	erat1vax	E.A2	Y			11:15	16:20	6:10	7	ERAT Invalidate Virtual Address Indexed
31	179	X	eratre	E.A2	Y		6:10		11:15	16:20	7	ERAT Read Entry
31	883	X	eratsrx.	E.A2	N			11:15	16:20		7	ERAT Search and Reserve Indexed and Record
31	147	X	eratsx	E.A2	Y		6:10	11:15	16:20		7	ERAT Search Indexed
31	147	X	eratsx.	E.A2	Y		6:10	11:15	16:20		7	ERAT Search Indexed and Record
31	211	X	eratwe	E.A2	Y			6:10	11:15	16:20	7	ERAT Write Entry
31	954	X	extsb	B	Y		11:15	6:10			2	Extend Sign Byte
31	954	X	extsb.	B	Y		11:15	6:10			2	Extend Sign Byte and Record
31	922	X	extsh	B	Y		11:15	6:10			2	Extend Sign Halfword
31	922	X	extsh.	B	Y		11:15	6:10			2	Extend Sign Halfword and Record
31	986	X	extsw	64	Y		11:15	6:10			2	Extend Sign Word
31	986	X	extsw.	64	Y		11:15	6:10			2	Extend Sign Word and Record
19	274	XL	hrfid	S	N							Hypervisor Return from Interrupt Doubleword
31	982	X	icbi	B	Y			11:15	16:20		3	Instruction Cache Block Invalidate
31	991	X	icbiep	E.PD	Y			11:15	16:20		3	Instruction Cache Block Invalidate by External PID

Table A-1. A2 Core Instructions by Mnemonic (Sheet 6 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
31	230	X	icblc	E.CL	Y			11:15	16:20		3	Instruction Cache Block Lock Clear
31	22	X	icbt	E	Y			11:15	16:20		3	Instruction Cache Block Touch
31	486	X	icbtls	E.CL	Y			11:15	16:20		3	Instruction Cache Block Touch and Lock Set
31	966	X	ici	E.CI	Y						3	Instruction Cache Invalidate
31	998	X	icread	E.CD	N			11:15	16:20			Instruction Cache Read
31	950	X	icswepx	Cop	Y			11:15	16:20	6:10	3	Initiate Coprocessor Store Word External PID Indexed
31	950	X	icswepx.	Cop	Y			11:15	16:20	6:10	7	Initiate Coprocessor Store Word External PID Indexed and Record
31	406	X	icswx	Cop	Y			11:15	16:20	6:10	3	Initiate Coprocessor Store Word Indexed
31	406	X	icswx.	Cop	Y			11:15	16:20	6:10	7	Initiate Coprocessor Store Word Indexed and Record
31	15	A	isel	B	Y		6:10	11:15	16:20		2	Integer Select
19	150	XL	isync	B	Y							Instruction Synchronize
31	52	X	lbarx	B	Y		6:10	11:15	16:20			Load Byte and Reserve Indexed
31	95	X	lbepx	E.PD	Y		6:10	11:15	16:20		3	Load Byte by External Process ID Indexed
34		D	lbz	B	Y		6:10	11:15			3	Load Byte and Zero
35		D	lbzu	B	Y	Y	6:10	11:15			3	Load Byte and Zero with Update
31	119	X	lbzux	B	Y	Y	6:10	11:15	16:20		3	Load Byte and Zero with Update Indexed
31	87	X	lbzx	B	Y		6:10	11:15	16:20		3	Load Byte and Zero Indexed
58	0	DS	ld	64	Y		6:10	11:15			3	Load Doubleword
31	84	X	ldarx	64	Y		6:10	11:15	16:20		7	Load Doubleword and Reserve Indexed
31	212	X	ldawx.		Y		6:10	11:15	16:20		7	Load Doubleword and Watch Indexed
31	532	X	ldbrx	64	Y		6:10	11:15	16:20		3	Load Double Byte and Reverse Indexed
31	29	X	ldepx	64	Y		6:10	11:15	16:20		3	Load Doubleword by External Process ID Indexed
58	1	DS	ldu	64	Y	Y	6:10	11:15			3	Load Doubleword with Update
31	53	X	ldux	64	Y	Y	6:10	11:15	16:20		3	Load Doubleword with Update Indexed
31	21	X	ldx	64	Y		6:10	11:15	16:20		3	Load Doubleword Indexed
42		D	lha	B	Y		6:10	11:15			3	Load Halfword Algebraic
31	116	X	lharx	B	Y		6:10	11:15	16:20			Load Halfword and Reserve Indexed
43		D	lhau	B	Y	Y	6:10	11:15			3	Load Halfword Algebraic with Update
31	375	X	lhaux	B	Y	Y	6:10	11:15	16:20		3	Load Halfword Algebraic with Update Indexed

Table A-1. A2 Core Instructions by Mnemonic (Sheet 7 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
31	343	X	lhax	B	Y		6:10	11:15	16:20		3	Load Halfword Algebraic Indexed
31	790	X	lhbrx	B	Y		6:10	11:15	16:20		3	Load Halfword Byte-Reverse Indexed
31	287	X	lhpepx	E.PD	Y		6:10	11:15	16:20		3	Load Halfword by External Process ID Indexed
40		D	lhz	B	Y		6:10	11:15			3	Load Halfword and Zero
41		D	lhzu	B	Y	Y	6:10	11:15			3	Load Halfword and Zero with Update
31	311	X	lhzux	B	Y	Y	6:10	11:15	16:20		3	Load Halfword and Zero with Update Indexed
31	279	X	lhzx	B	Y		6:10	11:15	16:20		3	Load Halfword and Zero Indexed
46		D	lmw	B	Y	Y	6:10	11:15			3	Load Multiple Word
56		DQ	lq	LSQ	N		6:10	11:15			3	Load Quadword
31	597	X	lswi	B	Y	Y	6:10	11:15			3	Load String Word Immediate
31	533	X	lswx	B	Y	Y	6:10	11:15	16:20		3	Load String Word Indexed
58	2	DS	lwa	64	Y		6:10	11:15			3	Load Word Algebraic
31	20	X	lwarx	B	Y		6:10	11:15	16:20		7	Load Word and Reserve Indexed
31	373	X	lwaux	64	Y	Y	6:10	11:15	16:20		3	Load Word Algebraic with Update Indexed
31	341	X	lwax	64	Y		6:10	11:15	16:20		3	Load Word Algebraic Indexed
31	534	X	lwbrx	B	Y		6:10	11:15	16:20		3	Load Word Byte-Reverse indexed
31	31	X	lwepx	E.PD	Y		6:10	11:15	16:20		3	Load Word by External Process ID Indexed
32		D	lwz	B	Y		6:10	11:15			3	Load Word and Zero
33		D	lwzu	B	Y	Y	6:10	11:15			3	Load Word and Zero with Update
31	55	X	lwzux	B	Y	Y	6:10	11:15	16:20		3	Load Word and Zero with update Indexed
31	23	X	lwzx	B	Y		6:10	11:15	16:20		3	Load Word and Zero Indexed
4	172	XO	macchw	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Modulo Signed
4	172	XO	macchw.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Modulo Signed and Record
4	172	XO	macchwo	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Modulo Signed with Overflow
4	172	XO	macchwo.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Modulo Signed with Record and Overflow
4	236	XO	macchws	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Saturate Signed
4	236	XO	macchws.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Saturate Signed and Record

Table A-1. A2 Core Instructions by Mnemonic (Sheet 8 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
4	236	XO	macchwso	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Saturate Signed with Overflow
4	236	XO	macchwso.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Saturate Signed with Record and Overflow
4	204	XO	macchwsu	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Saturate Unsigned
4	204	XO	macchwsu.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Saturate Unsigned and Record
4	204	XO	macchwsuo	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Saturate Unsigned with Overflow
4	204	XO	macchwsuo.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Saturate Unsigned with Record and Overflow
4	140	XO	macchwu	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Modulo Unsigned
4	140	XO	macchwu.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Modulo Unsigned and Record
4	140	XO	macchwuo	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Modulo Unsigned with Overflow
4	140	XO	macchwuo.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Cross Halfword to Word Modulo Unsigned with Overflow and Record
4	44	XO	machhw	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Signed
4	44	XO	machhw.	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Signed and Record
4	44	XO	machhwo	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Signed with Overflow
4	44	XO	machhwo.	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Signed with Overflow and Record
4	108	XO	machhws	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Saturate Signed
4	108	XO	machhws.	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Saturate Signed and Record
4	108	XO	machhwso	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Saturate Signed with Overflow
4	108	XO	machhwso.	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Saturate Signed with Record and Overflow
4	76	XO	machhwsu	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Saturate Unsigned
4	76	XO	machhwsu.	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Saturate Unsigned and Record

Table A-1. A2 Core Instructions by Mnemonic (Sheet 9 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
4	76	XO	machhwsuo	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Saturate Unsigned with Overflow
4	76	XO	machhwsuo.	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Saturate Unsigned with Record and Overflow
4	12	XO	machhwu	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Modulo Unsigned
4	12	XO	machhwu.	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Modulo Unsigned and Record
4	12	XO	machhwuo	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Modulo Unsigned with Overflow
4	12	XO	machhwuo.	LMA	N		6:10	11:15	16:20			Multiply Accumulate High Halfword to Word Modulo Unsigned with Record and Overflow
4	428	XO	maclhw	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Signed
4	428	XO	maclhw.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Signed and Record
4	428	XO	maclhwo	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Signed with Overflow
4	428	XO	maclhwo.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Signed with Record and Overflow
4	492	XO	maclhws	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Saturate Signed
4	492	XO	maclhws.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Saturate Signed and Record
4	492	XO	maclhwso	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Saturate Signed with Overflow
4	492	XO	maclhwso.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Saturate Signed with Record and Overflow
4	460	XO	maclhwsu	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Saturate Unsigned
4	460	XO	maclhwsu.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Saturate Unsigned and Record
4	460	XO	maclhwsuo	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Saturate Unsigned with Overflow
4	460	XO	maclhwsuo.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Saturate Unsigned with Record and Overflow
4	396	XO	maclhwu	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Modulo Unsigned
4	396	XO	maclhwu.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Modulo Unsigned and Record

Table A-1. A2 Core Instructions by Mnemonic (Sheet 10 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
4	396	XO	maclhwuo	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Modulo Unsigned with Overflow
4	396	XO	maclhwuo.	LMA	N		6:10	11:15	16:20			Multiply Accumulate Low Halfword to Word Modulo Unsigned with Record and Overflow
31	50	XL	makeitso	E	Y							Make In-Transit Stores Observable
31	854	AFX	mbar	E	Y						3	Memory Barrier
19	0	XL	mcrf	B	Y						3	Move Condition Register Field 
31	512	X	mcrxr	B	Y	Y						Move to Condition Register from <u>XER</u>
31	19	AFX	mfcr	B	Y		6:10					Move from Condition Register
31	323	AFX	mfocr	E	Y		6:10				3	Move from Device Control Register
31	291	X	mfocrux	E	Y		6:10		11:15		6	Move from Device Control Register User Indexed
31	259	X	mfocrx	E	Y		6:10		11:15		6	Move from Device Control Register Indexed
31	83	X	mfmsr	B	Y		6:10				6	Move from Machine State Register
31	19	AFX	mfocrf	B	Y		6:10				3	Move from One Condition Register Field
31	334	AFX	mfpmr	E.PM	N		6:10					Move from Performance Monitor Register
31	339	AFX	mfspir	B	Y		6:10				6	Move from Special Purpose Register
31	595	X	mfstr	S	N		6:10	12:15				Move from Segment Register
31	659	X	mfstrin	S	N		6:10		16:20			Move from Segment Register Indirect
31	371	AFX	mftrb	B	Y		6:10				6	Move from Time Base
31	238	X	msgclr	E.PC	Y				16:20		7	Message Clear
31	206	X	msgsnd	E.PC	Y				16:20		3	Message Send
31	144	AFX	mtcrf	B	Y			6:10			3	Move to Condition Register Fields
31	451	AFX	mtocr	E	Y			6:10			7	Move to Device Control Register
31	419	X	mtocrux	E	Y			6:10	11:15		7	Move to Device Control Register User Indexed
31	387	X	mtocrx	E	Y			6:10	11:15		7	Move to Device Control Register Indexed
31	146	X	mtmsr	E	Y			6:10			7	Move to Machine State Register
31	178	X	mtmsrd	S	N			6:10				Move to Machine State Register Doubleword
31	144	AFX	mtocrf	B	Y			6:10			3	Move to One Condition Register Field
31	462	AFX	mtpmr	E.PM	N			6:10				Move to Performance Monitor Register
31	467	AFX	mtspir	B	Y			6:10			6	Move to Special Purpose Register
31	210	X	mtstr	S	N			6:10				Move to Segment Register

Table A-1. A2 Core Instructions by Mnemonic (Sheet 11 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
31	242	X	mtsrin	S	N			6:10	16:20			Move to Segment Register Indirect
4	168	X	mulchw	LMA	N		6:10	11:15	16:20			Multiply Cross Halfword to Word Signed
4	168	X	mulchw.	LMA	N		6:10	11:15	16:20			Multiply Cross Halfword to Word Signed and Record
4	136	X	mulchwu	LMA	N		6:10	11:15	16:20			Multiply Cross Halfword to Word Unsigned
4	136	X	mulchwu.	LMA	N		6:10	11:15	16:20			Multiply Cross Halfword to Word Unsigned and Record
31	73	XO	mulhd	64	Y		6:10	11:15	16:20		8	Multiply High Doubleword
31	73	XO	mulhd.	64	Y		6:10	11:15	16:20		8	Multiply High Doubleword and Record
31	9	XO	mulhdu	64	Y		6:10	11:15	16:20		8	Multiply High Doubleword Unsigned
31	9	XO	mulhdu.	64	Y		6:10	11:15	16:20		8	Multiply High Doubleword Unsigned and Record
4	40	X	mulhhw	LMA	N		6:10	11:15	16:20			Multiply High Halfword to Word Signed
4	40	X	mulhhw.	LMA	N		6:10	11:15	16:20			Multiply High Halfword to Word Signed and Record
4	8	X	mulhhwu	LMA	N		6:10	11:15	16:20			Multiply High Halfword to Word Unsigned
4	8	X	mulhhwu.	LMA	N		6:10	11:15	16:20			Multiply High Halfword to Word Unsigned and Record
31	75	XO	mulhw	B	Y		6:10	11:15	16:20		5	Multiply High Word
31	75	XO	mulhw.	B	Y		6:10	11:15	16:20		5	Multiply High Word and Record
31	11	XO	mulhwu	B	Y		6:10	11:15	16:20		5	Multiply High Word Unsigned
31	11	XO	mulhwu.	B	Y		6:10	11:15	16:20		5	Multiply High Word Unsigned and Record
31	233	XO	mulld	64	Y		6:10	11:15	16:20		7	Multiply Low Doubleword
31	233	XO	mulld.	64	Y		6:10	11:15	16:20		7	Multiply Low Doubleword and Record
31	233	XO	mulldo	64	Y		6:10	11:15	16:20		7	Multiply Low Doubleword with Overflow
31	233	XO	mulldo.	64	Y		6:10	11:15	16:20		7	Multiply Low Doubleword with Overflow and Record
4	424	X	mulllhw	LMA	N		6:10	11:15	16:20			Multiply Low Halfword to Word Signed
4	424	X	mulllhw.	LMA	N		6:10	11:15	16:20			Multiply Low Halfword to Word Signed and Record
4	392	X	mulllhwu	LMA	N		6:10	11:15	16:20			Multiply Low Halfword to Word Unsigned
4	392	X	mulllhwu.	LMA	N		6:10	11:15	16:20			Multiply Low Halfword to Word Unsigned and Record
7		D	mulli	B	Y		6:10	11:15			6	Multiply Low Immediate
31	235	XO	mulllw	B	Y		6:10	11:15	16:20		5	Multiply Low Word

Table A-1. A2 Core Instructions by Mnemonic (Sheet 12 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
31	235	XO	mullw.	B	Y		6:10	11:15	16:20		5	Multiply Low Word and Record
31	235	XO	mullwo	B	Y		6:10	11:15	16:20		5	Multiply Low Word with Overflow
31	235	XO	mullwo.	B	Y		6:10	11:15	16:20		5	Multiply Low Word with Overflow and Record
31	476	X	nand	B	Y		11:15	6:10	16:20		2	NAND
31	476	X	nand.	B	Y		11:15	6:10	16:20		2	NAND and Record
19	434	XL	nap	S	N							Nap
31	104	XO	neg	B	Y		6:10	11:15			2	Negate
31	104	XO	neg.	B	Y		6:10	11:15			2	Negate and Record
31	104	XO	nego	B	Y		6:10	11:15			2	Negate with Overflow
31	104	XO	nego.	B	Y		6:10	11:15			2	Negate with Overflow and Record
4	174	XO	nmacchw	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Cross Halfword to Word Modulo Signed
4	174	XO	nmacchw.	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Cross Halfword to Word Modulo Signed and Record
4	174	XO	nmacchwo	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Cross Halfword to Word Modulo Signed with Overflow
4	174	XO	nmacchwo.	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Cross Halfword to Word Modulo Signed with Record and Overflow
4	238	XO	nmacchws	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Cross Halfword to Word Saturate Signed
4	238	XO	nmacchws.	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Cross Halfword to Word Saturate Signed and Record
4	238	XO	nmacchwso	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Cross Halfword to Word Saturate Signed with Overflow
4	238	XO	nmacchwso.	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Cross Halfword to Word Saturate Signed with Record and Overflow
4	46	XO	nmachhw	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate High Halfword to Word Modulo Signed
4	46	XO	nmachhw.	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate High Halfword to Word Modulo Signed and Record
4	46	XO	nmachhwo	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate High Halfword to Word Modulo Signed with Overflow
4	46	XO	nmachhwo.	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate High Halfword to Word Modulo Signed with Record and Overflow

Table A-1. A2 Core Instructions by Mnemonic (Sheet 13 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
4	110	XO	nmachhws	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate High Halfword to Word Saturate Signed
4	110	XO	nmachhws.	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate High Halfword to Word Saturate Signed and Record
4	110	XO	nmachhwsO	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate High Halfword to Word Saturate Signed with Overflow
4	110	XO	nmachhwsO.	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate High Halfword to Word Saturate Signed with Record and Overflow
4	430	XO	nmachlw	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Low Halfword to Word Modulo Signed
4	430	XO	nmachlw.	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Low Halfword to Word Modulo Signed and Record
4	430	XO	nmachlwo	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Low Halfword to Word Modulo Signed with Overflow
4	430	XO	nmachlwo.	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Low Halfword to Word Modulo Signed with Record and Overflow
4	494	XO	nmachhws	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Low Halfword to Word Saturate Signed
4	494	XO	nmachhws.	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Low Halfword to Word Saturate Signed and Record
4	494	XO	nmachhwsO	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Low Halfword to Word Saturate Signed with Overflow
4	494	XO	nmachhwsO.	LMA	N		6:10	11:15	16:20			Negative Multiply Accumulate Low Halfword to Word Saturate Signed with Record and Overflow
31	124	X	nor	B	Y		11:15	6:10	16:20		2	NOR
31	124	X	nor.	B	Y		11:15	6:10	16:20		2	NOR and Record
31	444	X	or	B	Y		11:15	6:10	16:20		2	OR
31	444	X	or.	B	Y		11:15	6:10	16:20		2	OR and Record
31	412	X	orc	B	Y		11:15	6:10	16:20		2	OR with Complement
31	412	X	orc.	B	Y		11:15	6:10	16:20		2	OR with Complement and Record
24		D	ori	B	Y		11:15	6:10			2	OR Immediate
25		D	oris	B	Y		11:15	6:10			2	OR Immediate Shifted
31	122	X	popcntb	B	Y	Y	11:15	6:10				Population Count Bytes
31	506	X	popcntd	64	Y	Y	11:15	6:10				Population Count Doubleword

Table A-1. A2 Core Instructions by Mnemonic (Sheet 14 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
31	378	X	popcntw	B	Y	Y	11:15	6:10				Population Count Words
31	186	X	prtyd	64	Y	Y	11:15	6:10				Parity Doubleword
31	154	X	prtyw	B	Y	Y	11:15	6:10				Parity Word
31	530	X	reserved	B	Y						0	Reserved Nop
31	562	X	reserved	B	Y						0	Reserved Nop
31	594	X	reserved	B	Y						0	Reserved Nop
31	626	X	reserved	B	Y						0	Reserved Nop
31	658	X	reserved	B	Y						0	Reserved Nop
31	690	X	reserved	B	Y						0	Reserved Nop
31	722	X	reserved	B	Y						0	Reserved Nop
31	754	X	reserved	B	Y						0	Reserved Nop
19	51	XL	rfdi	E	Y						6	Return from Critical Interrupt
19	39	X	rfdi	E.ED	N							Return from Debug Interrupt
19	102	XL	rfgi	E.HV	Y						6	Return From Guest Interrupt
19	50	XL	rfdi	E	Y						6	Return from Interrupt
19	18	XL	rfdi	S	N							Return from Interrupt Doubleword
19	38	XL	rfmci	E	Y						6	Return from Machine Check Interrupt
30	8	MDS	rldcl	64	Y		11:15	6:10	16:20		2	Rotate Left Doubleword then Clear Left
30	8	MDS	rldcl.	64	Y		11:15	6:10	16:20		2	Rotate Left Doubleword then Clear Left and Record
30	9	MDS	rldcr	64	Y		11:15	6:10	16:20		2	Rotate Left Doubleword then Clear Right
30	9	MDS	rldcr.	64	Y		11:15	6:10	16:20		2	Rotate Left Doubleword then Clear Right and Record
30	2	MD	rldic	64	Y		11:15	6:10			2	Rotate Left Doubleword Immediate then Clear
30	2	MD	rldic.	64	Y		11:15	6:10			2	Rotate Left Doubleword Immediate then Clear and Record
30	0	MD	rldicl	64	Y		11:15	6:10			2	Rotate Left Doubleword Immediate then Clear Left
30	0	MD	rldicl.	64	Y		11:15	6:10			2	Rotate Left Doubleword Immediate then Clear Left and Record
30	1	MD	rldicr	64	Y		11:15	6:10			2	Rotate Left Doubleword Immediate then Clear Right
30	1	MD	rldicr.	64	Y		11:15	6:10			2	Rotate Left Doubleword Immediate then Clear Right and Record
30	3	MD	rldimi	64	Y		11:15	6:10	11:15		2	Rotate Left Doubleword Immediate then Mask Insert

Table A-1. A2 Core Instructions by Mnemonic (Sheet 15 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
30	3	MD	rldimi.	64	Y		11:15	6:10	11:15		2	Rotate Left Doubleword Immediate then Mask Insert and Record
20		M	rlwimi	B	Y		11:15	6:10	11:15		2	Rotate Left Word Immediate then Mask Insert
20		M	rlwimi.	B	Y		11:15	6:10	11:15		2	Rotate Left Word Immediate then Mask Insert and Record
21		M	rlwinm	B	Y		11:15	6:10			2	Rotate Left Word Immediate then AND with Mask
21		M	rlwinm.	B	Y		11:15	6:10			2	Rotate Left Word Immediate then AND with Mask and Record
23		M	rlwnm	B	Y		11:15	6:10	16:20		2	Rotate Left Word then AND with Mask
23		M	rlwnm.	B	Y		11:15	6:10	16:20		2	Rotate Left Word then AND with Mask and Record
19	498	XL	rwinkle	S	N							Rip Van Winkle
17	1	SC	sc	B	Y						7	System Call
31	498	X	slbia	S	N							SLB Invalidate All
31	434	X	slbie	S	N			16:20				SLB Invalidate Entry
31	915	X	slbmfee	S	N		6:10	16:20				SLB Move From Entry
31	851	X	slbmfev	S	N		6:10	16:20				SLB Move From Entry <u>VSID</u>
31	402	X	slbmte	S	N		6:10	16:20				SLB Move to Entry
31	27	X	sld	64	Y		11:15	6:10	16:20		2	Shift Left Doubleword
31	27	X	sld.	64	Y		11:15	6:10	16:20		2	Shift Left Doubleword and Record
19	466	XL	sleep	S	N						6	Sleep
31	24	X	slw	B	Y		11:15	6:10	16:20		2	Shift Left Word
31	24	X	slw.	B	Y		11:15	6:10	16:20		2	Shift Left Word and Record
31	794	X	srad	64	Y		11:15	6:10	16:20		2	Shift Right Algebraic Doubleword
31	794	X	srad.	64	Y		11:15	6:10	16:20		2	Shift Right Algebraic Doubleword and Record
31	413	XS	sradi	64	Y		11:15	6:10			2	Shift Right Algebraic Doubleword Immediate
31	413	XS	sradi.	64	Y		11:15	6:10			2	Shift Right Algebraic Doubleword Immediate and Record
31	792	X	sraw	B	Y		11:15	6:10	16:20		2	Shift Right Algebraic Word
31	792	X	sraw.	B	Y		11:15	6:10	16:20		2	Shift Right Algebraic Word and Record
31	824	X	srawi	B	Y		11:15	6:10			2	Shift Right Algebraic Word Immediate
31	824	X	srawi.	B	Y		11:15	6:10			2	Shift Right Algebraic Word Immediate and Record
31	539	X	srd	64	Y		11:15	6:10	16:20		2	Shift Right Doubleword
31	539	X	srd.	64	Y		11:15	6:10	16:20		2	Shift Right Doubleword and Record

Table A-1. A2 Core Instructions by Mnemonic (Sheet 16 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
31	536	X	srw	B	Y		11:15	6:10	16:20		2	Shift Right Word
31	536	X	srw.	B	Y		11:15	6:10	16:20		2	Shift Right Word and Record
38		D	stb	B	Y			11:15		6:10	3	Store Byte
31	694	X	stbcx.	B	Y			11:15	16:20	6:10		Store Byte Conditional Indexed
31	223	X	stbepx	E.PD	Y			11:15	16:20	6:10	3	Store Byte by External Process ID Indexed
39		D	stbu	B	Y		11:15	11:15		6:10	3	Store Byte with Update
31	247	X	stbux	B	Y		11:15	11:15	16:20	6:10	3	Store Byte with Update Indexed
31	215	X	stbx	B	Y			11:15	16:20	6:10	3	Store Byte Indexed
62	0	DS	std	64	Y			11:15		6:10	3	Store Doubleword
31	660	X	stdbrx	64	Y			11:15	16:20	6:10	3	Store Double Byte and Reverse Indexed
31	214	X	stdcx.	64	Y			11:15	16:20	6:10	7	Store Doubleword Conditional Indexed
31	157	X	stdepX	64	Y			11:15	16:20	6:10	3	Store Doubleword by External Process ID Indexed
62	1	DS	stdu	64	Y		11:15	11:15		6:10	3	Store Doubleword with Update
31	181	X	stdux	64	Y		11:15	11:15	16:20	6:10	3	Store Doubleword with Update Indexed
31	149	X	stdx	64	Y			11:15	16:20	6:10	3	Store Doubleword Indexed
44		D	sth	B	Y			11:15		6:10	3	Store Halfword
31	726	X	sthcx.	B	Y			11:15	16:20	6:10		Store Halfword Conditional Indexed
31	918	X	sthbrx	B	Y			11:15	16:20	6:10	3	Store Halfword Byte-Reverse Indexed
31	415	X	sthepx	E.PD	Y			11:15	16:20	6:10	3	Store Halfword by External Process ID Indexed
45		D	sthu	B	Y		11:15	11:15		6:10	3	Store Halfword with Update
31	439	X	sthux	B	Y		11:15	11:15	16:20	6:10	3	Store Halfword with Update Indexed
31	407	X	sthx	B	Y			11:15	16:20	6:10	3	Store Halfword Indexed
47		D	stmw	B	Y	Y		11:15		6:10	3	Store Multiple Word
62		DS	stq	LSQ	N			11:15		6:10		Store Quadword
31	725	X	stswi	B	Y	Y		11:15		6:10	3	Store String Word Immediate
31	661	X	stswx	B	Y	Y		11:15	16:20	6:10	3	Store String Word Indexed
36		D	stw	B	Y			11:15		6:10	3	Store Word
31	662	X	stwbrx	B	Y			11:15	16:20	6:10	3	Store Word Byte-Reverse Indexed
31	150	X	stwcx.	B	Y			11:15	16:20	6:10	7	Store Word Conditional Indexed
31	159	X	stwepx	E.PD	Y			11:15	16:20	6:10	3	Store Word by External Process ID Indexed
37		D	stwu	B	Y		11:15	11:15		6:10	3	Store Word with Update

Table A-1. A2 Core Instructions by Mnemonic (Sheet 17 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
31	183	X	stwux	B	Y		11:15	11:15	16:20	6:10	3	Store Word with Update Indexed
31	151	X	stw	B	Y			11:15	16:20	6:10	3	Store Word Indexed
31	40	XO	subf	B	Y		6:10	11:15	16:20		1	Subtract From
31	40	XO	subf.	B	Y		6:10	11:15	16:20		2	Subtract From and Record
31	8	XO	subfc	B	Y		6:10	11:15	16:20		2	Subtract From Carrying
31	8	XO	subfc.	B	Y		6:10	11:15	16:20		2	Subtract From Carrying and Record
31	8	XO	subfco	B	Y		6:10	11:15	16:20		2	Subtract From Carrying with Overflow
31	8	XO	subfco.	B	Y		6:10	11:15	16:20		2	Subtract From Carrying with Overflow and Record
31	136	XO	subfe	B	Y		6:10	11:15	16:20		2	Subtract From Extended
31	136	XO	subfe.	B	Y		6:10	11:15	16:20		2	Subtract From Extended and Record
31	136	XO	subfeo	B	Y		6:10	11:15	16:20		2	Subtract From Extended with Overflow
31	136	XO	subfeo.	B	Y		6:10	11:15	16:20		2	Subtract From Extended with Overflow and Record
8		D	subfic	B	Y		6:10	11:15			2	Subtract From Immediate Carrying
31	232	XO	subfme	B	Y		6:10	11:15			2	Subtract From Minus One Extended
31	232	XO	subfme.	B	Y		6:10	11:15			2	Subtract From Minus One Extended and Record
31	232	XO	subfmeo	B	Y		6:10	11:15			2	Subtract From Minus One Extended with Overflow
31	232	XO	subfmeo.	B	Y		6:10	11:15			2	Subtract From Minus One Extended with Overflow and Record
31	40	XO	subfo	B	Y		6:10	11:15	16:20		2	Subtract From with Overflow
31	40	XO	subfo.	B	Y		6:10	11:15	16:20		2	Subtract From with Overflow and Record
31	200	XO	subfze	B	Y		6:10	11:15			2	Subtract From Zero Extended
31	200	XO	subfze.	B	Y		6:10	11:15			2	Subtract From Zero Extended and Record
31	200	XO	subfzeo	B	Y		6:10	11:15			2	Subtract From Zero Extended with Overflow
31	200	XO	subfzeo.	B	Y		6:10	11:15			2	Subtract From Zero Extended with Overflow and Record
31	598	X	sync	B	Y						3	Synchronize
31	68	X	td	64	Y			11:15	16:20		0	Trap Doubleword
2		D	tdi	64	Y			11:15			0	 Doubleword Immediate
31	370	X	tlbia	S	N							TLB Invalidate All
31	306	X	tlbie	S	N			6:10	16:20			TLB Invalidate Entry
1F	313	X	tlbiel	S	N			6:10	16:20			TLB Invalidate Entry Local

Table A-1. A2 Core Instructions by Mnemonic (Sheet 18 of 18)

Primary	Extended	Form	Mnemonic	Category	Implemented	Microcoded	Target 1 Bits	Source 1 Bits	Source 2 Bits	Source 3 Bits	Latency: Throughput	Instruction Description
31	18	X	tlbilx	E.MF	Y			11:15	16:20		6	TLB Invalidate Local Indexed
31	786	X	tlbivax	E.MF	Y			11:15	16:20		6	TLB Invalidate Virtual Address Indexed
31	946	X	tlbre	E.MF	Y						6	TLB Read Entry
31	850	X	tlbsrx.	E.MF	Y			11:15	16:20		6	TLB Search and Reserve Indexed and Record
31	914	X	tlbsx	E.MF	Y			11:15	16:20		6	TLB Search Indexed
31	914	X	tlbsx.	E.MF	Y			11:15	16:20		6	TLB Search Indexed and Record
31	566	X	tlbsync	B	Y						3	TLB Synchronize
31	978	X	tlbwe	E.MF	Y						6	TLB Write Entry
31	4	X	tw	B	Y			11:15	16:20		0	Trap Word
3		D	twi	B	Y			11:15			0	Trap Word Immediate
31	62	X	wait	WT	Y							Wait
31	902	X	wchkall		Y						3	Watch Check All
31	934	X	wclr		Y			11:15	16:20		3	Watch  r
31	131	X	wrttee	E	Y			6:10			6	Write <u>MSR</u> External Enable
31	163	X	wrttei	E	Y						6	Write MSR External Enable Immediate
31	316	X	xor	B	Y		11:15	6:10	16:20		2	XOR
31	316	X	xor.	B	Y		11:15	6:10	16:20		2	XOR and Record
26		D	xori	B	Y		11:15	6:10			2	XOR Immediate
27		D	xoris	B	Y		11:15	6:10			2	XOR Immediate Shifted



Appendix B. FU Instruction Summary

This appendix contains floating-point unit instructions summarized alphabetically and by opcode.

FU Instructions Sorted by Opcode lists all A2 processor instructions, sorted by primary and secondary opcodes. Extended mnemonics are not included in the opcode list.

Instruction Formats illustrates the A2 processor instruction forms (allowed arrangements of fields within instructions).

B.1 FU Instructions Sorted by Opcode

All instructions are 4 bytes long and word aligned. All instructions have a primary opcode field in bits 0:5. Some instructions also have a secondary opcode field. A2 FPU FU instructions, sorted by primary and secondary opcode, are listed in *Table B-1*.

The Form column in *Table B-1* refers to the arrangement of valid field combinations within the 4-byte instruction.

The A, X, D, and XFL instruction formats are described in PowerISA Version 2.06 Revision B.

Table B-1. FU Instructions by Opcode (Sheet 1 of 5)

Primary	Extended	Form	Mnemonic	Ucode	Latency: Throughput	Instruction Description
63	264	X	fabs		6	Floating Absolute Value
63	264	X	fabs.		6:4	Floating Absolute Value and record CR1
63	21	A	fadd		6	Floating Add
63	21	A	fadd.		6:4	Floating Add and record CR1
59	21	A	fadds		6	Floating Add Single
59	21	A	fadds.		6:4	Floating Add Single and record CR1
63	846	X	fcfid		6	Floating Convert From Integer Doubleword
63	846	X	fcfid.		6:4	Floating Convert From Integer Doubleword and record CR1
63	974	X	fcfidu		6	Floating Convert From Integer Doubleword Unsigned
63	974	X	fcfidu.		6:4	Floating Convert From Integer Doubleword Unsigned and record CR1
59	846	X	fcfids		6	Floating Convert From Integer Doubleword Single
59	846	X	fcfids.		6:4	Floating Convert From Integer Doubleword Single and record CR1
59	974	X	fcfidus		6	Floating Convert From Integer Doubleword Unsigned Single
59	974	X	fcfidus.		6:4	Floating Convert From Integer Doubleword Unsigned Single and record CR1
63	32	X	fcmpo		5	Floating Compare Ordered
63	0	X	fcmpu		5	Floating Compare Unordered
63	8	X	fcpsgn		6	Floating Copy Sign
63	8	X	fcpsgn.		6:4	Floating Copy Sign and record CR1

Table B-1. FU Instructions by Opcode (Sheet 2 of 5)

Primary	Extended	Form	Mnemonic	Ucode	Latency: Throughput	Instruction Description
63	814	X	fctid		6	Floating Convert to Integer Doubleword
63	814	X	fctid.		6:4	Floating Convert To Integer Doubleword and record CR1
63	942	X	fctidu		6	Floating Convert to Integer Doubleword Unsigned
63	942	X	fctidu.		6:4	Floating Convert To Integer Doubleword Unsigned and record CR1
63	815	X	fctidz		6	Floating Convert To Integer Doubleword with round toward Zero
63	815	X	fctidz.		6:4	Floating Convert To Integer Doubleword with round toward Zero and record CR1
63	943	X	fctiduz		6	Floating Convert To Integer Doubleword Unsigned with round toward Zero
63	943	X	fctiduz.		6:4	Floating Convert To Integer Doubleword Unsigned with round toward Zero and record CR1
63	14	X	fctiw		6	Floating Convert To Integer Word
63	14	X	fctiw.		6:4	Floating Convert To Integer Word and record CR1
63	142	X	fctiwu		6	Floating Convert To Integer Word Unsigned
63	142	X	fctiwu.		6:4	Floating Convert To Integer Word Unsigned and record CR1
63	15	X	fctiwz		6	Floating Convert To Integer Word with round toward Zero
63	15	X	fctiwz.		6:4	Floating Convert To Integer Word with round to Zero and record CR1
63	143	X	fctiwuz		6	Floating Convert To Integer Word Unsigned with round toward Zero
63	143	X	fctiwuz.		6:4	Floating Convert To Integer Word Unsigned with round to Zero and record CR1
63	18	A	fdiv	Y	72:72	Floating Divide
63	18	A	fdiv.	Y	75:75	Floating Divide and record CR1
59	18	A	fdivs	Y	59:59	Floating Divide Single
59	18	A	fdivs.	Y	62:62	Floating Divide Single and record CR1
63	29	A	fmadd		6	Floating Multiply-Add
63	29	A	fmadd.		6:4	Floating Multiply-Add and record CR1
59	29	A	fmadds		6	Floating Multiply-Add Single
59	29	A	fmadds.		6:4	Floating Multiply-Add Single and record CR1
63	72	X	fmr		6	Floating Move Register
63	72	X	fmr.		6:4	Floating Move Register and record CR1
63	28	A	fmsub		6	Floating Multiply-Subtract
63	28	A	fmsub.		6:4	Floating Multiply-Subtract and record CR1
59	28	A	fmsubs		6	Floating Multiply-Subtract Single
59	28	A	fmsubs.		6:4	Floating Multiply-Subtract Single and record CR1
63	25	A	fmul		6	Floating Multiply
63	25	A	fmul.		6:4	Floating Multiply and record CR1
59	25	A	fmuls		6	Floating Multiply Single
59	25	A	fmuls.		6:4	Floating Multiply Single and record CR1

Table B-1. FU Instructions by Opcode (Sheet 3 of 5)

Primary	Extended	Form	Mnemonic	Ucode	Latency: Throughput	Instruction Description
63	136	X	fnabs		6	Floating Negative Absolute
63	136	X	fnabs.		6:4	Floating Negative Absolute Value and record CR1
63	40	X	fneg		6	Floating Negate
63	40	X	fneg.		6:4	Floating Negate and record CR1
63	31	A	fnmadd		6	Floating Negative Multiply-Add
63	31	A	fnmadd.		6:4	Floating Negative Multiply-Add and record CR1
59	31	A	fnmadds		6	Floating Negative Multiply-Add Single
59	31	A	fnmadds.		6:4	Floating Negative Multiply-Add Single and record CR1
63	30	A	fnmsub		6	Floating Negative Multiply-Subtract
63	30	A	fnmsub.		6:4	Floating Negative Multiply-Subtract and record CR1
59	30	A	fnmsubs		6	Floating Negative Multiply-Subtract Single
59	30	A	fnmsubs.		6:4	Floating Negative Multiply-Subtract Single and record CR1
63	24	A	fre		6	Floating Reciprocal Estimate
63	24	A	fre.		6:4	Floating Reciprocal Estimate and record CR1
59	24	A	fres		6	Floating Reciprocal Estimate Single
59	24	A	fres.		6:4	Floating Reciprocal Estimate Single and record CR1
63	488	X	frim		6	Floating Round To Integer Minus
63	488	X	frim.		6:4	Floating Round To Integer Minus and record CR1
63	392	X	frin		6	Floating Round To Integer Nearest
63	392	X	frin.		6:4	Floating Round To Integer Nearest and record CR1
63	456	X	frip		6	Floating Round To Integer Plus
63	456	X	frip.		6:4	Floating Round To Integer Plus and record CR1
63	424	X	friz		6	Floating Round To Integer toward Zero
63	424	X	friz.		6:4	Floating Round To Integer toward Zero and record CR1
63	12	X	frsp		6	Floating Round to Single Precision
63	12	X	frsp.		6:4	Floating Round to Single-Precision and record CR1
63	26	A	frsqrte		6	Floating Reciprocal Square Root Estimate
63	26	A	frsqrte.		6:4	Floating Reciprocal Square Root Estimate and record CR1
59	26	A	frsqrtes		6	Floating Reciprocal Square Root Estimate Single
59	26	A	frsqrtes.		6:4	Floating Reciprocal Square Root Estimate Single and record CR1
63	23	A	fsel		6	Floating Select
63	23	A	fsel.		6:4	Floating Select and record CR1
63	22	A	fsqrt	Y	69:69	Floating Square Root
63	22	A	fsqrt.	Y	72:72	Floating Square Root and record CR1
59	22	A	fsqrts	Y	65:65	Floating Square Root Single

Table B-1. FU Instructions by Opcode (Sheet 4 of 5)

Primary	Extended	Form	Mnemonic	Ucode	Latency: Throughput	Instruction Description
59	22	A	fsqrts.	Y	68:68	Floating Square Root Single and record CR1
63	20	A	fsub		6	Floating Subtract
63	20	A	fsub.		6:4	Floating Subtract and record CR1
59	20	A	fsubs		6	Floating Subtract Single
59	20	A	fsubs.		6:4	Floating Subtract Single and record CR1
63	128	X	ftdiv		5	Floating Test for software Divide
63	160	X	ftsqrt		5	Floating Test for software Square Root
50		D	lfd		7	Load Floating-Point Double
31	607	X	lfdepX		7	Load Floating-Point Double by External Process ID Indexed
51		D	lfdu		7	Load Floating-Point Double with Update
31	631	X	lfdux		7	Load Floating-Point Double with Update Indexed
31	599	X	lfdx		7	Load Floating-Point Double Indexed
31	855	X	lfiwax		7	Load Floating-Point as Integer Word Algebraic Indexed
31	887	X	lfiwzx		7	Load Floating-Point as Integer Word and Zero Indexed
48		D	lfs		7	Load Floating-Point Single
49		D	lfsu		7	Load Floating-Point Single with Update
31	567	X	lfsux		7	Load Floating-Point Single with Update Indexed
31	535	X	lfsx		7	Load Floating-Point Single Indexed
63	64	X	mcrfs		8:4	Move to Condition Register from <u>FPSCR</u>
63	583	X	mffs		6	Move From FPSCR
63	583	X	mffs.		6:4	Move From FPSCR and record CR1
63	70	X	mtfsb0		6	Move To FPSCR Bit 0
63	70	X	mtfsb0.		6:4	Move To FPSCR Bit 0 and record CR1
63	38	X	mtfsb1		6	Move To FPSCR Bit 1
63	38	X	mtfsb1.		6:4	Move To FPSCR Bit 1 and record CR1
63	711	XFL	mtfsf		6	Move To FPSCR Fields
63	711	XFL	mtfsf.		6:4	Move To FPSCR Fields and record CR1
63	134	X	mtfsfi		6	Move to FPSCR Field Immediate
63	134	X	mtfsfi.		6:4	Move To FPSCR Field Immediate and record CR1
54		D	stfd		1	Store Floating-Point Double
31	735		stfdepX		1	Store Floating-Point Double by External Process ID Indexed
55		D	stfdu		1	Store Floating-Point Double with Update
31	759	X	stfdx		1	Store Floating-Point Double with Update Indexed
31	727	X	stfdx		1	Store Floating-Point Double Indexed
31	983	X	stfiwx		1	Store Floating-Point as Integer Word Indexed

Table B-2.

6
7
7
7
7
6
3
6
6
7
3
3
7
7
7
3
7
3
7
7
3
6
8
8
8
8

Table B-2.

6
6
2
2
2
2



Appendix C. Debug and Trigger Groups

Note: The current A2o plan is to implement a 64 bit debug bus and 12 bit trigger bus. Specific details of unit debug/trigger bus requirements (such as debug mux quantity and placement, or debug group assignments), have not been determined. This section will be updated as details of the A2O core design changes are identified.

C.1 Unit Debug Mux Component

TBD

C.2 Debug Mux Component Ordering on the Ramp Bus

TBD

C.3 Example Debug Mux Configuration Settings

TBD

C.4 Debug Select Registers and Debug Group Tables

TBD

